## Contents

Preface ........................................................................................................................................ 9

1 Floating Point Unit (FPU) ........................................................................................................ 11
  1.1 Overview .......................................................................................................................... 12
    1.1.1 Compatibility with the C28x Fixed-Point CPU ......................................................... 12
  1.2 Components of the C28x plus Floating-Point CPU ....................................................... 13
    1.2.1 Emulation Logic ....................................................................................................... 14
    1.2.2 Memory Map ........................................................................................................... 14
    1.2.3 On-Chip Program and Data ....................................................................................... 14
    1.2.4 CPU Interrupt Vectors ............................................................................................ 14
    1.2.5 Memory Interface .................................................................................................... 14
  1.3 CPU Register Set .............................................................................................................. 15
    1.3.1 CPU Registers ......................................................................................................... 15
  1.4 Pipeline ............................................................................................................................ 21
    1.4.1 Pipeline Overview ..................................................................................................... 21
    1.4.2 General Guidelines for Floating-Point Pipeline Alignment ...................................... 22
    1.4.3 Moves from FPU Registers to C28x Registers ....................................................... 23
    1.4.4 Moves from C28x Registers to FPU Registers ...................................................... 24
    1.4.5 Parallel Instructions .................................................................................................. 25
    1.4.6 Invalid Delay Instructions ......................................................................................... 25
    1.4.7 Optimizing the Pipeline ............................................................................................ 28
  1.5 Floating Point Unit Instruction Set .................................................................................... 29
    1.5.1 Instruction Descriptions ............................................................................................ 29
    1.5.2 Instructions ............................................................................................................... 32

2 Floating Point Unit (FPU64) .................................................................................................... 143
  2.1 Overview .......................................................................................................................... 144
    2.1.1 Compatibility with the C28x Fixed-Point CPU ......................................................... 144
  2.2 Components of the C28x plus Floating-Point CPU (FPU64) ............................................ 145
    2.2.1 Emulation Logic ...................................................................................................... 146
    2.2.2 Memory Map ........................................................................................................... 146
    2.2.3 On-Chip Program and Data ....................................................................................... 146
    2.2.4 CPU Interrupt Vectors ............................................................................................ 146
    2.2.5 Memory Interface .................................................................................................... 147
  2.3 CPU Register Set .............................................................................................................. 148
    2.3.1 CPU Registers ......................................................................................................... 148
  2.4 Pipeline ............................................................................................................................ 154
    2.4.1 Pipeline Overview ..................................................................................................... 154
    2.4.2 General Guidelines for Floating-Point Pipeline Alignment ...................................... 155
    2.4.3 Moves from FPU Registers to C28x Registers ....................................................... 156
    2.4.4 Moves from C28x Registers to FPU Registers ...................................................... 157
    2.4.5 Parallel Instructions .................................................................................................. 157
    2.4.6 Invalid Delay Instructions ......................................................................................... 158
    2.4.7 Optimizing the Pipeline ............................................................................................ 161
  2.5 Floating Point Unit (FPU64) Instruction Set ..................................................................... 162
    2.5.1 Instruction Descriptions ............................................................................................ 162
    2.5.2 Instructions ............................................................................................................... 165
3 Viterbi, Complex Math and CRC Unit (VCU) .......................................................... 338
  3.1 Overview ........................................................................................................... 339
  3.2 Components of the C28x plus VCU ................................................................. 340
  3.3 Emulation Logic ............................................................................................... 341
     3.3.1 Memory Map ............................................................................................... 341
     3.3.2 CPU Interrupt Vectors ............................................................................... 342
     3.3.3 Memory Interface ....................................................................................... 342
     3.3.4 Address and Data Buses ........................................................................... 342
     3.3.5 Alignment of 32-Bit Accesses to Even Addresses ...................................... 342
  3.4 Register Set ....................................................................................................... 344
     3.4.1 VCU Register Set ....................................................................................... 344
     3.4.2 VCU Status Register (VSTATUS) ............................................................. 346
     3.4.3 Repeat Block Register (RB) ...................................................................... 349
  3.5 Pipeline ............................................................................................................ 351
     3.5.1 Pipeline Overview ....................................................................................... 351
     3.5.2 General Guidelines for Floating-Point Pipeline Alignment ......................... 351
     3.5.3 Parallel Instructions .................................................................................... 352
     3.5.4 Invalid Delay Instructions ......................................................................... 352
  3.6 Instruction Set ................................................................................................... 356
     3.6.1 Instruction Descriptions .............................................................................. 356
     3.6.2 General Instructions ................................................................................... 358
     3.6.3 Complex Math Instructions ....................................................................... 389
     3.6.4 Cyclic Redundancy Check (CRC) Instructions .......................................... 427
     3.6.5 Viterbi Instructions .................................................................................... 439
  3.7 Rounding Mode ............................................................................................... 461
4 Cyclic Redundancy Check (VCRC) ....................................................................... 463
  4.1 Overview ........................................................................................................... 464
  4.2 VCRC Code Development ................................................................................ 464
  4.3 Components of the C28x Plus VCRC ............................................................... 464
     4.3.1 Emulation Logic ......................................................................................... 465
     4.3.2 Memory Map ............................................................................................... 466
     4.3.3 CPU Interrupt Vectors ............................................................................... 466
     4.3.4 Memory Interface ....................................................................................... 466
     4.3.5 Address and Data Buses ........................................................................... 466
     4.3.6 Alignment of 32-Bit Accesses to Even Addresses ...................................... 467
  4.4 Register Set ....................................................................................................... 467
     4.4.1 VCRC Register Set ..................................................................................... 468
  4.5 Pipeline ............................................................................................................ 469
     4.5.1 Pipeline Overview ....................................................................................... 469
     4.5.2 General Guidelines for VCRC Pipeline Alignment ..................................... 469
  4.6 Instruction Set ................................................................................................... 470
     4.6.1 Instruction Descriptions .............................................................................. 470
     4.6.2 General Instructions ................................................................................... 472
5 C28 Viterbi, Complex Math and CRC Unit-II (VCU-II) ........................................... 507
  5.1 Overview ........................................................................................................... 508
  5.2 Components of the C28x Plus VCU .................................................................. 509
     5.2.1 Emulation Logic .......................................................................................... 511
     5.2.2 Memory Map ............................................................................................... 511
     5.2.3 CPU Interrupt Vectors ............................................................................... 511
     5.2.4 Memory Interface ....................................................................................... 511
     5.2.5 Address and Data Buses ........................................................................... 511
     5.2.6 Alignment of 32-Bit Accesses to Even Addresses ...................................... 512
  5.3 Register Set ....................................................................................................... 513
7 Trigonometric Math Unit (TMU)

5.3.1 VCU Register Set
5.3.2 VCU Status Register (VSTATUS)
5.3.3 Repeat Block Register (RB)
5.4 Pipeline
5.4.1 Pipeline Overview
5.4.2 General Guidelines for VCU Pipeline Alignment
5.4.3 Effect of Delay Slot Operations on the Flags
5.4.4 Invalid Delay Instructions
5.4.5 Moves From FPU Registers to C28x Registers
5.4.6 Parallel Instructions
5.4.7 FFT Instructions
5.4.8 Deinterleaver Instructions
5.5 Instruction Set
5.5.1 Instruction Descriptions
5.5.2 General Instructions
5.5.3 Arithmetic Math Instructions
5.5.4 Complex Math Instructions
5.5.5 Cyclic Redundancy Check (CRC) Instructions
5.5.6 Deinterleaver Instructions
5.5.7 FFT Instructions
5.5.8 Galois Instructions
5.5.9 Viterbi Instructions
5.6 Rounding Mode

6 Fast Integer Division Unit (FINTDIV)

6.1 Overview
6.1.1 Compatibility With the C28x Fixed-Point CPU and C28x Floating Point CPU
6.1.2 Fast Integer Division Code development
6.2 Components of the C28x plus FINTDIV (C28x+FINTDIV)
6.3 CPU Register Set
6.4 Pipeline
6.5 Types of Divisions supported by C28x+FINTDIV
6.6 C28x+Fast Integer Division – Fast Integer Division Instruction Set
6.6.1 Instruction Descriptions
6.6.2 Instructions

7 Trigonometric Math Unit (TMU)
Revision History ................................................................. 799
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1.</td>
<td>FPU Functional Block Diagram</td>
<td>12</td>
</tr>
<tr>
<td>1-2.</td>
<td>C28x With Floating-Point Registers</td>
<td>16</td>
</tr>
<tr>
<td>1-3.</td>
<td>Floating-point Unit Status Register (STF)</td>
<td>18</td>
</tr>
<tr>
<td>1-4.</td>
<td>Repeat Block Register (RB)</td>
<td>20</td>
</tr>
<tr>
<td>1-5.</td>
<td>FPU Pipeline</td>
<td>21</td>
</tr>
<tr>
<td>2-1.</td>
<td>FPU64 Functional Block Diagram</td>
<td>145</td>
</tr>
<tr>
<td>2-2.</td>
<td>C28x With FPU64 Floating-Point Registers</td>
<td>148</td>
</tr>
<tr>
<td>2-3.</td>
<td>Floating-point Unit Status Register (STF)</td>
<td>151</td>
</tr>
<tr>
<td>2-4.</td>
<td>Repeat Block Register (RB)</td>
<td>153</td>
</tr>
<tr>
<td>2-5.</td>
<td>FPU64 Pipeline</td>
<td>154</td>
</tr>
<tr>
<td>3-1.</td>
<td>C28x + VCU Block Diagram</td>
<td>340</td>
</tr>
<tr>
<td>3-2.</td>
<td>C28x + FPU + VCU Registers</td>
<td>344</td>
</tr>
<tr>
<td>3-3.</td>
<td>VCU Status Register (VSTATUS)</td>
<td>346</td>
</tr>
<tr>
<td>3-4.</td>
<td>Repeat Block Register (RB)</td>
<td>349</td>
</tr>
<tr>
<td>3-5.</td>
<td>C28x + FCU + VCU Pipeline</td>
<td>351</td>
</tr>
<tr>
<td>4-1.</td>
<td>C28x + VCRC Block Diagram</td>
<td>464</td>
</tr>
<tr>
<td>4-2.</td>
<td>C28x + VCRC Registers</td>
<td>467</td>
</tr>
<tr>
<td>5-1.</td>
<td>C28x + VCU Block Diagram</td>
<td>509</td>
</tr>
<tr>
<td>5-2.</td>
<td>C28x + FPU + VCU Registers</td>
<td>513</td>
</tr>
<tr>
<td>5-3.</td>
<td>VCU Status Register (VSTATUS)</td>
<td>516</td>
</tr>
<tr>
<td>5-4.</td>
<td>Repeat Block Register (RB)</td>
<td>519</td>
</tr>
<tr>
<td>5-5.</td>
<td>C28x + FCU + VCU Pipeline</td>
<td>521</td>
</tr>
<tr>
<td>6-1.</td>
<td>Transfer Function for Different Types of Division</td>
<td>751</td>
</tr>
<tr>
<td>7-1.</td>
<td>Calculation of RaH (Quadrant) and RbH (Ratio)</td>
<td>793</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>28x Plus Floating-Point CPU Register Summary</td>
<td>17</td>
</tr>
<tr>
<td>1-2</td>
<td>Floating-point Unit Status (STF) Register Field Descriptions</td>
<td>18</td>
</tr>
<tr>
<td>1-3</td>
<td>Repeat Block (RB) Register Field Descriptions</td>
<td>20</td>
</tr>
<tr>
<td>1-4</td>
<td>Operand Nomenclature</td>
<td>30</td>
</tr>
<tr>
<td>1-5</td>
<td>Summary of Instructions</td>
<td>32</td>
</tr>
<tr>
<td>2-1</td>
<td>28x Plus Floating-Point FPU64 CPU Register Summary</td>
<td>149</td>
</tr>
<tr>
<td>2-2</td>
<td>Floating-point Unit Status (STF) Register Field Descriptions</td>
<td>151</td>
</tr>
<tr>
<td>2-3</td>
<td>Repeat Block (RB) Register Field Descriptions</td>
<td>153</td>
</tr>
<tr>
<td>2-4</td>
<td>Operand Nomenclature</td>
<td>163</td>
</tr>
<tr>
<td>2-5</td>
<td>Summary of Instructions</td>
<td>165</td>
</tr>
<tr>
<td>3-1</td>
<td>Viterbi Decode Performance</td>
<td>339</td>
</tr>
<tr>
<td>3-2</td>
<td>Complex Math Performance</td>
<td>339</td>
</tr>
<tr>
<td>3-3</td>
<td>VCU Register Set</td>
<td>345</td>
</tr>
<tr>
<td>3-4</td>
<td>28x CPU Register Summary</td>
<td>346</td>
</tr>
<tr>
<td>3-5</td>
<td>VCU Status (VSTATUS) Register Field Descriptions</td>
<td>347</td>
</tr>
<tr>
<td>3-6</td>
<td>Operation Interaction with VSTATUS Bits</td>
<td>347</td>
</tr>
<tr>
<td>3-7</td>
<td>Repeat Block (RB) Register Field Descriptions</td>
<td>349</td>
</tr>
<tr>
<td>3-8</td>
<td>Operand Nomenclature</td>
<td>356</td>
</tr>
<tr>
<td>3-9</td>
<td>INSTRUCTION dest, source1, source2 Short Description</td>
<td>357</td>
</tr>
<tr>
<td>3-10</td>
<td>General Instructions</td>
<td>358</td>
</tr>
<tr>
<td>3-11</td>
<td>Complex Math Instructions</td>
<td>389</td>
</tr>
<tr>
<td>3-12</td>
<td>CRC Instructions</td>
<td>427</td>
</tr>
<tr>
<td>3-13</td>
<td>Viterbi Instructions</td>
<td>439</td>
</tr>
<tr>
<td>3-14</td>
<td>Example: Values Before Shift Right</td>
<td>461</td>
</tr>
<tr>
<td>3-15</td>
<td>Example: Values after Shift Right</td>
<td>461</td>
</tr>
<tr>
<td>3-16</td>
<td>Example: Addition with Right Shift and Rounding</td>
<td>461</td>
</tr>
<tr>
<td>3-17</td>
<td>Example: Addition with Rounding After Shift Right</td>
<td>461</td>
</tr>
<tr>
<td>3-18</td>
<td>Shift Right Operation With and Without Rounding</td>
<td>461</td>
</tr>
<tr>
<td>4-1</td>
<td>VCRC Status (VSTATUS) Register Field Descriptions</td>
<td>468</td>
</tr>
<tr>
<td>4-2</td>
<td>VCRC: The CRC result register for unsecured memories</td>
<td>468</td>
</tr>
<tr>
<td>4-3</td>
<td>VCRCPOLY: The CRC Polynomial register for generic CRC instructions</td>
<td>468</td>
</tr>
<tr>
<td>4-4</td>
<td>VCRCSIZE: The CRC Polynomial and Data Size register for generic CRC instructions</td>
<td>468</td>
</tr>
<tr>
<td>4-5</td>
<td>VCUREV: VCU revision register</td>
<td>468</td>
</tr>
<tr>
<td>4-6</td>
<td>Operand Nomenclature</td>
<td>471</td>
</tr>
<tr>
<td>4-7</td>
<td>INSTRUCTION dest, source1, source2 Short Description</td>
<td>471</td>
</tr>
<tr>
<td>4-8</td>
<td>General Instructions</td>
<td>472</td>
</tr>
<tr>
<td>5-1</td>
<td>Viterbi Decode Performance</td>
<td>508</td>
</tr>
<tr>
<td>5-2</td>
<td>Complex Math Performance</td>
<td>508</td>
</tr>
<tr>
<td>5-3</td>
<td>VCU Register Set</td>
<td>514</td>
</tr>
<tr>
<td>5-4</td>
<td>28x CPU Register Summary</td>
<td>515</td>
</tr>
<tr>
<td>5-5</td>
<td>VCU Status (VSTATUS) Register Field Descriptions</td>
<td>516</td>
</tr>
<tr>
<td>5-6</td>
<td>Operation Interaction With VSTATUS Bits</td>
<td>517</td>
</tr>
<tr>
<td>5-7</td>
<td>Repeat Block (RB) Register Field Descriptions</td>
<td>519</td>
</tr>
<tr>
<td>5-8</td>
<td>Operations Requiring a Delay Slot(s)</td>
<td>522</td>
</tr>
<tr>
<td>5-9</td>
<td>Operand Nomenclature</td>
<td>526</td>
</tr>
<tr>
<td>5-10</td>
<td>INSTRUCTION dest, source1, source2 Short Description</td>
<td>527</td>
</tr>
<tr>
<td>5-11</td>
<td>General Instructions</td>
<td>528</td>
</tr>
<tr>
<td>Table No.</td>
<td>Table Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5-12</td>
<td>Arithmetic Math Instructions</td>
<td>572</td>
</tr>
<tr>
<td>5-13</td>
<td>Complex Math Instructions</td>
<td>579</td>
</tr>
<tr>
<td>5-14</td>
<td>CRC Instructions</td>
<td>638</td>
</tr>
<tr>
<td>5-15</td>
<td>Deinterleaver Instructions</td>
<td>654</td>
</tr>
<tr>
<td>5-16</td>
<td>FFT Instructions</td>
<td>670</td>
</tr>
<tr>
<td>5-17</td>
<td>Galois Field Instructions</td>
<td>698</td>
</tr>
<tr>
<td>5-18</td>
<td>Viterbi Instructions</td>
<td>711</td>
</tr>
<tr>
<td>5-19</td>
<td>Example: Values Before Shift Right</td>
<td>746</td>
</tr>
<tr>
<td>5-20</td>
<td>Example: Values after Shift Right</td>
<td>746</td>
</tr>
<tr>
<td>5-21</td>
<td>Example: Addition with Right Shift and Rounding</td>
<td>746</td>
</tr>
<tr>
<td>5-22</td>
<td>Example: Addition with Rounding After Shift Right</td>
<td>746</td>
</tr>
<tr>
<td>5-23</td>
<td>Shift Right Operation With and Without Rounding</td>
<td>747</td>
</tr>
<tr>
<td>6-1</td>
<td>Operand Nomenclature</td>
<td>752</td>
</tr>
<tr>
<td>6-2</td>
<td>Summary of Instructions</td>
<td>754</td>
</tr>
<tr>
<td>7-1</td>
<td>TMU Type 0 Instructions</td>
<td>773</td>
</tr>
<tr>
<td>7-2</td>
<td>TMU Type 1 Additional Instructions</td>
<td>773</td>
</tr>
<tr>
<td>7-3</td>
<td>IEEE 32-Bit Single Precision Floating-Point Format</td>
<td>774</td>
</tr>
<tr>
<td>7-4</td>
<td>Delay Slot Requirements for TMU Instructions</td>
<td>777</td>
</tr>
<tr>
<td>7-5</td>
<td>Operand Nomenclature</td>
<td>780</td>
</tr>
<tr>
<td>7-6</td>
<td>Summary of Instructions</td>
<td>782</td>
</tr>
<tr>
<td>7-7</td>
<td>Summary of Instructions</td>
<td>796</td>
</tr>
</tbody>
</table>
This document describes the architecture, pipeline, and instruction sets of the TMU, VCRC, VCU-II, FPU32, and FPU64 accelerators.

About This Manual

The TMS320C2000™ digital signal processor (DSP) platform is part of the TMS320™ DSP family.

Notational Conventions

This document uses the following conventions.

- Hexadecimal numbers are shown with the suffix h or with a leading 0x. For example, the following number is 40 hexadecimal (decimal 64): 40h or 0x40.
- Registers in this document are shown as figures and described in tables.
  - Each register figure shows a rectangle divided into fields that represent the fields of the register. Each field is labeled with its bit name, its beginning and ending bit numbers above, and its read/write properties below. A legend explains the notation used for the properties
  - Reserved bits in a register figure designate a bit that is used for future device expansion.

Related Documentation

The following books describe the TMS320x28x and related support tools that are available on the TI website:

Data Manual and Errata—

SPRS439— TMS320F2833x, TMS320F2823x Digital Signal Controllers (DSCs) Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ272— TMS320F2833x, TMS320F2823x DSC Silicon Errata describes known advisories on silicon and provides workarounds.

SPRS516— TMS320C2834x Delfino Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ267— TMS320C2834x Delfino™ MCUs Silicon Errata describes known advisories on silicon and provides workarounds.

SPRS698— TMS320F2806x Piccolo™ Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ342— TMS320F2806x Piccolo™ MCUs Silicon Errata describes known advisories on silicon and provides workarounds.

SPRS742— F28M35x Concerto™ Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ357— F28M35x Concerto™ MCUs Silicon Errata describes known advisories on silicon and provides workarounds.

SPRS825— F28M36x Concerto™ Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ375— F28M36x Concerto™ MCUs Silicon Errata describes known advisories on silicon and provides workarounds.
SPRS880— TMS320F2837xD Dual-Core Delfino™ Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ412— TMS320F2837xD Dual-Core Delfino™ MCUs Silicon Errata describes known advisories on silicon and provides workarounds.

SPRS881— TMS320F2837xS Delfino™ Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ422— TMS320F2837xS Delfino™ MCUs Silicon Errata describes known advisories on silicon and provides workarounds.

SPRS902— TMS320F2807x Piccolo™ Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ423— TMS320F2807x Piccolo™ MCUs Silicon Errata describes known advisories on silicon and provides workarounds.

SPRS945— TMS320F28004x Piccolo™ Microcontrollers Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ439— TMS320F28004x Piccolo™ Microcontrollers Silicon Errata describes known advisories on silicon and provides workarounds.

SPRSP14— TMS320F2838x Microcontrollers With Connectivity Manager Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications.

SPRZ458— TMS320F2838x MCUs Silicon Errata describes known advisories on silicon and provides workarounds.

Trademarks

Delfino, Piccolo, Concerto, TMS320C2000 are trademarks of Texas Instruments.
The TMS320C2000™ DSP family consists of fixed-point and floating-point digital signal controllers (DSCs). TMS320C2000™ Digital Signal Controllers combine control peripheral integration and ease of use of a microcontroller (MCU) with the processing power and C efficiency of TI’s leading DSP technology. This chapter provides an overview of the architectural structure and components of the C28x plus floating-point unit CPU.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Overview</td>
<td>12</td>
</tr>
<tr>
<td>1.2 Components of the C28x plus Floating-Point CPU</td>
<td>13</td>
</tr>
<tr>
<td>1.3 CPU Register Set</td>
<td>15</td>
</tr>
<tr>
<td>1.4 Pipeline</td>
<td>21</td>
</tr>
<tr>
<td>1.5 Floating Point Unit Instruction Set</td>
<td>29</td>
</tr>
</tbody>
</table>
1.1 Overview

The C28x plus floating-point (C28x+FPU) processor extends the capabilities of the C28x fixed-point CPU by adding registers and instructions to support IEEE single-precision floating point operations. This device draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The DSC features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and modified Harvard architecture (usable in Von Neumann mode). The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation. The modified Harvard architecture of the CPU enables instruction and data fetches to be performed in parallel. The CPU can read instructions and data while it writes data simultaneously to maintain the single-cycle instruction operation across the pipeline. The CPU does this over six separate address/data buses.

Throughout this document the following notations are used:

- C28x refers to the C28x fixed-point CPU.
- C28x plus Floating-Point and C28x+FPU both refer to the C28x CPU with enhancements to support IEEE single-precision floating-point operations.

1.1.1 Compatibility with the C28x Fixed-Point CPU

No changes have been made to the C28x base set of instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x CPU are completely compatible with the C28x+FPU and all of the features of the C28x documented in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430) apply to the C28x+FPU.

Figure 1-1 shows basic functions of the FPU.
1.1.1.1 Floating-Point Code Development

When developing C28x floating-point code use Code Composer Studio 3.3, or later, with at least service release 8. The C28x compiler V5.0, or later, is also required to generate C28x native floating-point opcodes. This compiler is available via Code Composer Studio update advisor as a separate download. V5.0 can generate both fixed-point as well as floating-point code. To build floating-point code use the compiler switches: -v28 and -float_support = fpu32. In Code Composer Studio 3.3 the float_support option is in the build options under compiler-> advanced: floating point support. Without the float_support flag, or with float_support = none, the compiler will generate fixed-point code.

When building for C28x floating-point make sure all associated libraries have also been built for floating-point. The standard run-time support (RTS) libraries built for floating-point included with the compiler have fpu32 in their name. For example rts2800_fpu32.lib and rts2800_fpu_eh.lib have been built for the floating-point unit. The "eh" version has exception handling for C++ code. Using the fixed-point RTS libraries in a floating-point project will result in the linker issuing an error for incompatible object files.

To improve performance of native floating-point projects, consider using the C28x FPU Fast RTS Library (SPRC664). This library contains hand-coded optimized math routines such as division, square root, atan2, sin and cos. This library can be linked into your project before the standard runtime support library to give your application a performance boost. As an example, the standard RTS library uses a polynomial expansion to calculate the sin function. The Fast RTS library, however, uses a math look-up table in the boot ROM of the device. Using this look-up table method results in approximately a 20 cycle savings over the standard RTS calculation.

1.2 Components of the C28x plus Floating-Point CPU

The C28x+FPU contains:

- A central processing unit for generating data and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory
- A floating-point unit for IEEE single-precision floating point operations.
- Emulation logic for monitoring and controlling various parts and functions of the device and for testing device operation. This logic is identical to that on the C28x fixed-point CPU.
- Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts. This logic is identical to the C28x fixed-point CPU.

Some features of the C28x+FPU central processing unit are:

- Fixed-Point instructions are pipeline protected. This pipeline for fixed-point instructions is identical to that on the C28x fixed-point CPU. The CPU implements an 8-phase pipeline that prevents a write to and a read from the same location from occurring out of order. See Figure 1-5.
- Some floating-point instructions require pipeline alignment. This alignment is done through software to allow the user to improve performance by taking advantage of required delay slots.
- Independent register space. These registers function as system-control registers, math registers, and data pointers. The system-control registers are accessed by special instructions.
- Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- Floating point unit (FPU). The 32-bit FPU performs IEEE single-precision floating-point operations.
- Address register arithmetic unit (ARAU). The ARAU generates data memory addresses and increments or decrements pointers in parallel with ALU operations.
- Barrel shifter. This shifter performs all left and right shifts of fixed-point data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.
- Fixed-Point Multiplier. The multiplier performs 32-bit × 32-bit 2s-complement multiplication with a 64-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.
1.2.1 *Emulation Logic*

The emulation logic is identical to that on the C28x fixed-point CPU. This logic includes the following features:

- Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline.
- A counter for performance benchmarking.
- Multiple debug events. Any of the following debug events can cause a break in program execution:
  - A breakpoint initiated by the ESTOP0 or ESTOP1 instruction.
  - An access to a specified program-space or data-space location.
When a debug event causes the C28x to enter the debug-halt state, the event is called a break event.
- Real-time mode of operation.

For more details about these features, refer to the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430).

1.2.2 *Memory Map*

Like the C28x, the C28x+FPU uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x+FPU designs are uniformly mapped to both program and data space. For specific details about each of the map segments, see the data sheet for your device.

1.2.3 *On-Chip Program and Data*

All C28x+FPU based devices contain at least two blocks of single access on-chip memory referred to as M0 and M1. Each of these blocks is 1K words in size. M0 is mapped at addresses 0x0000 – 0x03FF and M1 is mapped at addresses 0x0400 – 0x07FF. Like all other memory blocks on the C28x+FPU devices, M0 and M1 are mapped to both program and data space. Therefore, you can use M0 and M1 to execute code or for data variables. At reset, the stack pointer is set to the top of block M1. Depending on the device, it may also have additional random-access memory (RAM), read-only memory (ROM), external interface zones, or flash memory.

1.2.4 *CPU Interrupt Vectors*

The C28x+FPU interrupt vectors are identical to those on the C28x CPU. Sixty-four addresses in program space are set aside for a table of 32 CPU interrupt vectors. The CPU vectors can be mapped to the top or bottom of program space by way of the VMAP bit. For more information about the CPU vectors, see *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430). For devices with a peripheral interrupt expansion (PIE) block, the interrupt vectors will reside in the PIE vector table and this memory can be used as program memory.

1.2.5 *Memory Interface*

The C28x+FPU memory interface is identical to that on the C28x. The C28x+FPU memory map is accessible outside the CPU by the memory interface, which connects the CPU logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed. The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the C28x+FPU supports special byte-access instructions that can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.
1.2.5.1 Address and Data Buses

Like the C28x, the memory interface has three address buses:

- **PAB: Program address bus**
  The PAB carries addresses for reads and writes from program space. PAB is a 22-bit bus.

- **DRAB: Data-read address bus**
  The 32-bit DRAB carries addresses for reads from data space.

- **DWAB: Data-write address bus**
  The 32-bit DWAB carries addresses for writes to data space.

The memory interface also has three data buses:

- **PRDB: Program-read data bus**
  The PRDB carries instructions during reads from program space. PRDB is a 32-bit bus.

- **DRDB: Data-read data bus**
  The DRDB carries data during reads from data space. DRDB is a 32-bit bus.

- **DWDB: Data-/Program-write data bus**
  The 32-bit DWDB carries data during writes to data space or program space.

A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time. This behavior is identical to the C28x CPU.

1.2.5.2 Alignment of 32-Bit Accesses to Even Addresses

The C28x+FPU CPU expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU will begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the CPU.

You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space.

1.3 CPU Register Set

The C28x+FPU architecture is the same as the C28x CPU with an extended register and instruction set to support IEEE single-precision floating point operations. This section describes the extensions to the C28x architecture.

1.3.1 CPU Registers

Devices with the C28x+FPU include the standard C28x register set plus an additional set of floating-point unit registers. The additional floating-point unit registers are the following:

- Eight floating-point result registers, RnH (where n = 0 - 7)
- Floating-point Status Register (STF)
- Repeat Block Register (RB)

All of the floating-point registers except the repeat block register are shadowed. This shadowing can be used in high priority interrupts for fast context save and restore of the floating-point registers.
Figure 1-2 shows a diagram of both register sets and Table 1-1 shows a register summary. For information on the standard C28x register set, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430).

**Figure 1-2. C28x With Floating-Point Registers**

<table>
<thead>
<tr>
<th>Standard C28x Register Set</th>
<th>Additional 32-bit FPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC (32-bit)</td>
<td>R0H (32-bit)</td>
</tr>
<tr>
<td>P (32-bit)</td>
<td>R1H (32-bit)</td>
</tr>
<tr>
<td>XT (32-bit)</td>
<td>R2H (32-bit)</td>
</tr>
<tr>
<td>XAR0 (32-bit)</td>
<td>R3H (32-bit)</td>
</tr>
<tr>
<td>XAR1 (32-bit)</td>
<td>R4H (32-bit)</td>
</tr>
<tr>
<td>XAR2 (32-bit)</td>
<td>R5H (32-bit)</td>
</tr>
<tr>
<td>XAR3 (32-bit)</td>
<td>R6H (32-bit)</td>
</tr>
<tr>
<td>XAR4 (32-bit)</td>
<td>R7H (32-bit)</td>
</tr>
<tr>
<td>XAR5 (32-bit)</td>
<td>FPU Status Register (STF)</td>
</tr>
<tr>
<td>XAR6 (32-bit)</td>
<td>Repeat Block Register (RB)</td>
</tr>
<tr>
<td>XAR7 (32-bit)</td>
<td>FPU registers R0H - R7H and STF are shadowed for fast context save and restore</td>
</tr>
<tr>
<td>PC (22-bit)</td>
<td></td>
</tr>
<tr>
<td>RPC (22-bit)</td>
<td></td>
</tr>
<tr>
<td>DP (16-bit)</td>
<td></td>
</tr>
<tr>
<td>SP (16-bit)</td>
<td></td>
</tr>
<tr>
<td>ST0 (16-bit)</td>
<td></td>
</tr>
<tr>
<td>ST1 (16-bit)</td>
<td></td>
</tr>
<tr>
<td>IER (16-bit)</td>
<td></td>
</tr>
<tr>
<td>IFR (16-bit)</td>
<td></td>
</tr>
<tr>
<td>DBGIER (16-bit)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 1-1. 28x Plus Floating-Point CPU Register Summary

<table>
<thead>
<tr>
<th>Register</th>
<th>C28x CPU</th>
<th>C28x+FPU</th>
<th>Size</th>
<th>Description</th>
<th>Value After Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Accumulator</td>
<td>0x00000000</td>
</tr>
<tr>
<td>AH</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>High half of ACC</td>
<td>0x0000</td>
</tr>
<tr>
<td>AL</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of ACC</td>
<td>0x0000</td>
</tr>
<tr>
<td>XAR0</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 0</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR1</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR2</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 2</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR3</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 3</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR4</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 4</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR5</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 5</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR6</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 6</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR7</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 7</td>
<td>0x00000000</td>
</tr>
<tr>
<td>AR0</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR0</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR1</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR1</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR2</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR2</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR3</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR3</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR4</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR4</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR5</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR5</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR6</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR6</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR7</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR7</td>
<td>0x0000</td>
</tr>
<tr>
<td>DP</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Data-page pointer</td>
<td>0x0000</td>
</tr>
<tr>
<td>IFR</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Interrupt flag register</td>
<td>0x0000</td>
</tr>
<tr>
<td>IER</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Interrupt enable register</td>
<td>0x0000</td>
</tr>
<tr>
<td>DBGIER</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Debug interrupt enable register</td>
<td>0x0000</td>
</tr>
<tr>
<td>P</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Product register</td>
<td>0x00000000</td>
</tr>
<tr>
<td>PH</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>High half of P</td>
<td>0x0000</td>
</tr>
<tr>
<td>PL</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of P</td>
<td>0x0000</td>
</tr>
<tr>
<td>PC</td>
<td>Yes</td>
<td>Yes</td>
<td>22 bits</td>
<td>Program counter</td>
<td>0x3FFFFFF0</td>
</tr>
<tr>
<td>RPC</td>
<td>Yes</td>
<td>Yes</td>
<td>22 bits</td>
<td>Return program counter</td>
<td>0x00000000</td>
</tr>
<tr>
<td>SP</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Stack pointer</td>
<td>0x00400</td>
</tr>
<tr>
<td>ST0</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Status register 0</td>
<td>0x0000</td>
</tr>
<tr>
<td>ST1</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Status register 1</td>
<td>0x080B(1)</td>
</tr>
<tr>
<td>XT</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Multiplicand register</td>
<td>0x000000000</td>
</tr>
<tr>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>High half of XT</td>
<td>0x0000</td>
</tr>
<tr>
<td>TL</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XT</td>
<td>0x0000</td>
</tr>
<tr>
<td>ROH</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 0</td>
<td>0.0</td>
</tr>
<tr>
<td>R1H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 1</td>
<td>0.0</td>
</tr>
<tr>
<td>R2H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 2</td>
<td>0.0</td>
</tr>
<tr>
<td>R3H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 3</td>
<td>0.0</td>
</tr>
<tr>
<td>R4H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 4</td>
<td>0.0</td>
</tr>
<tr>
<td>R5H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 5</td>
<td>0.0</td>
</tr>
<tr>
<td>R6H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 6</td>
<td>0.0</td>
</tr>
<tr>
<td>R7H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point result register 7</td>
<td>0.0</td>
</tr>
<tr>
<td>STF</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point status register</td>
<td>0x00000000</td>
</tr>
<tr>
<td>RB</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Repeat block register</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

(1) Reset value shown is for devices without the VMAP signal and MOM1MAP signal pinned out. On these devices both of these signals are tied high internal to the device.
### 1.3.1.1 Floating-Point Status Register (STF)

The floating-point status register (STF) reflects the results of floating-point operations. There are three basic rules for floating point operation flags:

1. Zero and negative flags are set based on moves to registers.
2. Zero and negative flags are set based on the result of compare, minimum, maximum, negative and absolute value operations.
3. Overflow and underflow flags are set by math instructions such as multiply, add, subtract and 1/x.

These flags may also be connected to the peripheral interrupt expansion (PIE) block on your device. This can be useful for debugging underflow and overflow conditions within an application.

As on the C28x, program flow is controlled by C28x instructions that read status flags in the status register 0 (ST0). If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into ST0 flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVST0 FLAG instruction is used to load the current value of specified STF flags into the respective bits of ST0. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified.

---

**Example 1-1. Moving STF Flags to the ST0 Register**

```plaintext
Loop:
MOV32 R0H,*XAR4++
MOV32 R1H,*XAR3++
CMPF32 R1H, R0H
MOVST0 ZF, NF    ; Move ZF and NF to ST0
BF Loop, GT      ; Loop if (R1H > R0H)
```

---

**Figure 1-3. Floating-point Unit Status Register (STF)**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>SHDWS</td>
<td>0</td>
<td>Shadow Mode Status Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>This bit is set to 1 by the SAVE instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bit is not affected by loading the status register either from memory or from the shadow values.</td>
</tr>
<tr>
<td>30-10</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>9</td>
<td>RND32</td>
<td>0</td>
<td>Round 32-bit Floating-Point Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>If this bit is one, the MPYF32, ADDF32 and SUBF32 instructions will round to the nearest even value.</td>
</tr>
<tr>
<td>8-7</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>6</td>
<td>TF</td>
<td>0</td>
<td>Test Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The condition tested with the TESTTF instruction is true.</td>
</tr>
</tbody>
</table>

---

**Table 1-2. Floating-point Unit Status (STF) Register Field Descriptions**
Table 1-2. Floating-point Unit Status (STF) Register Field Descriptions (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>ZI</td>
<td>0</td>
<td>The following instructions modify this flag based on the integer value stored in the destination register: MOV32, MOV32D, MOVDD32. The SETFLG and SAVE instructions can also be used to modify this flag. The integer value is not negative.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The integer value is negative.</td>
</tr>
<tr>
<td>4</td>
<td>NI</td>
<td>0</td>
<td>The following instructions modify this flag based on the integer value stored in the destination register: MOV32, MOV32D, MOVDD32. The SETFLG and SAVE instructions can also be used to modify this flag. The integer value is not negative.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The integer value is negative.</td>
</tr>
<tr>
<td>3</td>
<td>ZF</td>
<td>0</td>
<td>The following instructions modify this flag based on the floating-point value stored in the destination register: MOV32, MOV32D, MOVDD32. The SETFLG and SAVE instructions can also be used to modify this flag. The floating-point value is not zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The floating-point value is zero.</td>
</tr>
<tr>
<td>2</td>
<td>NF</td>
<td>0</td>
<td>The following instructions modify this flag based on the floating-point value stored in the destination register: MOV32, MOV32D, MOVDD32. The SETFLG and SAVE instructions can also be used to modify this flag. The floating-point value is not negative.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The floating-point value is negative.</td>
</tr>
<tr>
<td>1</td>
<td>LUF</td>
<td>0</td>
<td>The following instructions will set this flag to 1 if an underflow occurs: MPYF32, ADDF32, SUBF32, MACF32, EINVF32, EISQRTF32. An underflow condition has not been latched. If the MOVST0 instruction is used to copy this bit to ST0, then LUF will be cleared.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>An underflow condition has been latched.</td>
</tr>
<tr>
<td>0</td>
<td>LVF</td>
<td>0</td>
<td>The following instructions will set this flag to 1 if an overflow occurs: MPYF32, ADDF32, SUBF32, MACF32, EINVF32, EISQRTF32. An overflow condition has not been latched. If the MOVST0 instruction is used to copy this bit to ST0, then LVF will be cleared.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>An overflow condition has been latched.</td>
</tr>
</tbody>
</table>

(1) A negative zero floating-point value is treated as a positive zero value when configuring the ZF and NF flags.

(2) A DeNorm floating-point value is treated as a positive zero value when configuring the ZF and NF flags.
1.3.1.2 Repeat Block Register (RB)

The repeat block instruction (RPTB) is a new instruction for C28x+FPU. This instruction allows you to repeat a block of code as shown in Example 1-2.

Example 1-2. The Repeat Block (RPTB) Instruction uses the RB Register

```assembly
; find the largest element and put its address in XAR6
MOV32 R0H, *XAR0++;
.align 2; Aligns the next instruction to an even address
NOP; Makes RPTB odd aligned - required for a block size of 8
RPTB VECTOR_MAX_END, AR7; RA is set to 1
MOVL ACC,XAR0
MOV32 R1H,*XAR0++; RSIZE reflects the size of the RPTB block
MAXF32 R0H,R1H; in this case the block size is 8
MOVST0 NF,ZF
MOVL XAR6,ACC,LT
VECTOR_MAX_END: ; RE indicates the end address. RA is cleared
```

The C28x_FPU hardware automatically populates the RB register based on the execution of a RPTB instruction. This register is not normally read by the application and does not accept debugger writes.

![Figure 1-4. Repeat Block Register (RB)](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RAS</td>
<td>0</td>
<td>Repeat Block Active Shadow Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Repeat Block was not active when the interrupt was taken.</td>
</tr>
<tr>
<td>30</td>
<td>RA</td>
<td>0</td>
<td>Repeat Block Active Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Repeat Block was active when the interrupt was taken.</td>
</tr>
<tr>
<td>29-23</td>
<td>RSIZE</td>
<td>0-7</td>
<td>Repeat Block Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8/9-0x7F</td>
<td>Illegal block size. A RPTB block that starts at an even address must include at least 9 16-bit words and a block that starts at an odd address must include at least 8 16-bit words. The maximum block size is 127 16-bit words. The codegen assembler will check for proper block size and alignment.</td>
</tr>
</tbody>
</table>
Table 1-3. Repeat Block (RB) Register Field Descriptions (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-16</td>
<td>RE</td>
<td>Repeat Block End Address&lt;br&gt;&lt;br&gt;This 7-bit value specifies the end address location of the repeat block. The RE value is calculated by hardware based on the RSIZE field and the PC value when the RPTB instruction is executed.&lt;br&gt;&lt;br&gt;RE = lower 7 bits of (PC + 1 + RSIZE)</td>
<td></td>
</tr>
<tr>
<td>15-0</td>
<td>RC</td>
<td>0</td>
<td>Repeat Count&lt;br&gt;&lt;br&gt;The block will not be repeated; it will be executed only once. In this case the repeat active, RA, bit will not be set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1-0xFFFF</td>
<td>This 16-bit value determines how many times the block will repeat. The counter is initialized when the RPTB instruction is executed and is decremented when the PC reaches the end of the block. When the counter reaches zero, the repeat active bit is cleared and the block will be executed one more time. Therefore the total number of times the block is executed is RC+1.</td>
</tr>
</tbody>
</table>

1.4 Pipeline

The pipeline flow for C28x instructions is identical to that of the C28x CPU described in TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430). Some floating-point instructions, however, use additional execution phases and thus require a delay to allow the operation to complete. This pipeline alignment is achieved by inserting NOPs or non-conflicting instructions when required. Software control of delay slots allows you to improve performance of an application by taking advantage of the delay slots and filling them with non-conflicting instructions. This section describes the key characteristics of the pipeline with regards to floating-point instructions. The rules for avoiding pipeline conflicts are small in number and simple to follow and the C28x+FPU assembler will help you by issuing errors for conflicts.

1.4.1 Pipeline Overview

The C28x FPU pipeline is identical to the C28x pipeline for all standard C28x instructions. In the decode2 stage (D2), it is determined if an instruction is a C28x instruction or a floating-point unit instruction. The pipeline flow is shown in Figure 1-5. Notice that stalls due to normal C28x pipeline stalls (D2) and memory waitstates (R2 and W) will also stall any C28x FPU instruction. Most C28x FPU instructions are single cycle and will complete in the FPU E1 or W stage which aligns to the C28x pipeline. Some instructions will take an additional execute cycle (E2). For these instructions you must wait a cycle for the result from the instruction to be available. The rest of this section will describe when delay cycles are required. Keep in mind that the assembly tools for the C28x+FPU will issue an error if a delay slot has not been handled correctly.

Figure 1-5. FPU Pipeline
1.4.2 General Guidelines for Floating-Point Pipeline Alignment

While the C28x+FPU assembler will issue errors for pipeline conflicts, you may still find it useful to understand when software delays are required. This section describes three guidelines you can follow when writing C28x+FPU assembly code.

Floating-point instructions that require delay slots have a 'p' after their cycle count. For example ‘2p’ stands for 2 pipelined cycles. This means that an instruction can be started every cycle, but the result of the instruction will only be valid one instruction later.

There are three general guidelines to determine if an instruction needs a delay slot:
1. Floating-point math operations (multiply, addition, subtraction, 1/x and MAC) require 1 delay slot.
2. Conversion instructions between integer and floating-point formats require 1 delay slot.
3. Everything else does not require a delay slot. This includes minimum, maximum, compare, load, store, negative and absolute value instructions.

There are two exceptions to these rules. First, moves between the CPU and FPU registers require special pipeline alignment that is described later in this section. These operations are typically infrequent. Second, the MACF32 R7H, R3H, mem32, *XAR7 instruction has special requirements that make it easier to use. Refer to the MACF32 instruction description for details.

An example of the 32-bit ADDF32 instruction is shown in Example 1-3. ADDF32 is a 2p instruction and therefore requires one delay slot. The destination register for the operation, R0H, will be updated one cycle after the instruction for a total of 2 cycles. Therefore, a NOP or instruction that does not use R0H must follow this instruction.

Any memory stall or pipeline stall will also stall the floating-point unit. This keeps the floating-point unit aligned with the C28x pipeline and there is no need to change the code based on the waitstates of a memory block.

Please note that on certain devices instructions make take additional cycles to complete under specific conditions. These exceptions will be documented in the device errata.

Example 1-3. 2p Instruction Pipeline Alignment

```
ADDF32 R0H, #1.5, R1H ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- ADDF32 completes, R0H updated
NOP ; Any instruction
```
1.4.3 Moves from FPU Registers to C28x Registers

When transferring from the floating-point unit registers to the C28x CPU registers, additional pipeline alignment is required as shown in Example 1-4 and Example 1-5.

Example 1-4. Floating-Point to C28x Register Software Pipeline Alignment

```
; MINF32: 32-bit floating-point minimum: single-cycle operation
; An alignment cycle is required before copying R0H to ACC
MINF32 R0H, R1H ; Single-cycle instruction
               ; -- R0H is valid
NOP ; Alignment cycle
MOV32 @ACC, R0H ; Copy R0H to ACC
```

For 1-cycle FPU instructions, one delay slot is required between a write to the floating-point register and the transfer instruction as shown in Example 1-4. For 2p FPU instructions, two delay slots are required between a write to the floating-point register and the transfer instruction as shown in Example 1-5.
Example 1-5. Floating-Point to C28x Register Software Pipeline Alignment

```
; ADDF32: 32-bit floating-point addition: 2p operation
; An alignment cycle is required before copying R0H to ACC
ADDF32 R0H, R1H, #2 ; R0H = R1H + 2, 2 pipeline cycle instruction
NOP ; 1 delay cycle or non-conflicting instruction
; <=-- R0H is valid
NOP ; Alignment cycle
NOP :
MOV32 @ACC, R0H ; Copy R0H to ACC
```

1.4.4 Moves from C28x Registers to FPU Registers

Transfers from the standard C28x CPU registers to the floating-point registers require four alignment cycles. For the 2833x, 2834x, 2806x, 28M35xx and 28M26xx, the four alignment cycles can be filled with NOPs or any non-conflicting instruction except for F32TOUI32 RaH, RbH, FRACF32 RaH, RbH, UI16TOF32 RaH, mem16 and UI16TOF32 RaH, RbH. These instructions cannot replace any of the four alignment NOPs. On newer devices any non-conflicting instruction can go into the four alignment cycles. Please refer to the device errata for specific exceptions to these rules.

Example 1-6. C28x Register to Floating-Point Register Software Pipeline Alignment

```
; Four alignment cycles are required after copying a standard 28x CPU
; register to a floating-point register.
;
MOV32 R0H, @ACC ; Copy ACC to R0H
NOP
NOP
NOP
; Wait 4 cycles
NOP
ADDF32 R2H, R1H, R0H ; R0H is valid
```
1.4.5 Parallel Instructions

Parallel instructions are single opcodes that perform two operations in parallel. This can be a math operation in parallel with a move operation, or two math operations in parallel. Math operations with a parallel move are referred to as 2p/1 instructions. The math portion of the operation takes two pipelined cycles while the move portion of the operation is single cycle. This means that NOPs or other non-conflicting instructions must be inserted to align the math portion of the operation. An example of an add with parallel move instruction is shown in Example 1-7.

Example 1-7. 2p/1 Parallel Instruction Software Pipeline Alignment

```
; ADDF32 || MOV32 instruction: 32-bit floating-point add with parallel move
; ADDF32 is a 2p operation
; MOV32 is a 1 cycle operation

ADDF32 R0H, R1H, #2 ; R0H = R1H + 2, 2 pipeline cycle operation
|| MOV32 R1H, @Val ; R1H gets the contents of Val, single cycle operation
; <-- MOV32 completes here (R1H is valid)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- ADDF32 completes here (R0H is valid)
NOP ; Any instruction
```

Parallel math instructions are referred to as 2p/2p instructions. Both math operations take 2 cycles to complete. This means that NOPs or other non-conflicting instructions must be inserted to align the both math operations. An example of a multiply with parallel add instruction is shown in Example 1-8.

Example 1-8. 2p/2p Parallel Instruction Software Pipeline Alignment

```
; MPYF32 || ADDF32 instruction: 32-bit floating-point multiply with parallel add
; MPYF32 is a 2p operation
; ADDF32 is a 2p cycle operation

MPYF32 R0H, R1H, R3H ; R0H = R1H * R3H, 2 pipeline cycle operation
|| ADDF32 R1H, R2H, R4H ; R1H = R2H + R4H, 2 pipeline cycle operation
NOP ; 1 cycle delay or non-conflicting instruction
; <-- MPYF32 and ADDF32 complete here (R0H and R1H are valid)
NOP ; Any instruction
```

1.4.6 Invalid Delay Instructions

Most instructions can be used in delay slots as long as source and destination register conflicts are avoided. The C28x+FPU assembler will issue an error anytime you use an conflicting instruction within a delay slot. The following guidelines can be used to avoid these conflicts.

**NOTE:** Destination register conflicts in delay slots:

Any operation used for pipeline alignment delay must not use the same destination register as the instruction requiring the delay. See Example 1-9.

In Example 1-9 the MPYF32 instruction uses R2H as its destination register. The next instruction should not use R2H as its destination. Since the MOV32 instruction uses the R2H register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than R2H for the MOV32 instruction as shown in Example 1-10.
Example 1-9. Destination Register Conflict

; Invalid delay instruction. Both instructions use the same destination register
MPYF32 R2H, R1H, R0H ; 2p instruction
MOV32 R2H, mem32 ; Invalid delay instruction

Example 1-10. Destination Register Conflict Resolved

; Valid delay instruction
MPYF32 R2H, R1H, R0H ; 2p instruction MOV32 R1H, mem32
MOV32 R3H, mem32 ; Valid delay
; <-- MPYF32 completes, R2H valid

NOTE: Instructions in delay slots cannot use the instruction’s destination register as a source register.
Any operation used for pipeline alignment delay must not use the destination register of the
instruction requiring the delay as a source register as shown in Example 1-11. For parallel
instructions, the current value of a register can be used in the parallel operation before it is
overwritten as shown in Example 1-13.

In Example 1-11 the MPYF32 instruction again uses R2H as its destination register. The next instruction
should not use R2H as its source since the MPYF32 will take an additional cycle to complete. Since the
ADDF32 instruction uses the R2H register a pipeline conflict will be issued by the assembler. This conflict
can be resolved by using a register other than R2H or by inserting a non-conflicting instruction between
the MPYF32 and ADDF32 instructions. Since the SUBF32 does not use R2H this instruction can be
moved before the ADDF32 as shown in Example 1-12.

Example 1-11. Destination/Source Register Conflict

; Invalid delay instruction. ADDF32 should not use R2H as a source operand
MPYF32 R2H, R1H, R0H ; 2p instruction
ADDF32 R3H, R3H, R2H ; Invalid delay instruction
SUBF32 R4H, R1H, R0H

Example 1-12. Destination/Source Register Conflict Resolved

; Valid delay instruction.
MPYF32 R2H, R1H, R0H ; 2p instruction
SUBF32 R4H, R1H, R0H ; Valid delay for MPYF32
ADDF32 R3H, R3H, R2H ; <-- MPYF32 completes, R2H valid
NOP ; <-- SUBF32 completes, R4H valid

It should be noted that a source register for the 2nd operation within a parallel instruction can be the same
as the destination register of the first operation. This is because the two operations are started at the
same time. The 2nd operation is not in the delay slot of the first operation. Consider Example 1-13 where
the MPYF32 uses R2H as its destination register. The MOV32 is the 2nd operation in the instruction and
can freely use R2H as a source register. The contents of R2H before the multiply will be used by MOV32.
Example 1-13. Parallel Instruction Destination/Source Exception

```
; Valid parallel operation.
MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32 mem32, R2H ; <-- Uses R2H before the MPYF32
|| NOP ; <-- mem32 updated
|| NOP ; <-- Delay for MPYF32
|| NOP ; <-- R2H updated
```

Likewise, the source register for the 2nd operation within a parallel instruction can be the same as one of the source registers of the first operation. The MPYF32 operation in Example 1-14 uses the R1H register as one of its sources. This register is also updated by the MOV32 register. The multiplication operation will use the value in R1H before the MOV32 updates it.

Example 1-14. Parallel Instruction Destination/Source Exception

```
; Valid parallel instruction
MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32 R1H, mem32 ; Valid
|| NOP ; <-- MOV32 completes, R1H valid
|| NOP ; <-- MPYF32, R2H valid
```

**NOTE:** Operations within parallel instructions cannot use the same destination register.

When two parallel operations have the same destination register, the result is invalid. For example, see Example 1-15.

If both operations within a parallel instruction try to update the same destination register as shown in Example 1-15 the assembler will issue an error.

Example 1-15. Invalid Destination Within a Parallel Instruction

```
; Invalid parallel instruction. Both operations use the same destination register
MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32 R2H, mem32 ; Invalid
```

Some instructions access or modify the STF flags. Because the instruction requiring a delay slot will also be accessing the STF flags, these instructions should not be used in delay slots. These instructions are SAVE, SETFLG, RESTORE and MOVST0.

**NOTE:** Do not use SAVE, SETFLG, RESTORE, or the MOVST0 instruction in a delay slot.
1.4.7 Optimizing the Pipeline

The following example shows how delay slots can be used to improve the performance of an algorithm. The example performs two \( Y = MX+B \) operations. In Example 1-16, no optimization has been done. The \( Y = MX+B \) calculations are sequential and each takes 7 cycles to complete. Notice there are NOPs in the delay slots that could be filled with non-conflicting instructions. The only requirement is these instructions must not cause a register conflict or access the STF register flags.

Example 1-16. Floating-Point Code Without Pipeline Optimization

```
; Using NOPs for alignment cycles, calculate the following:
;
; Y1 = M1*X1 + B1
; Y2 = M2*X2 + B2
;
; Calculate Y1

MOV32 R0H,@M1 ; Load R0H with M1 - single cycle
MOV32 R1H,@X1 ; Load R1H with X1 - single cycle
MPYF32 R1H,R1H,R0H ; R1H = M1 * X1 - 2p operation
| MOV32 R0H,@B1 ; Load R0H with B1 - single cycle
NOP ; Wait for MPYF32 to complete
| ADDF32 R1H,R1H,R0H ; R1H = R1H + R0H - 2p operation
NOP ; Wait for ADDF32 to complete
| MOV32 @Y1,R1H ; Save R1H in Y1 - single cycle

; Calculate Y2

MOV32 R0H,@M2 ; Load R0H with M2 - single cycle
MOV32 R1H,@X2 ; Load R1H with X2 - single cycle
MPYF32 R1H,R1H,R0H ; R1H = M2 * X2 - 2p operation
| MOV32 R0H,@B2 ; Load R0H with B2 - single cycle
NOP ; Wait for MPYF32 to complete
| ADDF32 R1H,R1H,R0H ; R1H = R1H + R0H
NOP ; Wait for ADDF32 to complete
| MOV32 @Y2,R1H ; Save R1H in Y2

; 14 cycles
; 48 bytes
```

The code shown in Example 1-17 was generated by the C28x+FPU compiler with optimization enabled. Notice that the NOPs in the first example have now been filled with other instructions. The code for the two \( Y = MX+B \) calculations are now interleaved and both calculations complete in only nine cycles.
Example 1-17. Floating-Point Code With Pipeline Optimization

```
; Using non-conflicting instructions for alignment cycles,
; calculate the following:
; Y1 = M1*X1 + B1
; Y2 = M2*X2 + B2

MOV32  R2H,@X1 ; Load R2H with X1 - single cycle
MOV32  R1H,@M1 ; Load R1H with M1 - single cycle
MPYF32 R3H,R2H,R1H ; R3H = M1 * X1 - 2p operation

| MOV32  R0H,@M2 ; Load R0H with M2 - single cycle
| MOV32  R1H,@X2 ; Load R1H with X2 - single cycle
| <-- MPYF32 completes, R3H is valid
MPYF32 R0H,R1H,R0H ; R0H = M2 * X2 - 2p operation
| MOV32  R4H,@B1 ; Load R4H with B1 - single cycle
| <-- MOV32 completes, R4H is valid
ADDF32 R1H,R4H,R3H ; R1H = B1 + M1*X1 - 2p operation
| <-- MPYF32 completes, R0H is valid
ADDF32 R0H,R2H,R0H ; R0H = B2 + M2*X2 - 2p operation
| <-- ADDF32 completes, R1H is valid
MOV32  @Y1,R1H ; Store Y1
| <-- ADDF32 completes, R0H is valid
MOV32  @Y2,R0H ; Store Y2
```

9 cycles
36 bytes

1.5 Floating Point Unit Instruction Set

This chapter describes the assembly language instructions of the TMS320C28x plus floating-point processor. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are an extension to the standard C28x instruction set. For information on standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430).

1.5.1 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Operands
- Opcode
- Description
- Exceptions
- Pipeline
- Examples
- See also

The example INSTRUCTION is shown to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information. On the C28x+FPU instructions, follow the same format as the C28x. The source operand(s) are always on the right and the destination operand(s) are on the left.

The explanations for the syntax of the operands used in the instruction descriptions for the TMS320C28x plus floating-point processor are given in Table 1-4. For information on the operands of standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430).
### Table 1-4. Operand Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>16-bit immediate (hex or float) value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FHHiHex</td>
<td>16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value.</td>
</tr>
<tr>
<td>#32Fhex</td>
<td>32-bit immediate value that represents an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32F</td>
<td>Immediate float value represented in floating-point representation</td>
</tr>
<tr>
<td>#0.0</td>
<td>Immediate zero</td>
</tr>
<tr>
<td>#RC</td>
<td>16-bit immediate value for the repeat count</td>
</tr>
<tr>
<td>&quot;(0:16bitAddr)&quot;</td>
<td>16-bit immediate address, zero extended</td>
</tr>
<tr>
<td>CNDF</td>
<td>Condition to test the flags in the STF register</td>
</tr>
<tr>
<td>FLAG</td>
<td>Selected flags from STF register (OR) 11 bit mask indicating which floating-point status flags to change</td>
</tr>
<tr>
<td>label</td>
<td>Label representing the end of the repeat block</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
<tr>
<td>RaH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RbH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RcH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RdH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>ReH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RfH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RB</td>
<td>Repeat Block Register</td>
</tr>
<tr>
<td>STF</td>
<td>FPU Status Register</td>
</tr>
<tr>
<td>VALUE</td>
<td>Flag value of 0 or 1 for selected flag (OR) 11 bit mask indicating the flag value; 0 or 1</td>
</tr>
</tbody>
</table>
INSTRUCTION dest1, source1, source2  

Short Description

Operands

<table>
<thead>
<tr>
<th>dest1</th>
<th>description for the 1st operand for the instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>source1</td>
<td>description for the 2nd operand for the instruction</td>
</tr>
<tr>
<td>source2</td>
<td>description for the 3rd operand for the instruction</td>
</tr>
</tbody>
</table>

Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).

Opcode

This section shows the opcode for the instruction.

Description

Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.

Restrictions

Any constraints on the operands or use of the instruction imposed by the processor are discussed.

Pipeline

This section describes the instruction in terms of pipeline cycles as described in Section 1.4.

Example

Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution. All examples assume the device is running with the OBJMODE set to 1. Normally the boot ROM or the c-code initialization will set this bit.

See Also

Lists related instructions.
1.5.2 Instructions

The instructions are listed alphabetically, preceded by a summary.

Table 1-5. Summary of Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSF32 RaH, RbH — 32-bit Floating-Point Absolute Value</td>
<td>34</td>
</tr>
<tr>
<td>ADDF32 RaH, #16FH, RbH — 32-bit Floating-Point Addition</td>
<td>35</td>
</tr>
<tr>
<td>ADDF32 RaH, RbH, #16FH — 32-bit Floating-Point Addition</td>
<td>37</td>
</tr>
<tr>
<td>ADDF32 RaH, RbH, RcH — 32-bit Floating-Point Addition</td>
<td>39</td>
</tr>
<tr>
<td>ADDF32 RdH, ReH, RfH</td>
<td>MOV32 mem32, RaH — 32-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>ADDF32 RdH, ReH, RfH</td>
<td>MOV32 RaH, mem32 — 32-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>CMPF32 RaH, RbH — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>45</td>
</tr>
<tr>
<td>CMPF32 RaH, #16FH — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>46</td>
</tr>
<tr>
<td>CMPF32 RaH, #0.0 — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>48</td>
</tr>
<tr>
<td>EINF32 RaH — 32-bit Floating-Point Reciprocal Approximation</td>
<td>49</td>
</tr>
<tr>
<td>EISQRTF32 RaH, RbH — 32-bit Floating-Point Square-Root Reciprocal Approximation</td>
<td>51</td>
</tr>
<tr>
<td>MOV32 RcH, RdH</td>
<td>MOV32 RdH, ReH</td>
</tr>
<tr>
<td>MOV32 RaH, mem32</td>
<td>MOV32 mem32, RaH — 32-bit Floating-Point Addition</td>
</tr>
<tr>
<td>MOV32 RaH, #16FH</td>
<td>MOV32 RaH, #16FH — 32-bit Floating-Point Minimum</td>
</tr>
<tr>
<td>I16TOF32 RaH, #16FH — 32-bit Floating-Point Minimum</td>
<td>58</td>
</tr>
<tr>
<td>I16TOF32 RaH, mem16 — 32-bit Floating-Point Minimum</td>
<td>60</td>
</tr>
<tr>
<td>I2TOF32 RaH, mem32 — 32-bit Floating-Point Minimum</td>
<td>62</td>
</tr>
<tr>
<td>MACF32 R3H, R2H, RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Add</td>
<td>64</td>
</tr>
<tr>
<td>MACF32 R3H, R2H, RdH, ReH, RfH</td>
<td>MOV32 RaH, mem32 — 32-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MACF32 R7H, R3H, mem32, *XAR7++ — 32-bit Floating-Point Multiply and Accumulate</td>
<td>68</td>
</tr>
<tr>
<td>MACF32 R7H, R6H, RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Add</td>
<td>70</td>
</tr>
<tr>
<td>MACF32 R7H, R6H, RdH, ReH, RfH</td>
<td>MOV32 RaH, mem32 — 32-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MAXF32 RaH, RbH — 32-bit Floating-Point Maximum</td>
<td>74</td>
</tr>
<tr>
<td>MAXF32 RaH, #16FH — 32-bit Floating-Point Maximum</td>
<td>75</td>
</tr>
<tr>
<td>MAXF32 RaH, RbH</td>
<td>MOV32 RcH, RdH — 32-bit Floating-Point Maximum with Parallel Move</td>
</tr>
<tr>
<td>MINF32 RaH, RbH — 32-bit Floating-Point Minimum</td>
<td>77</td>
</tr>
<tr>
<td>MINF32 RaH, #16FH — 32-bit Floating-Point Minimum</td>
<td>78</td>
</tr>
<tr>
<td>MINF32 RaH, RbH</td>
<td>MOV32 RcH, RdH — 32-bit Floating-Point Minimum with Parallel Move</td>
</tr>
<tr>
<td>MOV16 mem16, RaH — Move 16-bit Floating-Point Register Contents to Memory</td>
<td>80</td>
</tr>
<tr>
<td>MOV32 * (0:16bitAddr), loc32 — Move the Contents of loc32 to Memory</td>
<td>81</td>
</tr>
<tr>
<td>MOV32 ACC, RaH — Move 32-bit Floating-Point Register Contents to ACC</td>
<td>82</td>
</tr>
<tr>
<td>MOV32 loc32, * (0:16bitAddr) — Move 32-bit Value from Memory to loc32</td>
<td>83</td>
</tr>
<tr>
<td>MOV32 mem32, RaH — Move 32-bit Floating-Point Register Contents to Memory</td>
<td>84</td>
</tr>
<tr>
<td>MOV32 mem32, STF — Move 32-bit STF Register to Memory</td>
<td>86</td>
</tr>
<tr>
<td>MOV32 P, RaH — Move 32-bit Floating-Point Register Contents to P</td>
<td>87</td>
</tr>
<tr>
<td>MOV32 RaH, ACC — Move the Contents of ACC to a 32-bit Floating-Point Register</td>
<td>88</td>
</tr>
<tr>
<td>MOV32 RaH, mem32 (, CNDF) — Conditional 32-bit Move</td>
<td>89</td>
</tr>
</tbody>
</table>
Table 1-5. Summary of Instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV32 RaH, P</td>
<td>Move the Contents of P to a 32-bit Floating-Point Register</td>
<td>91</td>
</tr>
<tr>
<td>MOV32 RaH, RbH, (CNDF)</td>
<td>Conditional 32-bit Move</td>
<td>92</td>
</tr>
<tr>
<td>MOV32 RaH, XARn</td>
<td>Move the Contents of XARn to a 32-bit Floating-Point Register</td>
<td>93</td>
</tr>
<tr>
<td>MOV32 RaH, XT</td>
<td>Move the Contents of XT to a 32-bit Floating-Point Register</td>
<td>94</td>
</tr>
<tr>
<td>MOV32 STF, mem32</td>
<td>Move 32-bit Value from Memory to the STF Register</td>
<td>95</td>
</tr>
<tr>
<td>MOV32 XARn, RaH</td>
<td>Move 32-bit Floating-Point Register Contents to XARn</td>
<td>96</td>
</tr>
<tr>
<td>MOV32 XT, RaH</td>
<td>Move 32-bit Floating-Point Register Contents to XT</td>
<td>97</td>
</tr>
<tr>
<td>MOV32 RaH, mem32</td>
<td>Move 32-bit Value from Memory with Data Copy</td>
<td>98</td>
</tr>
<tr>
<td>MOV32 RaH, #32F</td>
<td>Load the 32-bits of a 32-bit Floating-Point Register</td>
<td>99</td>
</tr>
<tr>
<td>MOV32 RaH, #32FHex</td>
<td>Load the 32-bits of a 32-bit Floating-Point Register with the immediate</td>
<td>100</td>
</tr>
<tr>
<td>MOVIZ RaH, #16FHiHex</td>
<td>Load the Upper 16-bits of a 32-bit Floating-Point Register</td>
<td>101</td>
</tr>
<tr>
<td>MOVIZF32 RaH, #16FHi</td>
<td>Load the Upper 16-bits of a 32-bit Floating-Point Register</td>
<td>102</td>
</tr>
<tr>
<td>MOVST0 FLAG</td>
<td>Load Selected STF Flags into ST0</td>
<td>103</td>
</tr>
<tr>
<td>MOVXI RaH, #16FLoHex</td>
<td>Move Immediate to the Low 16-bits of a Floating-Point Register</td>
<td>104</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RcH</td>
<td>32-bit Floating-Point Multiply</td>
<td>105</td>
</tr>
<tr>
<td>MPYF32 RaH, #16FHi, RbH</td>
<td>32-bit Floating-Point Multiply</td>
<td>106</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, #16FHi</td>
<td>32-bit Floating-Point Multiply</td>
<td>108</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RcH, ADDF32 RdH, ReH, RFH</td>
<td>32-bit Floating-Point Multiply with Parallel Add</td>
<td>110</td>
</tr>
<tr>
<td>MPYF32 RdH, ReH, RFH, MOV32 RaH, mem32</td>
<td>32-bit Floating-Point Multiply with Parallel Move</td>
<td>112</td>
</tr>
<tr>
<td>MPYF32 RdH, ReH, RFH, MOV32 mem32, RaH</td>
<td>32-bit Floating-Point Multiply with Parallel Move</td>
<td>114</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RcH, SUBF32 RdH, ReH, RFH</td>
<td>32-bit Floating-Point Multiply with Parallel Subtract</td>
<td>115</td>
</tr>
<tr>
<td>NEGF32 RaH, RbH, (CNDF)</td>
<td>Conditional Negation</td>
<td>116</td>
</tr>
<tr>
<td>POP RB</td>
<td>Pop the RB Register from the Stack</td>
<td>117</td>
</tr>
<tr>
<td>PUSH RB</td>
<td>Push the RB Register onto the Stack</td>
<td>119</td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore the Floating-Point Registers</td>
<td>120</td>
</tr>
<tr>
<td>RPTB label, loc16</td>
<td>Repeat A Block of Code</td>
<td>122</td>
</tr>
<tr>
<td>RPTB label, #RC</td>
<td>Repeat a Block of Code</td>
<td>124</td>
</tr>
<tr>
<td>SAVE FLAG, VALUE</td>
<td>Save Register Set to Shadow Registers and Execute SETFLG</td>
<td>126</td>
</tr>
<tr>
<td>SETFLG FLAG, VALUE</td>
<td>Set or clear selected floating-point status flags</td>
<td>128</td>
</tr>
<tr>
<td>SUBF32 RaH, RbH, RcH</td>
<td>32-bit Floating-Point Subtraction</td>
<td>129</td>
</tr>
<tr>
<td>SUBF32 RaH, #16FHi, RbH</td>
<td>32-bit Floating Point Subtraction</td>
<td>130</td>
</tr>
<tr>
<td>SUBF32 RdH, ReH, RFH, MOV32 RaH, mem32</td>
<td>32-bit Floating-Point Subtraction with Parallel Move</td>
<td>131</td>
</tr>
<tr>
<td>SUBF32 RdH, ReH, RFH, MOV32 mem32, RaH</td>
<td>32-bit Floating-Point Subtraction with Parallel Move</td>
<td>133</td>
</tr>
<tr>
<td>SWAPF RaH, RbH, (CNDF)</td>
<td>Conditional Swap</td>
<td>135</td>
</tr>
<tr>
<td>TESTTF CNDF</td>
<td>Test STF Register Flag Condition</td>
<td>136</td>
</tr>
<tr>
<td>UI16TOF32 RaH, mem16</td>
<td>Convert unsigned 16-bit integer to 32-bit floating-point value</td>
<td>137</td>
</tr>
<tr>
<td>UI16TOF32 RaH, RbH</td>
<td>Convert unsigned 16-bit integer to 32-bit floating-point value</td>
<td>138</td>
</tr>
<tr>
<td>UI32TOF32 RaH, mem32</td>
<td>Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value</td>
<td>139</td>
</tr>
<tr>
<td>UI32TOF32 RaH, RbH</td>
<td>Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value</td>
<td>140</td>
</tr>
<tr>
<td>ZERO RaH</td>
<td>Zero the Floating-Point Register RaH</td>
<td>141</td>
</tr>
<tr>
<td>ZEROA</td>
<td>Zero All Floating-Point Registers</td>
<td>142</td>
</tr>
</tbody>
</table>
**ABSF32 RaH, RbH**  
32-bit Floating-Point Absolute Value

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0110 1001 0101</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

**Description**

The absolute value of RbH is loaded into RaH. Only the sign bit of the operand is modified by the ABSF32 instruction.

\[
\text{if (RbH < 0)} \{\text{RaH} = -\text{RbH}\} \\
\text{else} \{\text{RaH} = \text{RbH}\}
\]

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

\[
\text{NF} = 0; \\
\text{ZF} = 0; \\
\text{if (RaH}[30:23] == 0) ZF = 1;
\]

**Pipeline**

This is a single-cycle instruction.

**Example**

MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
ABSF32 R1H, R1H ; R1H = 2.0 (0x40000000), ZF = NF = 0

MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
ABSF32 R0H, R0H ; R0H = 5.0 (0x40A00000), ZF = NF = 0

MOVIZF32 R0H, #0.0 ; R0H = 0.0
ABSF32 R1H, R0H ; R1H = 0.0 ZF = 1, NF = 0

**See also**

NEGF32 RaH, RbH{, CNDF}
ADDF32 RaH, #16FHi, RbH  

32-bit Floating-Point Addition

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 10II IIII
MSW: IIII IIII IIbb baaa

Description

Add RbH to the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value.

RaH = RbH + #16FHi:0

This instruction can also be written as ADDF32 RaH, RbH, #16FHi.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

```
ADDF32 RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; ADDF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

; Add to R1H the value 2.0 in 32-bit floating-point format
ADDF32 R0H, #2.0, R1H ; R0H = 2.0 + R1H
NOP ; Delay for ADDF32 to complete
    ; ADDF32 completes, R0H updated
NOP

; Add to R3H the value -2.5 in 32-bit floating-point format
ADDF32 R2H, #-2.5, R3H ; R2H = -2.5 + R3H
NOP ; Delay for ADDF32 to complete
    ; ADDF32 completes, R2H updated
NOP

; Add to R5H the value 0x3FC00000 (1.5)
ADDF32 R5H, #0x3FC0, R5H ; R5H = 1.5 + R5H
NOP ; Delay for ADDF32 to complete
    ; ADDF32 completes, R5H updated

Copyright © 2014–2019, Texas Instruments Incorporated
ADDF32 RaH, #16FHi, RbH — 32-bit Floating-Point Addition

NOP ;

See also

ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
ADD32 RaH, RbH, #16FHi — 32-bit Floating-Point Addition

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 10II IIII
MSW: IIII IIII IIbb baaa

Description

Add RbH to the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

RaH = RbH + #16FHi:0

This instruction can also be written as ADDF32 RaH, #16FHi, RbH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

```
ADDF32 RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
      ; <-- ADDF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
; Add to R1H the value 2.0 in 32-bit floating-point format
ADDF32 ROH, R1H, #2.0 ; R0H = R1H + 2.0
NOP ; Delay for ADDF32 to complete
      ; <-- ADDF32 completes, R0H updated
NOP

; Add to R3H the value -2.5 in 32-bit floating-point format
ADDF32 R2H, R3H, #-2.5 ; R2H = R3H + (-2.5)
NOP ; Delay for ADDF32 to complete
      ; <-- ADDF32 completes, R2H updated
NOP

; Add to R5H the value 0x3FC00000 (1.5)
ADDF32 R5H, R5H, #0x3FC0 ; R5H = R5H + 1.5
NOP ; Delay for ADDF32 to complete
      ; <-- ADDF32 completes, R5H updated
```
ADDF32 RaH, RbH, #16FHi — 32-bit Floating-Point Addition

See also

ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
ADDF32 RaH, RbH, RcH  32-bit Floating-Point Addition

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0001 0000
MSW: 0000 000c ccbb baaa

Description

Add the contents of RcH to the contents of RbH and load the result into RaH.

RaH = RbH + RcH

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

ADDF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- ADDF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate Y = M1*X1 + B1. This example assumes that M1, X1, B1 and Y are all on the same data page.

```
MOVW DP, #M1 ; Load the data page
MOV32 R0H, @M1 ; Load R0H with M1
MOV32 R1H, @X1 ; Load R1H with X1
MPYF32 R1H, R1H, R0H ; Multiply M1*X1
|| MOV32 R0H, @B1 ; and in parallel load R0H with B1
NOP ; <-- MOV32 complete
; <-- MPYF32 complete
ADDF32 R1H, R1H, R0H ; Add M*X1 to B1 and store in R1H
NOP ; <-- ADDF32 complete
MOV32 @Y1, R1H ; Store the result
```

Calculate Y = A + B

```
MOVL XAR4, #A
MOV32 R0H, *XAR4 ; Load R0H with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
ADDF32 R0H, R1H, R0H ; Add A + B R0H+R0H+R1H
MOVL XAR4, #Y
; <-- ADDF32 complete
MOV32 *XAR4, R0H ; Store the result
```

See also

ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, #16F, RbH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RaH, RbH, RcH — 32-bit Floating-Point Addition

```
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
```
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH  
32-bit Floating-Point Addition with Parallel Move

**Operands**

| RdH | floating-point destination register for the ADDF32 (R0H to R7H) |
| ReH | floating-point source register for the ADDF32 (R0H to R7H) |
| RfH | floating-point source register for the ADDF32 (R0H to R7H) |
| mem32 | pointer to a 32-bit memory location. This will be the destination of the MOV32. |
| RaH | floating-point source register for the MOV32 (R0H to R7H) |

**Opcode**

LSW: 1110 0000 0001 fffe  
MSW: eedd daaa mem32

**Description**

Perform an ADDF32 and a MOV32 in parallel. Add RfH to the contents of ReH and store the result in RdH. In parallel move the contents of RaH to the 32-bit location pointed to by mem32. mem32 addresses memory using any of the direct or indirect addressing modes supported by the C28x CPU.

RdH = ReH + RfH,  
[mem32] = RaH

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

**Pipeline**

ADDF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

ADDF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)  
|| MOV32 mem32, RaH ; 1 cycle  
| NOP | 1 cycle delay or non-conflicting instruction  
| NOP |

Any instruction in the delay slot must not use RdH as a destination register or use RdH as a source operand.

**Example**

ADDF32 R3H, R6H, R4H ; (A) R3H = R6H + R4H and R7H = I3  
|| MOV32 R7H, *-SP[2] ;  
| SUBF32 R6H, R6H, R4H ; (B) R6H = R6H - R4H  
| SUBF32 R3H, R1H, R7H ; (C) R3H = R1H - R7H and store R3H (A)  
|| MOV32 *+XAR5[2], R3H ;  
| ADDF32 R4H, R7H, R1H ; R4H = D = R7H + R1H and store R6H (B)  
|| MOV32 *+XAR5[6], R6H ;  
| MOV32 *+XAR5[0], R3H ; store R3H (C)  
| MOV32 *+XAR5[4], R4H ; store R4H (D);
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH — 32-bit Floating-Point Addition with Parallel Move

; <-- MOV32 completes, (D) stored

See also

ADDF32 RaH, #16FHi, RbH
ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH
|| MOV32 RaH, mem32 — 32-bit Floating-Point Addition with Parallel Move

Operands

| RdH | floating-point destination register for the ADDF32 (R0H to R7H). RdH cannot be the same register as RaH. |
| ReH | floating-point source register for the ADDF32 (R0H to R7H) |
| RfH | floating-point source register for the ADDF32 (R0H to R7H) |
| RaH | floating-point destination register for the MOV32 (R0H to R7H). RaH cannot be the same register as RdH. |
| mem32 | pointer to a 32-bit memory location. This is the source for the MOV32. |

Opcode

LSW: 1110 0011 0001 fffe
MSW: eedd daaa mem32

Description

Perform an ADDF32 and a MOV32 operation in parallel. Add RfH to the contents of ReH and store the result in RdH. In parallel move the contents of the 32-bit location pointed to by mem32 to RaH. mem32 addresses memory using any of the direct or indirect addressing modes supported by the C28x CPU.

RdH = ReH + RfH,
RaH = [mem32]

Restrictions

The destination register for the ADDF32 and the MOV32 must be unique. That is, RaH and RdH cannot be the same register.

Any instruction in the delay slot must not use RdH as a destination register or use RdH as a source operand.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;

Pipeline

The ADDF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32 ; 2 pipeline cycles (2p)
|| MOV32 RaH, mem32 ; 1 cycle
|| MOV32 completes, RaH updated NOP ; 1 cycle delay or non-conflicting instruction
|| ADDF32 completes, RdH updated NOP
Example

Calculate \( Y = A + B - C \):

```assembly
MOVL XAR4, #A
MOV32 R0H, *XAR4 ; Load R0H with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
MOVL XAR4, #C
ADDF32 R0H,R1H,R0H ; Add A + B and in parallel
|| MOV32 R2H, *XAR4 ; Load R2H with C
; <-- MOV32 complete
MOVL XAR4,#Y
; ADDF32 complete
SUBF32 R0H,R0H,R2H ; Subtract C from (A + B)
NOP ; <-- SUBF32 completes
MOV32 *XAR4,R0H ; Store the result
```

See also

ADDF32 RaH, #16FHi, RbH
ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
### CMPF32 RaH, RbH — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than

#### Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

#### Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1001 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

#### Description

Set ZF and NF flags on the result of RaH - RbH. The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE format offsetting the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

- Negative zero will be treated as positive zero.
- A denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

#### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- If(RaH == RbH) {ZF=1, NF=0}
- If(RaH > RbH) {ZF=0, NF=0}
- If(RaH < RbH) {ZF=0, NF=1}

#### Pipeline

This is a single-cycle instruction.

#### Example

; Behavior of ZF and NF flags for different comparisons

```assembly
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
CMPF32 R1H, R0H ; ZF = 0, NF = 1
CMPF32 R0H, R1H ; ZF = 0, NF = 0
CMPF32 R0H, R0H ; ZF = 1, NF = 0

; Using the result of a compare for loop control

Loop:
MOV32 R0H,*XAR4++ ; Load R0H
MOV32 R1H,*XAR3++ ; Load R1H
CMPF32 R1H, R0H ; Set/clear ZF and NF
MOVST ZF, NF ; Copy ZF and NF to ST0 Z and N bits
BF Loop, GT ; Loop if R1H > R0H
```

#### See also

- CMPF32 RaH, #16FHi
- CMPF32 RaH, #0.0
- MAXF32 RaH, #16FHi
- MAXF32 RaH, RbH
- MINF32 RaH, #16FHi
- MINF32 RaH, RbH
CMPF32 RaH, #16FHi — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

| RaH | floating-point source register (R0H to R7H) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

| LSW: | 1110 1000 0001 0III |
| MSW: | IIII IIII IIII Iaaa |

Description

Compare the value in RaH with the floating-point value represented by the immediate operand. Set the ZF and NF flags on (RaH - #16FHi:0).

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as # -1.5 or #0xBFC0.

The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:
- Negative zero will be treated as positive zero.
- Denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

If(RaH == #16FHi:0) {ZF=1, NF=0}
If(RaH > #16FHi:0) {ZF=0, NF=0}
If(RaH < #16FHi:0) {ZF=0, NF=1}

Pipeline

This is a single-cycle instruction

Example

; Behavior of ZF and NF flags for different comparisons
MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
CMPF32 R1H, #.2.2 ; ZF = 0, NF = 0
CMPF32 R0H, #6.5 ; ZF = 0, NF = 1
CMPF32 R0H, #5.0 ; ZF = 1, NF = 0

; Using the result of a compare for loop control
Loop:
MOV32 R1H,*XAR3++ ; Load R1H
CMPF32 R1H, #2.0 ; Set/clear ZF and NF
MOVST0 ZF, NF ; Copy ZF and NF to ST0 Z and N bits
BF Loop, GT ; Loop if R1H > #2.0

See also

CMPF32 RaH, #0.0
CMPF32 RaH, RbH
MAXF32 RaH, #16FHi
MAXF32 RaH, RbH
MINF32 RaH, #16FHi
MINF32 RaH, RbH
CMPF32 RaH, #0.0 — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0.0</td>
<td>zero</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1010 0aaa

Description

Set the ZF and NF flags on (RaH - #0.0). The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

- Negative zero will be treated as positive zero.
- Denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- If(RaH == #0.0) {ZF=1, NF=0}
- If(RaH > #0.0) {ZF=0, NF=0}
- If(RaH < #0.0) {ZF=0, NF=1}

Pipeline

This is a single-cycle instruction.

Example

; Behavior of ZF and NF flags for different comparisons
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
MOVIZF32 R2H, #0.0 ; R2H = 0.0 (0x00000000)
CMPF32 R0H, #0.0 ; ZF = 0, NF = 0
CMPF32 R1H, #0.0 ; ZF = 0, NF = 1
CMPF32 R2H, #0.0 ; ZF = 0, NF = 0

; Using the result of a compare for loop control
Loop:
MOV32 R1H,*XAR3++ ; Load R1H
CMPF32 R1H, #0.0 ; Set/clear ZF and NF
MOVST0 ZF, NF ; Copy ZF and NF to ST0 Z and N bits
BF Loop, GT ; Loop if R1H > #0.0

See also

- CMPF32 RaH, #0.0
- CMPF32 RaH, #16FHi
- MAXF32 RaH, #16FHi
- MAXF32 RaH, RbH
- MINF32 RaH, #16FHi
- MINF32 RaH, RbH
EINVF32 RaH, RbH — 32-bit Floating-Point Reciprocal Approximation

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 0110 1001 0011
- MSW: 0000 0000 00bb baaa

Description

This operation generates an estimate of 1/X in 32-bit floating-point format accurate to approximately 8 bits. This value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

\[
Ye = \text{Estimate}(1/X); \\
Ye = Ye \times (2.0 - Ye \times X) \\
Ye = Ye \times (2.0 - Ye \times X)
\]

After two iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to the 32-bit floating-point format. On each iteration the mantissa bit accuracy approximately doubles. The EINVF32 operation will not generate a negative zero, DeNorm or NaN value.

RaH = Estimate of 1/RbH

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if EINVF32 generates an underflow condition.
- LVF = 1 if EINVF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
EINVF32 RaH, RbH ; 2p
NOP ; 1 cycle delay or non-conflicting instruction
; <-- EINVF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.
Example

Calculate $Y = A/B$. A fast division routine similar to that shown below can be found in the C28x FPU Fast RTS Library (SPRC664).

```
MOVL XAR4, #A
MOVL XAR4, #B
LCR DIV ; Calculate R0H = R0H / R1H
MOV32 *XAR4, R0H ; ....

DIV:
EINVF32 R2H, R1H ; R2H = Ye = Estimate(1/B)
CMPF32 R0H, #0.0 ; Check if A == 0
MPYF32 R3H, R2H, R1H ; R3H = Ye*B
NOP
SUBF32 R3H, #2.0, R3H ; R3H = 2.0 - Ye*B
NOP
MPYF32 R2H, R2H, R3H ; R2H = Ye = Ye*(2.0 - Ye*B)
NOP
MPYF32 R3H, R2H, R1H ; R3H = Ye*B
CMPF32 R1H, #0.0 ; Check if B == 0.0
NEGF32 R0H, R0H, EQ ; Fixes sign for A/0.0
MPYF32 R2H, R2H, R3H ; R2H = Ye = Ye*(2.0 - Ye*B)
NOP
MPYF32 R0H, R0H, R2H ; R0H = Y = A*Ye = A/B
LRETR
```

See also

EISQRTF32 RaH, RbH
EISQRTF32 RaH, RbH  

32-bit Floating-Point Square-Root Reciprocal Approximation

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 0010  
MSW: 0000 0000 00bb baaa

Description

This operation generates an estimate of 1/\sqrt{X} in 32-bit floating-point format accurate to approximately 8 bits. This value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

\[
Y_e = \text{Estimate}(1/\sqrt{X});  
Y_e = Y_e(1.5 - Y_eY_eX/2.0)
\]

After 2 iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to the 32-bit floating-point format. On each iteration the mantissa bit accuracy approximately doubles. The EISQRTF32 operation will not generate a negative zero, DeNorm or NaN value.

RaH = Estimate of 1/\sqrt{X}(RbH)

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if EISQRTF32 generates an underflow condition.
- LVF = 1 if EISQRTF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

\[
\text{EINVF32} \text{ RaH, RbH} ; 2 \text{ pipeline cycles (2p)}  
\text{NOP} ; 1 \text{ cycle delay or non-conflicting instruction}  
\text{NOP} ; \text{ <-- EISQRTF32 completes, RaH updated}  
\text{NOP}
\]

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.
Example

Calculate the square root of X. A square-root routine similar to that shown below can be found in the C28x FPU Fast RTS Library (SPRC664).

```c
; Y = sqrt(X)
; Ye = Estimate(1/sqrt(X));
; Ye = Ye*(1.5 - Ye*Ye*X*0.5)
; Ye = Ye*(1.5 - Ye*Ye*X*0.5)
; Y = X*Ye
__sqrt:
    ; R0H = X on entry
    EISQRTF32 R1H, R0H ; R1H = Ye = Estimate(1/sqrt(X))
    MPYF32 R2H, R0H, #0.5 ; R2H = X*0.5
    MPYF32 R3H, R1H, R1H ; R3H = Ye*Ye
    NOP
    MPYF32 R3H, R3H, R2H ; R3H = Ye*Ye*X*0.5
    NOP
    SUBF32 R3H, #1.5, R3H ; R3H = 1.5 - Ye*Ye*X*0.5
    NOP
    MPYF32 R1H, R1H, R3H ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    NOP
    MPYF32 R3H, R1H, R2H ; R3H = Ye*X*0.5
    NOP
    MPYF32 R3H, R1H, R3H ; R3H = Ye*Ye*X*0.5
    NOP
    SUBF32 R3H, #1.5, R3H ; R3H = 1.5 - Ye*Ye*X*0.5
    CMPF32 R0H, #0.0 ; Check if X == 0
    MPYF32 R1H, R1H, R3H ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    NOP
    MOV32 R1H, R0H, EQ ; If X is zero, change the Ye estimate to 0
    MPYF32 R0H, R0H, R1H ; R0H = Y = X*Ye = sqrt(X)
    LRETR

See also
EINVF32 RaH, RbH
F32TOI16 RaH, RbH  
*Convert 32-bit Floating-Point Value to 16-bit Integer*

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1000 1100
MSW: 0000 0000 00bb baaa

**Description**

Convert a 32-bit floating point value in RbH to a 16-bit integer and truncate. The result will be stored in RaH.

RaH(15:0) = F32TOI16(RbH)
RaH(31:16) = sign extension of RaH(15)

**Flags**

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

F32TOI16 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

**Example**

MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
F32TOI16 R1H, R0H ; R1H(15:0) = F32TOI16(R0H)
; R1H(31:16) = Sign extension of R1H(15)
MOVIZF32 R2H, #-5.0 ; R2H = -5.0 (0xC0A00000)
; <-- F32TOI16 complete, R1H(15:0) = 5 (0x00005)
; R1H(31:16) = 0 (0x0000)
F32TOI16 R3H, R2H ; R3H(15:0) = F32TOI16(R2H)
; R3H(31:16) = Sign extension of R3H(15)
NOP ; 1 Cycle delay for F32TOI16 to complete
; <-- F32TOI16 complete, R3H(15:0) = -5 (0xFFFBB)
; R3H(31:16) = (0xFFFF)

**See also**

F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH
### F32TOI16R RaH, RbH

**Convert 32-bit Floating-Point Value to 16-bit Integer and Round**

#### Operands

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

#### Opcode

```
LSW: 1110 0110 1000 1100
MSW: 1000 0000 00bb baaa
```

#### Description

Convert the 32-bit floating point value in RbH to a 16-bit integer and round to the nearest even value. The result is stored in RaH.

- RaH(15:0) = F32ToI16round(RbH)
- RaH(31:16) = sign extension of RaH(15)

#### Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOI16R RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- F32TOI16R completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

#### Example

```
MOVIZ R0H, #0x3FD9 ; R0H [31:16] = 0x3FD9
MOVXI R0H, #0x999A ; R0H [15:0] = 0x999A
; R0H = 1.7 (0x3FD9999A)
F32TOI16R R1H, R0H ; R1H(15:0) = F32TOI16round (R0H)
; R1H(31:16) = Sign extension of R1H(15)
MOVF32 R2H, #1.7 ; R2H = -1.7 (0xFBFD9999A)
; <- F32TOI16R complete, R1H(15:0) = 2 (0x0002)
; R1H(31:16) = 0 (0x0000)
F32TOI16R R3H, R2H ; R3H(15:0) = F32TOI16round (R2H)
; R3H(31:16) = Sign extension of R2H(15)
NOP ; 1 Cycle delay for F32TOI16R to complete
; <-- F32TOI16R complete, R1H(15:0) = -2 (0xFFFE)
; R1H(31:16) = (0xFFFF)
```

#### See also

- `F32TOI16 RaH, RbH`
- `F32TOUI16 RaH, RbH`
- `F32TOUI16R RaH, RbH`
- `I16TOF32 RaH, RbH`
- `I16TOF32 RaH, mem16`
- `UI16TOF32 RaH, mem16`
- `UI16TOF32 RaH, RbH`
**F32TOI32 RaH, RbH — Convert 32-bit Floating-Point Value to 32-bit Integer**

### Operands
- **RaH**: floating-point destination register (R0H to R7H)
- **RbH**: floating-point source register (R0H to R7H)

### Opcode
- **LSW**: 1110 0110 1000 1000
- **MSW**: 0000 0000 00bb baaa

### Description
Convert the 32-bit floating-point value in RbH to a 32-bit integer value and truncate. Store the result in RaH.

RaH = F32TOI32(RbH)

### Flags
This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Pipeline
This is a 2 pipeline cycle (2p) instruction. That is:

- **F32TOI32 RaH, RbH**: 2 pipeline cycles (2p)
- **NOP**: 1 cycle delay or non-conflicting instruction
- **NOP**: \(\text{<- F32TOI32 completes, RaH updated}\)

- **NOP**

   Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

### Example

```
MOVF32 R2H, #11204005.0  ; R2H = 11204005.0 (0x4B2AF5A5)
F32TOI32 R3H, R2H        ; R3H = F32TOI32 (R2H)
MOVF32 R4H, #-11204005.0 ; R4H = -11204005.0 (0xCB2AF5A5)
; \(\text{<- F32TOI32 complete, R3H = 11204005 (0x00AAF5A5)}\)
F32TOI32 R5H, R4H        ; R5H = F32TOI32 (R4H)
NOP                       ; 1 Cycle delay for F32TOI32 to complete
; \(\text{<- F32TOI32 complete, R5H = -11204005 (0xFF550A5B)}\)
```

### See also
- F32TOUI32 RaH, RbH
- I32TOF32 RaH, RbH
- I32TOF32 RaH, mem32
- UI32TOF32 RaH, RbH
- UI32TOF32 RaH, mem32
**F32TOUI16 RaH, RbH** — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer

### Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0110 1000 1110</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

### Description

Convert the 32-bit floating point value in RbH to an unsigned 16-bit integer value and truncate to zero. The result will be stored in RaH. To instead round the integer to the nearest even value use the F32TOUI16R instruction. The instruction will saturate the float to what can fit in 16-bit integer and then convert to 16-bit. For example 300000 will be saturated to 65535.

RaH(15:0) = F32ToUI16(RbH) RaH(31:16) = 0x0000

### Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOUI16 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- F32TOUI16 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

### Example

```
MOVIZF32 R4H, #9.0 ; R4H = 9.0 (0x41100000)
F32TOUI16 R5H, R4H ; R5H (15:0) = F32TOUI16 (R4H)
    ; R5H (31:16) = 0x0000
MOVIZF32 R6H, # -9.0 ; R6H = -9.0 (0xC1100000)
    ; <-- F32TOUI16 complete, R5H (15:0) = 9.0 (0x0009)
    ;        R5H (31:16) = 0.0 (0x0000)
F32TOUI16 R7H, R6H ; R7H (15:0) = F32TOUI16 (R6H)
    ; R7H (31:16) = 0x0000
NOP ; 1 cycle delay for F32TOUI16 to complete
    ;        F32TOUI16 complete, R7H (15:0) = 0.0 (0x0000)
    ;        R7H (31:16) = 0.0 (0x0000)
```

### See also

- F32TOI16 RaH, RbH
- F32TOUI16R RaH, RbH
- I16TOF32 RaH, RbH
- I16TOF32 RaH, mem16
- UI16TOF32 RaH, mem16
- UI16TOF32 RaH, RbH
F32TOUI16R RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1110
MSW: 1000 0000 00bb baaa

Description

Convert the 32-bit floating-point value in RbH to an unsigned 16-bit integer and round to the closest even value. The result will be stored in RaH. To instead truncate the converted value, use the F32TOUI16 instruction. The instruction will saturate the float to what can fit in 16bit integer and then convert to 16bit. For example 300000 will be saturated to 65535.

RaH(15:0) = F32ToUI16round(RbH)
RaH(31:16) = 0x0000

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

F32TOUI16R RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
      ; <-- F32TOUI16R completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVIZ R5H, #0x412C ; R5H = 0x412C
MOVX1 R5H, #0xCCCD ; R5H = 0xCCCD
      ; R5H = 10.8 (0x412CCCCD)
F32TOUI16R R6H, R5H ; R6H (15:0) = F32TOUI16round (R5H)
      ; R6H (31:16) = 0x0000
MOVF32 R7H, #-10.8 ; R7H = -10.8 (0x0xc12CCCCD)
      ; <-- F32TOUI16R complete,
      ; R6H (15:0) = 11.0 (0x0000B)
      ; R6H (31:16) = 0.0 (0x0000)
F32TOUI16R R0H, R7H ; R0H (15:0) = F32TOUI16round (R7H)
      ; R0H (31:16) = 0x0000
NOP ; 1 Cycle delay for F32TOUI16R to complete
      ; <-- F32TOUI16R complete,
      ; R0H (15:0) = 0.0 (0x0000)
      ; R0H (31:16) = 0.0 (0x0000)

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH
F32TOUI32 RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer

Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1010
MSW: 0000 0000 00bb baaa

Description

Convert the 32-bit floating-point value in RbH to an unsigned 32-bit integer and store the result in RaH.

RaH = F32ToUI32(RbH)

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

F32TOUI32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- F32TOUI32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVIZF32 R6H, #12.5 ; R6H = 12.5 (0x41480000)
F32TOUI32 R7H, R6H ; R7H = F32TOUI32 (R6H)
MOVIZF32 R1H, #-6.5 ; R1H = -6.5 (0xC0D00000)
    ; <-- F32TOUI32 complete, R7H = 12.0 (0x0000000C)
F32TOUI32 R2H, R1H ; R2H = F32TOUI32 (R1H)
NOP ; 1 Cycle delay for F32TOUI32 to complete
    ; <-- F32TOUI32 complete, R2H = 0.0 (0x00000000)

See also

F32TOI32 RaH, RbH
I32TOF32 RaH, RbH
I32TOF32 RaH, mem32
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32
FRACF32 RaH, RbH — Fractional Portion of a 32-bit Floating-Point Value

Operands

| RaH | floating-point destination register (R0H to R7H) |
| RbH | floating-point source register (R0H to R7H) |

Opcode

LSW: 1110 0110 1111 0001
MSW: 0000 0000 00bb baaa

Description

Returns in RaH the fractional portion of the 32-bit floating-point value in RbH

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

\[
\text{FRACF32 RaH, RbH} ; 2 \text{ pipeline cycles (2p)}
\]

\[
\text{NOP} ; \text{1 cycle delay or non-conflicting instruction}
\]

\[
\langle-- \text{FRACF32 completes, RaH updated}
\]

\[
\text{NOP}
\]

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

\[
\text{MOVIZF32 R2H, } #19.625 ; \text{R2H = 19.625 (0x419D0000)}
\]

\[
\text{FRACF32 R3H, R2H} ; \text{R3H = FRACF32 (R2H)}
\]

\[
\text{NOP} ; \text{1 Cycle delay for FRACF32 to complete}
\]

\[
\langle-- \text{FRACF32 complete, R3H = 0.625 (0x3F200000)}
\]

See also
I16TOF32 RaH, RbH — Convert 16-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1101
MSW: 0000 0000 00bb baaa

Description

Convert the 16-bit signed integer in RbH to a 32-bit floating point value and store the result in RaH.

RaH = I16ToF32 RbH

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

I16TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- I16TOF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVIZ R0H, #0x0000 ; R0H[31:16] = 0.0 (0x0000)
MOVXI R0H, #0x0004 ; R0H[15:0] = 4.0 (0x0004)
I16TOF32 R1H, R0H ; R1H = I16TOF32 (R0H)
MOVIZ R2H, #0x0000 ; R2H[31:16] = 0.0 (0x0000)
    ; <--I16TOF32 complete, R1H = 4.0 (0x40800000)
MOVXI R2H, #0xFFFC ; R2H[15:0] = -4.0 (0xFFFC) I16TOF32 R3H, R2H ; R3H = I16TOF32 (R2H)
NOP ; 1 Cycle delay for I16TOF32 to complete
    ; <-- I16TOF32 complete, R3H = -4.0 (0xC0800000)

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH
I16TOF32 RaH, mem16 — Convert 16-bit Integer to 32-bit Floating-Point Value

Operands

| RaH       | floating-point destination register (R0H to R7H) |
| mem316    | 16-bit source memory location to be converted |

Opcode

LSW: 1110 0010 1100 1000  
MSW: 0000 0aaa mem16  

Description

Convert the 16-bit signed integer indicated by the mem16 pointer to a 32-bit floating-point value and store the result in RaH.

RaH = I16ToF32[mem16]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

I16TOF32 RaH, mem16 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction  
; <-- I16TOF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVW DP, #0x0280 ; DP = 0x0280
MOV @0, #0x0004 ; [0x00A000] = 4.0 (0x0004)
I16TOF32 R0H, @0 ; R0H = I16TOF32 [0x00A000]
MOV @1, #0xFFFFC ; [0x00A001] = -4.0 (0xFFFFC)
; <--I16TOF32 complete, R0H = 4.0 (0x0004)
I16TOF32 R1H, @1 ; R1H = I16TOF32 [0x00A001]
NOP ; 1 Cycle delay for I16TOF32 to complete  
; <-- I16TOF32 complete, R1H = -4.0 (0x0004)

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH
I32TOF32 RaH, mem32 — Convert 32-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>32-bit source for the MOV32 operation. mem32 means that the operation can only address memory using any of the direct or indirect addressing modes supported by the C28x CPU</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 1000
MSW: 0000 0aaa mem32

Description

Convert the 32-bit signed integer indicated by the mem32 pointer to a 32-bit floating point value and store the result in RaH.

RaH = I32ToF32[mem32]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

I32TOF32 RaH, mem32 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVW DP, #0x0280 ; DP = 0x0280
MOV @0, #0x1111 ; [0x00A000] = 4369 (0x1111)
MOV @1, #0x1111 ; [0x00A001] = 4369 (0x1111)
; Value of the 32 bit signed integer present in
; 0x00A001 and 0x00A000 is +286331153 (0x11111111)
I32TOF32 R1H, @0 ; R1H = I32ToF32 (0x11111111)
NOP ; 1 Cycle delay for I32TOF32 to complete
; <-- I32TOF32 complete, R1H = 286331153 (0x4D888888)

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, RbH
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32
I32TOF32 RaH, RbH — Convert 32-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1001
MSW: 0000 0000 00bb baaa

Description

Convert the signed 32-bit integer in RbH to a 32-bit floating-point value and store the result in RaH.

RaH = I32ToF32(RbH)

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

I32TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- I32TOF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVIZ R2H, #0x1111 ; R2H[31:16] = 4369 (0x1111)
MOVXI R2H, #0x1111 ; R2H[15:0] = 4369 (0x1111)
; Value of the 32 bit signed integer present
; in R2H is +286331153 (0x11111111)
I32TOF32 R3H, R2H ; R3H = I32TOF32 (R2H)
NOP ; 1 Cycle delay for I32TOF32 to complete
; <-- I32TOF32 complete, R3H = 286331153 (0x4D888888)

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, mem32
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32
MACF32 R3H, R2H, RdH, ReH, RfH 32-bit Floating-Point Multiply with Parallel Add

Operands
This instruction is an alias for the parallel multiply and add instruction. The operands are translated by the assembler such that the instruction becomes:

MPYF32 RdH, RaH, RbH
|| ADDF32 R3H, R3H, R2H

- **R3H**: floating-point destination and source register for the ADDF32
- **R2H**: floating-point source register for the ADDF32 operation (R0H to R7H)
- **RdH**: floating-point destination register for MPYF32 operation (R0H to R7H)
  - RdH cannot be R3H
- **ReH**: floating-point source register for MPYF32 operation (R0H to R7H)
- **RfH**: floating-point source register for MPYF32 operation (R0H to R7H)

Opcode

| LSW: 1110 0111 0100 00ff |
| MSW: feee dddc ccbb baaa |

Description
This instruction is an alias for the parallel multiply and add, MACF32 || ADDF32, instruction.

RdH = ReH * RfH
R3H = R3H + R2H

Restrictions
The destination register for the MPYF32 and the ADDF32 must be unique. That is, RdH cannot be R3H.

Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline
Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

MPYF32 RaH, RbH, Rch ; 2 pipeline cycles (2p)
|| ADDF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.
Example

; Perform 5 multiply and accumulate operations:

; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3

; Result = A + B + C + D + E

MOV32 R0H, *XAR4++       ; R0H = X0
MOV32 R1H, *XAR5++        ; R1H = Y0
MPYF32 R2H, R0H, R1H      ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++        ; R1H = Y1
MPYF32 R3H, R0H, R1H      ; In parallel R0H = X2
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++        ; R1H = Y2
; R2H = A + X0 * Y0
; R3H = B + X1 * Y1
; R2H = C + X2 * Y2
; R3H = A + B
; R2H = C + X2 * Y2
MACF32 R3H, R2H, R2H, R0H, R1H; In parallel R0H = X3
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++        ; R1H = Y3
; R2H = (A + B) + C
; R3H = (A + B) + C
; R2H = D + X3 * Y3
MACF32 R3H, R2H, R2H, R0H, R1H; In parallel R0H = X4
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5          ; R1H = Y4

; The next MACF32 is an alias for
; MPYF32 || ADDF32
MACF32 R3H, R2H, R2H, R0H, R1H; In parallel R3H = (A + B + C) + D
NOP ; Wait for MPYF32 || ADDF32 to complete
ADDF32 R3H, R3H, R2H      ; R3H = (A + B + C + D) + E
NOP ; Wait for ADDF32 to complete
MOV32 @Result, R3H       ; Store the result

See also

MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32 — 32-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

| R3H | floating-point destination/source register R3H for the add operation |
| R2H | floating-point source register R2H for the add operation |
| RdH | floating-point destination register (R0H to R7H) for the multiply operation |
| ReH | floating-point source register (R0H to R7H) for the multiply operation |
| RfH | floating-point source register (R0H to R7H) for the multiply operation |
| RaH | floating-point destination register for the MOV32 operation (R0H to R7H). |
| mem32 | 32-bit source for the MOV32 operation |

Opcode

LSW: 1110 0011 0011 fffe
MSW: eedd daaa mem32

Description

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF32.

\[
\begin{align*}
R3H &= R3H + R2H, \\
RdH &= ReH \times RfH, \\
RaH &= \text{[mem32]}
\end{align*}
\]

Restrictions

The destination registers for the MACF32 and the MOV32 must be unique. That is, RaH cannot be R3H and RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MACF32 (add or multiply) generates an underflow condition.
- LVF = 1 if MACF32 (add or multiply) generates an overflow condition.

MOV32 sets the NF, ZF, NI and ZI flags as follows:

\[
\begin{align*}
NF &= RaH(31); \\
ZF &= 0; \\
\text{if}(RaH(30:23) == 0) \{ ZF = 1; NF = 0; \} \\
NI &= RaH(31); \\
ZI &= 0; \\
\text{if}(RaH(31:0) == 0) ZI = 1;
\end{align*}
\]

Pipeline

The MACF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

\[
\begin{align*}
\text{MACF32} \ R3H, \ R2H, \ RdH, \ ReH, \ RfH \ ; \ 2 \text{ pipeline cycles (2p)} \\
\text{MOV32} \ RaH, \ \text{mem32} \ ; \ 1 \text{ cycle} \\
\text{NOP} \ ; \ 1 \text{ cycle delay for MACF32} \\
\text{NOP} \ ; \ \text{<-- MACF32 completes, R3H, RdH updated} \\
\text{NOP} \ ; \ \text{<-- MOV32 completes, RaH updated}
\end{align*}
\]

Any instruction in the delay slot for this version of MACF32 must not use R3H or RdH as a destination register or R3H or RdH as a source operand.

Floating Point Unit (FPU)

Copyright © 2014–2019, Texas Instruments Incorporated

Submit Documentation Feedback
Example

; Perform 5 multiply and accumulate operations:
; 1ST multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3
;
; Result = A + B + C + D + E

MOV32 R0H, *XAR4++; ; R0H = X0
MOV32 R1H, *XAR5++; ; R1H = Y0

MPYF32 R2H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++; ; R1H = Y1

MPYF32 R3H, R0H, R1H ; In parallel R0H = X2
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++; ; R1H = Y2

MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++; ; R1H = Y3

MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X4
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5++; ; R1H = Y4

MPYF32 R2H, R0H, R1H ; in parallel R3H = (A + B + C) + D
|| ADDF32 R3H, R3H, R2H
NOP ; Wait for MPYF32 || ADDF32 to complete

ADD F32 R3H, R3H, R2H
NOP ; Wait for ADDF32 to complete
MOV32 @Result, R3H ; Store the result

See also
MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MACF32 R7H, R3H, mem32, *XAR7++ — 32-bit Floating-Point Multiply and Accumulate

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7H</td>
<td>floating-point destination register</td>
</tr>
<tr>
<td>R3H</td>
<td>floating-point destination register</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit source location</td>
</tr>
<tr>
<td>*XAR7++</td>
<td>32-bit location pointed to by auxiliary register 7, XAR7 is post incremented.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0101 0000
MSW: 0001 1111 mem32

Description

Perform a multiply and accumulate operation. When used as a standalone operation, the MACF32 will perform a single multiply as shown below:

Cycle 1: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]

This instruction is the only floating-point instruction that can be repeated using the single repeat instruction (RPT ||). When repeated, the destination of the accumulate will alternate between R3H and R7H on each cycle and R2H and R6H are used as temporary storage for each multiply.

Cycle 1: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
Cycle 2: R7H = R7H + R6H, R6H = [mem32] * [XAR7++]
Cycle 3: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
Cycle 4: R7H = R7H + R6H, R6H = [mem32] * [XAR7++]
etc...

Restrictions

R2H and R6H will be used as temporary storage by this instruction.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MACF32 generates an underflow condition.
- LVF = 1 if MACF32 generates an overflow condition.

Pipeline

When repeated the MACF32 takes 3 + N cycles where N is the number of times the instruction is repeated. When repeated, this instruction has the following pipeline restrictions:

```
<instruction1>
<instruction2>
RPT #/(N-1)
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
<instruction3>
```

- No restriction
- Cannot be a 2p instruction that writes to R2H, R3H, R6H or R7H
- Execute N times, where N is even
- No restrictions.
- Can read R2H, R3H, R6H and R7H
MACF32 can also be used standalone. In this case, the instruction takes 2 cycles and the following pipeline restrictions apply:

\[
\begin{align*}
\text{<instruction1>} & \quad \text{; No restriction} \\
\text{<instruction2>} & \quad \text{; Cannot be a 2p instruction that writes} \\
& \quad \text{; to R2H, R3H, R6H or R7H} \\
\text{MACF32 R7H, R3H, *XAR6, *XAR7} & \quad \text{; R3H = R3H + R2H, R2H = \{mem32\} \times \{XAR7++\}} \\
& \quad ; <-- \\
\text{R2H and R3H are valid (note: no delay required)} \\
\text{NOP} &
\end{align*}
\]

**Example**

\[
\begin{align*}
\text{ZERO R2H} & \quad \text{; Zero the accumulation registers} \\
\text{ZERO R3H} & \quad \text{; and temporary multiply storage} \\
\text{registers} & \\
\text{ZERO R6H} & \\
\text{ZERO R7H} & \\
\text{RPT #3} & \quad \text{; Repeat MACF32 N+1 (4) times} \\
\text{|| MACF32 R7H, R3H, *XAR6++, *XAR7++} & \\
\text{ADDF32 R7H, R7H, R3H} & \quad \text{; Final accumulate} \\
\text{NOP} & \quad <-- \text{ADDF32 completes, R7H valid} \\
\text{NOP} &
\end{align*}
\]

Cascading of \text{RPT || MACF32} is allowed as long as the first and subsequent counts are even. Cascading is useful for creating interruptible windows so that interrupts are not delayed too long by the \text{RPT} instruction. For example:

\[
\begin{align*}
\text{ZERO R2H} & \quad \text{; Zero the accumulation registers} \\
\text{ZERO R3H} & \quad \text{; and temporary multiply storage} \\
\text{registers} & \\
\text{ZERO R6H} & \\
\text{ZERO R7H} & \\
\text{RPT #3} & \quad \text{; Execute MACF32 N+1 (4) times} \\
\text{|| MACF32 R7H, R3H, *XAR6++, *XAR7++ \text{ RPT #5} ; Execute MACF32 N+1 (6) times} & \\
\text{|| MACF32 R7H, R3H, *XAR6++, *XAR7++ \text{ RPT #N} ; Repeat MACF32 N+1 times where N+1} & \text{ is even} \\
\text{|| MACF32 R7H, R3H, *XAR6++, *XAR7++} & \\
\text{ADDF32 R7H, R7H, R3H} & \quad \text{; Final accumulate} \\
\text{NOP} & \quad <-- \text{ADDF32 completes, R7H valid} \\
\end{align*}
\]

**See also**

\[
\begin{align*}
\text{MACF32 R3H, R2H, RdH, ReH, RiH || MOV32 RaH, mem32} \\
\text{MACF32 R7H, R6H, RdH, ReH, RiH || MOV32 RaH, mem32} \\
\text{MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RiH}
\end{align*}
\]
MACF32 R7H, R6H, RdH, ReH, RfH  
32-bit Floating-Point Multiply with Parallel Add

Operands
This instruction is an alias for the parallel multiply and add instruction. The operands are translated by the assembler such that the instruction becomes:

\[
\text{MPYF32 RdH, RaH, RbH} \ || \ \text{ADDF32 R7H, R7H, R6H}
\]

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7H</td>
<td>floating-point destination and source register for the ADDF32</td>
</tr>
<tr>
<td>R6H</td>
<td>floating-point source register for the ADDF32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>RdH</td>
<td>floating-point destination register for MPYF32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>ReH</td>
<td>floating-point source register for MPYF32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register for MPYF32 operation (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0111 0100 00ff
MSW: feee dddc ccb baaa

Description
This instruction is an alias for the parallel multiply and add, MACF32 || ADDF32, instruction.

\[
\text{RdH} = \text{RaH} * \text{RbH} \\
\text{R7H} = \text{R6H} + \text{R6H}
\]

Restrictions
The destination register for the MPYF32 and the ADDF32 must be unique. That is, RdH cannot be R7H.

Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline
Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

\[
\text{MPYF32 RaH, RbH, RcH} \ ; \ 2 \text{ pipeline cycles (2p)} \\
\text{|| ADDF32 RdH, ReH, RfH} \ ; \ 2 \text{ pipeline cycles (2p)} \\
\text{NOP} \ ; \ 1 \text{ cycle delay or non-conflicting instruction} \\
\text{NOP} \ ; \ <-- \text{MPYF32, ADDF32 complete, RaH, RdH updated}
\]

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.
Example

; Perform 5 multiply and accumulate operations:

; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3

; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0
MPYF32 R6H, R0H, R1H ; In parallel R0H = X1

|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1
MPYF32 R7H, R0H, R1H ; In parallel R0H = X2

|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2
MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3

|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3
MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4

|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

; Next MACF32 is an alias for
; ADDF32
MACF32 R7H, R6H, R6H, R0H, R1H ; R6H = E = X4 * Y4
; in parallel R7H = (A + B + C) + D
NOP ; Wait for MPYF32 || ADDF32 to complete
ADDF32 R7H, R7H, R6H ; R7H = (A + B + C + D) + E
NOP ; Wait for ADDF32 to complete
MOV32 @Result, R7H ; Store the result

See also

MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH | MOV32 RaH, mem32 — 32-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7H</td>
<td>floating-point destination/source register R7H for the add operation</td>
</tr>
<tr>
<td>R6H</td>
<td>floating-point source register R6H for the add operation</td>
</tr>
<tr>
<td>RdH</td>
<td>floating-point destination register (R0H to R7H) for the multiply operation. RdH cannot be the same register as RaH.</td>
</tr>
<tr>
<td>ReH</td>
<td>floating-point source register (R0H to R7H) for the multiply operation</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register (R0H to R7H) for the multiply operation</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point destination register for the MOV32 operation (R0H to R7H). RaH cannot be R3H or the same as RdH.</td>
</tr>
<tr>
<td>mem32</td>
<td>32-bit source for the MOV32 operation</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1100 fffe
MSW: eedd daaa mem32

Description
Multiply/accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF32.

R7H = R7H + R6H
RdH = ReH * RfH,
RaH = [mem32]

Restrictions
The destination registers for the MACF32 and the MOV32 must be unique. That is, RaH cannot be R7H and RaH cannot be the same register as RdH.

Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MACF32 (add or multiply) generates an underflow condition.
- LVF = 1 if MACF32 (add or multiply) generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) {ZF = 1;
NF = 0;} NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;

Pipeline
The MACF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

MACF32 R7H, R6H, RdH, ReH, RfH ; 2 pipeline cycles (2p)
MOV32 RaH, mem32
NOP ; 1 cycle
NOP ; 1 cycle delay

---

Floating Point Unit (FPU)

72

SPRUHS1C—October 2014—Revised November 2019

Copyright © 2014–2019, Texas Instruments Incorporated

Submit Documentation Feedback
Example

Perform 5 multiply and accumulate operations:

; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3

; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0

MPYF32 R6H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1

MPYF32 R7H, R0H, R1H ; In parallel R0H = X2
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2

MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3

MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

MPYF32 R6H, R0H, R1H ; in parallel R7H = (A + B + C) + D
|| ADDF32 R7H, R7H, R6H NOP ; Wait for MPYF32 || ADDF32 to complete

ADDF32 R7H, R7H, R6H ; R7H = (A + B + C + D) + E
NOP ; Wait for ADDF32 to complete
MOV32 @Result, R7H ; Store the result

See also

MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
**MAXF32 RaH, RbH — 32-bit Floating-Point Maximum**

**32-bit Floating-Point Maximum**

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

| LSW: 1110 0110 1001 0110 | MSW: 0000 0000 00bb baaa |

**Description**

if(RaH < RbH) RaH = RbH

Special cases for the output from the MAXF32 operation:
- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

if(RaH == RbH) {ZF=1, NF=0}
if(RaH > RbH) {ZF=0, NF=0}
if(RaH < RbH) {ZF=0, NF=1}

**Pipeline**

This is a single-cycle instruction.

**Example**

MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
MOVIZF32 R2H, #-1.5 ; R2H = -1.5 (0xBFC00000)
MAXF32 R2H, R1H ; R2H = -1.5, ZF = NF = 0
MAXF32 R1H, R2H ; R1H = -1.5, ZF = 0, NF = 1
MAXF32 R2H, R0H ; R2H = 5.0, ZF = 0, NF = 1
MAXF32 R0H, R2H ; R2H = 5.0, ZF = 1, NF = 0

**See also**

CMPF32 RaH, RbH
CMPF32 RaH, #16FHi
CMPF32 RaH, #0.0
MAXF32 RaH, RbH || MOV32 RcH, RdH
MAXF32 RaH, #16FHi
MINF32 RaH, RbH
MINF32 RaH, #16FHi
MAXF32 RaH, #16FHi — 32-bit Floating-Point Maximum

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

| LSW: | 1110 1000 0010 IIII |
| MSW: | IIII IIII IIII Iaaa |

Description

Compare RaH with the floating-point value represented by the immediate operand. If the immediate value is larger, then load it into RaH.

\[
\text{if}(\text{RaH} < \text{#16FHi}:0) \text{ RaH} = \text{#16FHi}:0
\]

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

\[
\text{if}(\text{RaH} == \text{#16FHi}:0) \{\text{ZF}=1, \text{NF}=0\}
\]

\[
\text{if}(\text{RaH} > \text{#16FHi}:0) \{\text{ZF}=0, \text{NF}=0\}
\]

\[
\text{if}(\text{RaH} < \text{#16FHi}:0) \{\text{ZF}=0, \text{NF}=1\}
\]

Pipeline

This is a single-cycle instruction.

Example

MOVIZF32 ROH, #5.0 ; ROH = 5.0 (0x40A00000)

MOVIZF32 RH1, #4.0 ; RH1 = 4.0 (0x40800000)

MOVIZF32 RH2, #-1.5 ; RH2 = -1.5 (0xBFC00000)

MAXF32 ROH, #5.5 ; ROH = 5.5, ZF = 0, NF = 1

MAXF32 RH1, #2.5 ; RH1 = 4.0, ZF = 0, NF = 0

MAXF32 RH2, #-1.0 ; RH2 = -1.0, ZF = 0, NF = 1

MAXF32 RH2, #-1.0 ; RH2 = -1.5, ZF = 1, NF = 0

See also

MAXF32 RaH, RbH

MAXF32 RaH, RbH || MOV32 RcH, RdH

MINF32 RaH, RbH

MINF32 RaH, #16FHi
**MAXF32 RaH, RbH  || MOV32 RcH, RdH — 32-bit Floating-Point Maximum with Parallel Move**

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register for the MAXF32 operation (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register for the MAXF32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point destination register for the MOV32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>RdH</td>
<td>floating-point source register for the MOV32 operation (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

- LSW: 1110 0110 1001 1100
- MSW: 0000 dddc ccb baaa

**Description**

If RaH is less than RbH, then load RaH with RbH. Thus RaH will always have the maximum value. If RaH is less than RbH, then, in parallel, also load RcH with the contents of RdH.

```c
if(RaH < RbH) { RaH = RbH; RcH = RdH; }
```

The **MAXF32** instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

### Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

**Restrictions**

The destination register for the MAXF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RcH.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```c
if(RaH == RbH){ZF=1, NF=0}
if(RaH > RbH) {ZF=0, NF=0}
if(RaH < RbH) {ZF=0, NF=1}
```

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
MOV1ZF32 R0H, #5.0  ; R0H = 5.0 (0x40A00000)
MOV1ZF32 R1H, #4.0  ; R1H = 4.0 (0x40800000)
MOV1ZF32 R2H, #-1.5 ; R2H = -1.5 (0xBFCD0000)
MOV1ZF32 R3H, #-2.0 ; R3H = -2.0 (0xC0000000)
MAXF32 R0H, R1H    ; R0H = 5.0, R3H = -1.5, ZF = 0, NF = 0
|| MOV32 R3H, R2H
MAXF32 R1H, R0H    ; R1H = 5.0, R3H = -1.5, ZF = 0, NF = 1
|| MOV32 R3H, R2H
MAXF32 R0H, R1H    ; R0H = 5.0, R2H = -1.5, ZF = 1, NF = 0
|| MOV32 R2H, R1H
```

**See also**

- MAXF32 RaH, RbH
- MAXF32 RaH, #16FHi
MINF32 RaH, RbH — 32-bit Floating-Point Minimum

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW:</th>
<th>1110 0110 1001 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW:</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

Description

if(RaH > RbH) RaH = RbH

Special cases for the output from the MINF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

if(RaH == RbH){ZF=1, NF=0}
if(RaH > RbH) {ZF=0, NF=0}
if(RaH < RbH) {ZF=0, NF=1}

Pipeline

This is a single-cycle instruction.

Example

```
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #4.0 ; R1H = 4.0 (0x40800000)
MOVIZF32 R2H, #-1.5 ; R2H = -1.5 (0xBFC00000)
MINF32 R0H, R1H ; R0H = 4.0, ZF = 0, NF = 0
MINF32 R1H, R2H ; R1H = -1.5, ZF = 0, NF = 0
MINF32 R2H, R1H ; R2H = -1.5, ZF = 1, NF = 0
MINF32 R1H, R0H ; R2H = -1.5, ZF = 0, NF = 1
```

See also

- MAXF32 RaH, RbH
- MAXF32 RaH, #16FHi
- MINF32 RaH, #16FHi
- MINF32 RaH, RbH || MOV32 RcH, RdH
MINF32 RaH, #16FHi — 32-bit Floating-Point Minimum

MINF32 RaH, #16FHi  32-bit Floating-Point Minimum

Operands

RaH  floating-point source/destination register (R0H to R7H)
#16FHi  A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

LSW: 1110 1000 0011 IIII
MSW: IIII IIII IIII Iaaa

Description

Compare RaH with the floating-point value represented by the immediate operand. If the immediate value is smaller, then load it into RaH.

if(RaH > #16FHi:0) RaH = #16FHi:0

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as #-.15 or #0xBFC0.

Special cases for the output from the MINF32 operation:
- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>Ni</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

if(RaH == #16FHi:0) {ZF=1, NF=0}
if(RaH > #16FHi:0) {ZF=0, NF=0}
if(RaH < #16FHi:0) {ZF=0, NF=1}

Pipeline

This is a single-cycle instruction.

Example

MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #4.0 ; R1H = 4.0 (0x40800000)
MOVIZF32 R2H, #-1.5 ; R2H = -1.5 (0xBFC00000)
MINF32 R0H, #5.5 ; R0H = 5.5, ZF = 0, NF = 1
MINF32 R1H, #2.5 ; R1H = 2.5, ZF = 0, NF = 0
MINF32 R2H, #-1.0 ; R2H = -1.5, ZF = 0, NF = 1
MINF32 R2H, #-1.5 ; R2H = -1.5, ZF = 1, NF = 0

See also

MAXF32 RaH, #16FHi
MAXF32 RaH, RbH
MINF32 RaH, RbH
MINF32 RaH, RbH || MOV32 RcH, RdH
MINF32 RaH, RbH | MOV32 RcH, RdH — 32-bit Floating-Point Minimum with Parallel Move

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register for the MIN32 operation (R0H to R7H)</th>
<th>RaH cannot be the same register as RcH</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register for the MIN32 operation (R0H to R7H)</td>
<td></td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point destination register for the MOV32 operation (R0H to R7H)</td>
<td>RcH cannot be the same register as RaH</td>
</tr>
<tr>
<td>RdH</td>
<td>floating-point source register for the MOV32 operation (R0H to R7H)</td>
<td></td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 1101
MSW: 0000 dddc ccbb baaa

Description

if(RaH > RbH) { RaH = RbH; RcH = RdH; }

Special cases for the output from the MINF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Restrictions

The destination register for the MINF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RcH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

- if(RaH == RbH) { ZF=1, NF=0; }
- if(RaH > RbH) { ZF=0, NF=0; }
- if(RaH < RbH) { ZF=0, NF=1; }

Pipeline

This is a single-cycle instruction.

Example

```
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #4.0 ; R1H = 4.0 (0x40800000)
MOVIZF32 R2H, #-1.5 ; R2H = -1.5 (0xBF000000)
MOVIZF32 R3H, #-2.0 ; R3H = -2.0 (0xC0000000)
MINF32 R0H, R1H ; R0H = 4.0, R3H = -1.5, ZF = 0, NF = 0
|| MOV32 R3H, R2H
MINF32 R1H, R0H ; R1H = 4.0, R3H = -1.5, ZF = 1, NF = 0
|| MOV32 R3H, R2H
MINF32 R2H, R1H ; R2H = -1.5, R1H = 4.0, ZF = 1, NF = 1
|| MOV32 R1H, R3H
```

See also

- MINF32 RaH, RbH
- MINF32 RaH, #16FHi
MOV16 mem16, RaH — Move 16-bit Floating-Point Register Contents to Memory

**Operands**

<table>
<thead>
<tr>
<th>mem16</th>
<th>points to the 16-bit destination memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW: 1110 0010 0001 0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0aaa mem16</td>
</tr>
</tbody>
</table>

**Description**

Move 16-bit value from the lower 16-bits of the floating-point register (RaH[15:0]) to the location pointed to by mem16.

\[ \text{[mem16]} = \text{RaH}[15:0] \]

**Flags**

No flags STF flags are affected.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.

**Example**

```
MOVW DP, #0x02C0 ; DP = 0x02C0
MOVXI R4H, #0x0003 ; R4H = 3.0 (0x0003)
MOV16 @0, R4H ; [0x00B000] = 3.0 (0x0003)
```

**See also**

- MOVIZ RaH, #16FHiHex
- MOVIZF32 RaH, #16FHi
- MOVXI RaH, #16FLoHex
MOV32 *(0:16bitAddr), loc32 — Move the Contents of loc32 to Memory

Operands

<table>
<thead>
<tr>
<th>0:16bitAddr</th>
<th>16-bit immediate address, zero extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc32</td>
<td>32-bit source location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1101 loc32  
MSW: IIII IIII IIII IIII

Description

Move the 32-bit value in loc32 to the memory location addressed by 0:16bitAddr. The EALLOW bit in the ST1 register is ignored by this operation.

\[ [0:16bitAddr] = [loc32] \]

Flags

This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a two-cycle instruction.

Example

MOVIZ R5H, #0x1234 ; R5H[31:16] = 0x1234
MOVXI R5H, #0xABCD ; R5H[15:0] = 0xABCD
NOP ; 1 Alignment Cycle
MOV32 ACC, R5H ; ACC = 0x1234ABCD
MOV32 *(0xA000), @ACC ; [0x00A000] = ACC
NOP ; 1 Cycle delay for MOV32 to complete
; <-- MOV32 *(0:16bitAddr), loc32 complete,
; [0x00A000] = 0xABCD, [0x00A001] = 0x1234

See also

MOV32 mem32, RaH
MOV32 mem32, STF
MOV32 loc32, *(0:16bitAddr)
MOV32 ACC, RaH — Move 32-bit Floating-Point Register Contents to ACC

Operands

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>28x accumulator</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1111 loc32
MSW: IIII IIII IIII IIII

Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

ACC = RaH

Flags

No STF flags are affected.

Z and N flag in status register zero (ST0) of the 28x CPU are affected.

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H ; Single-cycle instruction
NOP ; 1 alignment cycle
MOV32 @ACC,R0H ; Copy R0H to ACC
NOP ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP ; 1 cycle delay for ADDF32 to complete
; <-- ADDF32 completes, R2H is valid
NOP ; 1 alignment cycle MOV32 ACC, R2H
; copy R2H into ACC, takes 2 cycles
; <-- MOV32 completes, ACC is valid
NOP ; Any instruction
```

Example

```
ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP ; 1 cycle delay for ADDF32 to complete
; <-- ADDF32 completes, R2H is valid
NOP ; 1 alignment cycle
MOV32 ACC, R2H ; copy R2H into ACC, takes 2 cycles
; <-- MOV32 completes, ACC is valid
NOP ; Any instruction
MOV12F32 ROH, #2.5 ; ROH = 2.5 = 0x40200000
F32TOUI32 ROH, ROH ; Delay for conversion instruction
; <-- Conversion complete, ROH valid
NOP ; Alignment cycle
MOV32 P, ROH ; P = 2 = 0x00000002
```

See also

MOV32 P, RaH
MOV32 XARn, RaH
MOV32 XT, RaH
MOV32 loc32, *(0:16bitAddr) — Move 32-bit Value from Memory to loc32

Operands

<table>
<thead>
<tr>
<th>loc32</th>
<th>destination location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:16bitAddr</td>
<td>16-bit address of the 32-bit source value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1111 loc32
MSW: IIII IIII IIII IIII

Description

Copy the 32-bit value referenced by 0:16bitAddr to the location indicated by loc32.

[loc32] = [0:16bitAddr]

Flags

No STF flags are affected. If loc32 is the ACC register, then the Z and N flag in status register zero (ST0) of the 28x CPU are affected.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 cycle instruction.

Example

MOVW DP, #0x0300 ; DP = 0x0300
MOV @0, #0xFFFF ; [0x00C000] = 0xFFFF;
MOV @1, #0x1111 ; [0x00C001] = 0x1111;
MOV32 @ACC, *(0xC000) ; AL = [0x00C000], AH = [0x00C001]
NOP ; 1 Cycle delay for MOV32 to complete
    ; <-- MOV32 complete, AL = 0xFFFF, AH = 0x1111

See also

MOV32 RaH, mem32{, CNDF}
MOV32 *(0:16bitAddr), loc32
MOV32 STF, mem32
MOVD32 RaH, mem32
**MOV32 mem32, RaH — Move 32-bit Floating-Point Register Contents to Memory**

### Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>points to the 32-bit destination memory</td>
</tr>
</tbody>
</table>

### Opcode

| LSW: 1110 0010 0000 0011 | MSW: 0000 0aaa mem32 |

### Description

Move from memory to STF.

\[[\text{mem32}] = \text{RaH}\]

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

### Pipeline

This is a single-cycle instruction.

### Example

```plaintext
; Perform 5 multiply and accumulate operations:
; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3
;
; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0

| || MPYF32 R6H, R0H, R1H ; In parallel R0H = X1
| || MOV32 R0H, *XAR4++
| || MOV32 R1H, *XAR5++ ; R1H = Y1

| || MPYF32 R7H, R0H, R1H ; In parallel R0H = X2
| || MOV32 R0H, *XAR4++
| || MOV32 R1H, *XAR5++ ; R1H = Y2

| || R7H = A + B
| || R6H = C + X2 * Y2

| || MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3
| || MOV32 R0H, *XAR4++
| || MOV32 R1H, *XAR5++ ; R1H = Y3

| || R3H = (A + B) + C
| || R6H = D + X3 * Y3

| || MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4
| || MOV32 R0H, *XAR4
| || MOV32 R1H, *XAR5 ; R1H = Y4

| || R6H = E * X4
| || in parallel R7H = (A + B + C) + D

| || MPYF32 R6H, R0H, R1H
| || ADDF32 R7H, R7H, R2H
| || NOP ; Wait for MPYF32 || ADDF32 to complete
```
MOV32 mem32, RaH — Move 32-bit Floating-Point Register Contents to Memory

ADDF32 R7H, R7H, R6H ; R7H = (A + B + C + D) + E NOP

MOV32 @Result, R7H ; Wait for ADDF32 to complete
; Store the result

See also

MOV32 *(0:16bitAddr), loc32
MOV32 mem32, STF
**MOV32 mem32, STF**  
*Move 32-bit STF Register to Memory*

### Operands

<table>
<thead>
<tr>
<th>STF</th>
<th>floating-point status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>points to the 32-bit destination memory</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 mem32</td>
</tr>
</tbody>
</table>

### Description

Copy the floating-point status register, STF, to memory.

\[ [\text{mem32}] = \text{STF} \]

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

### Pipeline

This is a single-cycle instruction.

### Example 1

MOVW DP, #0x0280 ; DP = 0x0280  
MOVI2F32 R0H, #2.0 ; R0H = 2.0 (0x40000000)  
MOVI2F32 R1H, #3.0 ; R1H = 3.0 (0x40400000)  
CMPPF32 R0H, R1H ; ZF = 0, NF = 1, STF = 0x00000004  
MOV32 00, STF ; [0x00A000] = 0x00000004

### Example 2

MOV32 *SP++, STF ; Store STF in stack  
MOVF32 R2H, #3.0 ; R2H = 3.0 (0x40400000)  
MOVF32 R3H, #5.0 ; R3H = 5.0 (0x40A00000)  
CMPPF32 R2H, R3H ; ZF = 0, NF = 1, STF = 0x00000004  
MOV32 R3H, R2H, LT ; R3H = 3.0 (0x40400000)  
MOV32 STF, *--SP ; Restore STF from stack

### See also

MOV32 mem32, RaH  
MOV32 *(0:16bitAddr), loc32  
MOVST0 FLAG
MOV32 P, RaH

Move 32-bit Floating-Point Register Contents to P

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>28x product register P</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1111 loc32
MSW: IIII IIII IIII IIII

Description

Move the 32-bit value in RaH to the 28x product register P.

P = RaH

Flags

No flags affected in floating-point unit.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

MINF32 R0H, R1H ; Single-cycle instruction
NOP ; 1 alignment cycle
MOV32 @ACC, R0H ; Copy R0H to ACC
NOP ; Any instruction

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP ; 1 cycle delay for ADDF32 to complete
NOP ; 1 alignment cycle
MOV32 ACC, R2H ; copy R2H into ACC, takes 1 cycle
; <= MOV32 completes, ACC is valid
NOP ; Any instruction

Example

MOV3F32 R0H, #2.5 ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP ; Delay for conversion instruction
NOP ; Alignment cycle
MOV32 P, R0H ; P = 2 = 0x00000002

See also

MOV32 ACC, RaH
MOV32 XARn, RaH
MOV32 XT, RaH
**MOV32 RaH, ACC**  —  *Move the Contents of ACC to a 32-bit Floating-Point Register*

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>accumulator</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1011 1101 loc32  
MSW: IIII IIII IIII IIII

**Description**

Move the 32-bit value in ACC to the floating-point register RaH.

RaH = ACC

**Flags**

This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

MOV32 R0H,@ACC ; Copy ACC to R0H  
NOP ; Wait 4 cycles  
NOP ; Do not use FRACF32, UI16TOF32  
NOP ; I16TOF32, F32TOUI32 or F32TOI32  
NOP ;  
; <-- R0H is valid

**Example**

MOV AH, #0x0000  
MOV AL, #0x0200 ; ACC = 512  
MOV32 R0H, ACC  
NOP  
NOP  
NOP  
NOP UI32TOF32 R0H, R0H ; R0H = 512.0 (0x44000000)

**See also**

MOV32 RaH, P  
MOV32 RaH, XARn  
MOV32 RaH, XT
MOV32 RaH, mem32 {, CNDF}  Conditional 32-bit Move

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>pointer to the 32-bit source memory location</td>
</tr>
<tr>
<td>CNDF</td>
<td>optional condition.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1010 CNDF  
MSW: 0000 0aaa mem32

Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

\[
\text{if (CNDF == TRUE) RaH} = \text{[mem32]}
\]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.

(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{if(CNDF == UNCF)} \\
\{ \\
\text{NF} = \text{RaH[31]; ZF} = 0; \\
\text{if(RaH[30:23] == 0) } \{ \text{ZF} = 1; \text{NF} = 0; \} \text{NI} = \text{RaH[31]; ZI} = 0; \\
\text{if(RaH[31:0] == 0) ZI} = 1; \\
\}
\]

else No flags modified;

Pipeline

This is a single-cycle instruction.
Example

MOVW DP, #0x0300 ; DP = 0x0300
MOV @0, #0x5555 ; [0x00C000] = 0x5555
MOV @1, #0x5555 ; [0x00C001] = 0x5555
MOVI2F32 R3H, #7.0 ; R3H = 7.0 (0x40E00000)
MOVI2F32 R4H, #7.0 ; R4H = 7.0 (0x40E00000)
MAXF32 R3H, R4H ; ZF = 1, NF = 0
MOV32 R1H, @0, EQ ; R1H = 0x55555555

See also

MOV32 RaH, RbH{, CNDF}
MOVD32 RaH, mem32
**MOV32 RaH, P — Move the Contents of P to a 32-bit Floating-Point Register**

### Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>product register</td>
</tr>
</tbody>
</table>

### Opcode

LSW: 1011 1101 loc32  
MSW: IIII IIII IIII IIII

### Description

Move the 32-bit value in the product register, P, to the floating-point register RaH.

RaH = P

### Flags

This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

```
MOV32 R0H,@P ; Copy P to R0H
NOP ; Wait 4 alignment cycles
NOP ; Do not use FRACF32, UI16TOF32
NOP ; I16TOF32, F32TOUI32 or F32TOI32
NOP ;
    ; <-- R0H is valid
    ; Instruction can use R0H as a source
```

### Example

```
MOV  PH, #0x0000
MOV  PL, #0x0200 ; P = 512
MOV32 R0H, P
NOP
NOP
NOP

UI32TOF32 R0H, R0H ; R0H = 512.0 (0x44000000)
```

### See also

- MOV32 RaH, ACC
- MOV32 RaH, XARn
- MOV32 RaH, XT
MOV32 RaH, RbH \{, CNDF\} — Conditional 32-bit Move

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>CNDF</td>
<td>optional condition.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1100 CNDF
MSW: 0000 0000 00bb baaa

Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

\[
\text{if} \ (\text{CNDF} == \text{TRUE}) \ RaH = RbH
\]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.

(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\[
\text{if} \ (\text{CNDF} == \text{UNC}) \ 
(\text{NF} = \text{RaH} (31); \ ZF = 0; \\
\text{if} \ (\text{RaH}[30:23] == 0) \ (ZF = 1; \ NF = 0;) \ \text{NI} = \text{RaH}(31); \ ZI = 0; \\
\text{if} \ (\text{RaH}[31:0] == 0) \ ZI = 1; \text{ else No flags modified;}
\]

Pipeline

This is a single-cycle instruction.

Example

MOVIZF32 R3H, #8.0 ; R3H = 8.0 (0x41000000)
MOVIZF32 R4H, #7.0 ; R4H = 7.0 (0x40E00000)
MAXF32 R3H, R4H ; ZF = 0, NF = 0
MOV32 R1H, R3H, GT ; R1H = 8.0 (0x41000000)

See also
MOV32 RaH, mem32\{, CNDF\}
MOV32 RaH, XARn — Move the Contents of XARn to a 32-bit Floating-Point Register

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XARn</td>
<td>auxiliary register (XAR0 - XAR7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1101 loc32
MSW: IIII IIImm IIImm IIImm

Description

Move the 32-bit value in the auxiliary register XARn to the floating point register RaH.

RaH = XARn

Flags

This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

MOV32 R0H,@XAR7 ; Copy XAR7 to R0H
NOP ; Wait 4 alignment cycles
NOP ; Do not use FRACF32, UI16TOF32
NOP ; I16TOF32, F32TOUI32 or F32TOI32
NOP ;
; <-- R0H is valid
ADDF32 R2H,R1H ,R0H ; Instruction can use R0H as a source

Example

MOVL XAR1, #0x0200 ; XAR1 = 512
MOV32 R0H, XAR1
NOP
NOP
NOP
NOP
UI32TOF32 ROH, ROH ; ROH = 512.0 (0x44000000)

See also

MOV32 RaH, ACC
MOV32 RaH, P
MOV32 RaH, XT
MOV32 RaH, XT — Move the Contents of XT to a 32-bit Floating-Point Register

| Operands | RaH | floating-point register (R0H to R7H) |
| XT | auxiliary register (XAR0 - XAR7) |

| Opcode | LSW: 1011 1101 loc32 |
| MSW: IIII IIII IIII IIII |

| Description | Move the Contents of XT to a 32-bit Floating-Point Register |
| RaH = XT |

| Flags | This instruction does not modify any STF register flags. |

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

| Pipeline | While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32. |
| MOV32 R0H, XT ; Copy XT to R0H |
| NOP ; Wait 4 alignment cycles |
| NOP ; Do not use FRACF32, UI16TOF32 |
| NOP ; I16TOF32, F32TOUI32 or F32TOI32 |
| NOP ; |
| ADDF32 R2H,R1H,R0H ; Instruction can use R0H as a source |

| Example | MOVIZF32 R6H, #5.0 ; R6H = 5.0 (0x40A00000) |
| NAP ; 1 Alignment cycle |
| MOV32 XT, R6H ; XT = 5.0 (0x40A00000) |
| MOV32 R1H, XT ; R1H = 5.0 (0x40A00000) |

| See also | MOV32 RaH, ACC |
| MOV32 RaH, P |
| MOV32 RaH, XARn |
MOV32 STF, mem32 — Move 32-bit Value from Memory to the STF Register

Operands

<table>
<thead>
<tr>
<th>STF</th>
<th>floating-point unit status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>pointer to the 32-bit source memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 0000
MSW: 0000 0000 mem32

Description

Move from memory to the floating-point unit's status register STF.

\[ \text{STF} = [\text{mem32}] \]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Restoring status register will overwrite all flags.

Pipeline

This is a single-cycle instruction.

Example 1

MOVW DP, #0x0300 ; DP = 0x0300
MOV @2, #0x020C ; [0x00C002] = 0x020C
MOV @3, #0x0000 ; [0x00C003] = 0x0000
MOV32 STF, @2 ; STF = 0x0000020C

Example 2

MOV32 *SP++, STF ; Store STF in stack
MOVF32 R2H, #3.0 ; R2H = 3.0 (0x40400000)
MOVF32 R3H, #5.0 ; R3H = 5.0 (0x40A00000)
CMFF32 R2H, R3H ; SF = 0, NF = 1, STF = 0x00000004
MOV32 R3H, R2H, LT ; R3H = 3.0 (0x40400000)
MOV32 STF, --SP ; Restore STF from stack

See also

MOV32 mem32, STF
MOVST0 FLAG
MOV32 XARn, RaH — Move 32-bit Floating-Point Register Contents to XARn

Operands

<table>
<thead>
<tr>
<th>XARn</th>
<th>28x auxiliary register (XAR0 - XAR7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1011 1111 loc32</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>IIII IIII IIII IIII</td>
</tr>
</tbody>
</table>

Description

Move the 32-bit value from the floating-point register RaH to the auxiliary register XARn.

XARn = RaH

Flags

No flags affected in floating-point unit.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H    ; Single-cycle instruction
NOP               ; 1 alignment cycle
MOV32 @ACC,R0H    ; Copy R0H to ACC
NOP               ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H    ; 2 pipeline instruction (2p)
NOP                      ; 1 cycle delay for ADDF32 to complete
                       ; <-- ADDF32 completes, R2H is valid
NOP                      ; 1 alignment cycle
MOV32 ACC, R2H           ; copy R2H into ACC, takes 1 cycle
                       ; <-- MOV32 completes, ACC is valid
NOP                      ; Any instruction
```

Example

```
MOVIZF32 R0H, #2.5  ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H  ; Delay for conversion instruction
                       ; <-- Conversion complete, R0H valid
NOP                     ; Alignment cycle
MOV32 XAR0, R0H       ; XAR0 = 2 = 0x00000002
```

See also

MOV32 ACC, RaH
MOV32 P, RaH
MOV32 XT, RaH
MOV32 XT, RaH — Move 32-bit Floating-Point Register Contents to XT

**Operands**

<table>
<thead>
<tr>
<th>XT</th>
<th>temporary register</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1011 1111 loc32
MSW: IIII IIII IIII IIII

**Description**

Move the 32-bit value in RaH to the temporary register XT.

XT = RaH

**Flags**

No flags affected in floating-point unit.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H  ; Single-cycle instruction
NOP             ; 1 alignment cycle
MOV32 @XT,R0H   ; Copy R0H to ACC NOP
                 ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H  ; 2 pipeline instruction (2p)
NOP             ; 1 cycle delay for ADDF32 to complete
NOP             ; 1 alignment cycle
MOV32 XT, R2H   ; copy R2H into ACC, takes 1 cycle
                 ; <-- MOV32 completes, ACC is valid
NOP             ; Any instruction
```

**Example**

```
MOVI2F32 R0H, #2.5  ; R0H = 2.5 = 0x40200000
F32TOU32 R0H, R0H
NOP             ; Delay for conversion instruction
                 ; <-- Conversion complete, R0H valid
NOP             ; Alignment cycle
MOV32 XT, R0H   ; XT = 2 = 0x00000002
```

**See also**

- MOV32 ACC, RaH
- MOV32 P, RaH
- MOV32 XARn, RaH
MOVD32 RaH, mem32 — Move 32-bit Value from Memory with Data Copy

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>pointer to the 32-bit source memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0010 0011
MSW: 0000 0aaa mem32

Description

Move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

RaH = [mem32] [mem32+2] = [mem32]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

NF = RaH[31];
ZF = 0;
if(RaH[30:23] == 0) { ZF = 1; NF = 0; }
NI = RaH[31];
ZI = 0;
if(RaH[31:0] == 0) ZI = 1;

Pipeline

This is a single-cycle instruction.

Example

MOVW DP, #0x02C0 ; DP = 0x02C0
MOV @2, #0x0000 ; [0x00B002] = 0x0000
MOV @3, #0x4110 ; [0x00B003] = 0x4110
MOVD32 R7H, @2 ; R7H = 0x41100000,
; [0x00B004] = 0x0000, [0x00B005] = 0x4110

See also

MOV32 RaH, mem32 {,CNDF}
MOVF32 RaH, #32F  Load the 32-bits of a 32-bit Floating-Point Register

Operands
This instruction is an alias for MOVIZ and MOVXI instructions. The second operand is translated by the assembler such that the instruction becomes:

- MOVIZ RaH, #16FHiHex
- MOVXI RaH, #16FLoHex

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#32F</td>
<td>immediate float value represented in floating-point representation</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 1000 0000 0III (opcode of MOVIZ RaH, #16FHiHex)
- MSW: IIII IIII IIII Iaaa

- LSW: 1110 1000 0000 1III (opcode of MOVXI RaH, #16FLoHex)
- MSW: IIII IIII IIII Iaaa

Description

Note: This instruction accepts the immediate operand only in floating-point representation. To specify the immediate value as a hex value (IEEE 32-bit floating-point format) use the MOVI32 RaH, #32FHex instruction.

Load the 32-bits of RaH with the immediate float value represented by #32F.

- #32F is a float value represented in floating-point representation. The assembler will only accept a float value represented in floating-point representation. That is, 3.0 can only be represented as #3.0. #0x40400000 will result in an error.

- RaH = #32F

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

Depending on #32FH, this instruction takes one or two cycles. If all of the lower 16-bits of the IEEE 32-bit floating-point format of #32F are zeros, then the assembler will convert MOVF32 into only MOVIZ instruction. If the lower 16-bits of the IEEE 32-bit floating-point format of #32F are not zeros, then the assembler will convert MOVF32 into MOVIZ and MOVXI instructions.

Example

- MOVF32 R1H, #3.0 ; R1H = 3.0 (0x40400000)
  ; Assembler converts this instruction as
  ; MOVIZ R1H, #0x4040
- MOVF32 R2H, #0.0 ; R2H = 0.0 (0x00000000)
  ; Assembler converts this instruction as
  ; MOVIZ R2H, #0x0
- MOVF32 R3H, #12.265 ; R3H = 12.625 (0x41443D71)
  ; Assembler converts this instruction as
  ; MOVIZ R3H, #0x4144
  ; MOVXI R3H, #0x3D71

See also

- MOVIZ RaH, #16FHiHex
- MOVXI RaH, #16FLoHex
- MOVI32 RaH, #32FHex
- MOVIZF32 RaH, #16FHi
MOVI32 RaH, #32FHex — Load the 32-bits of a 32-bit Floating-Point Register with the immediate

MOVI32 RaH, #32FHex  Load the 32-bits of a 32-bit Floating-Point Register with the immediate

Operands
This instruction is an alias for MOVIZ and MOVXI instructions. The second operand is translated by the assembler such that the instruction becomes:

\[
\text{MOVIZ RaH, #16FHiHex} \\
\text{MOVXI RaH, #16FLoHex}
\]

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#32FHex</td>
<td>A 32-bit immediate value that represents an IEEE 32-bit floating-point value.</td>
</tr>
</tbody>
</table>

Opcode

| LSW: 1110 1000 0000 0III (opcode of MOVIZ RaH, #16FHiHex) | MSW: IIII IIII IIII Iaaa |
| LS\_W: 1110 1000 0000 1III (opcode of MOVXI RaH, #16FLoHex) | MS\_W: IIII IIII IIII Iaaa |

Description
Note: This instruction only accepts a hex value as the immediate operand. To specify the immediate value with a floating-point representation use the MOVF32 RaH, #32F instruction.

Load the 32-bits of RaH with the immediate 32-bit hex value represented by #32Fhex. #32Fhex is a 32-bit immediate hex value that represents the IEEE 32-bit floating-point value of a floating-point number. The assembler will only accept a hex immediate value. That is, 3.0 can only be represented as #0x40400000. #3.0 will result in an error.

RaH = #32FHex

Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline
Depending on #32Fhex, this instruction takes one or two cycles. If all of the lower 16-bits of #32Fhex are zeros, then assembler will convert MOVI32 to the MOVIZ instruction. If the lower 16-bits of #32Fhex are not zeros, then assembler will convert MOVI32 to a MOVIZ and a MOVXI instruction.

Example

\[
\begin{align*}
\text{MOVI32 R1H, #0x40400000} & ; \text{R1H} = 0x40400000 \\
& ; \text{Assembler converts this instruction as} \\
& ; \text{MOVIZ R1H, #0x4040} \\
\text{MOVI32 R2H, #0x00000000} & ; \text{R2H} = 0x00000000 \\
& ; \text{Assembler converts this instruction as} \\
& ; \text{MOVIZ R2H, #0x0} \\
\text{MOVI32 R3H, #0x40004001} & ; \text{R3H} = 0x40004001 \\
& ; \text{Assembler converts this instruction as} \\
& ; \text{MOVIZ R3H, #0x4000} \; \text{MOVXI R3H, #0x4001} \\
\text{MOVI32 R4H, #0x00000404} & ; \text{R4H} = 0x00000404 \\
& ; \text{Assembler converts this instruction as} \\
& ; \text{MOVIZ R4H, #0x0000} \; \text{MOVXI R4H, #0x4040}
\end{align*}
\]

See also

MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex
MOVF32 RaH, #32F
MOVIZF32 RaH, #16FHi
MOVIZ RaH, #16FHiHex — Load the Upper 16-bits of a 32-bit Floating-Point Register

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHiHex</td>
<td>A 16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 0000 0III
MSW: IIII IIII IIII Iaaa

Description

Note: This instruction only accepts a hex value as the immediate operand. To specify the immediate value with a floating-point representation use the MOVIZF32 pseudo instruction.

Load the upper 16-bits of RaH with the immediate value #16FHiHex and clear the low 16-bits of RaH.

#16FHiHex is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. The assembler will only accept a hex immediate value. That is, -1.5 can only be represented as #0xBFC0. #1.5 will result in an error.

By itself, MOVIZ is useful for loading a floating-point register with a constant in which the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). If a constant requires all 32-bits of a floating-point register to be initialized, then use MOVIZ along with the MOVXI instruction.

RaH\[31:16\] = #16FHiHex
RaH\[15:0\] = 0

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.

Example

; Load R0H with -1.5 (0xBFC00000)
MOVIZ R0H, #0xBFC0 ; R0H = 0xBFC00000

; Load R0H with pi = 3.141593 (0x40490FDB)
MOVIZ R0H, #0x4049 ; R0H = 0x40490000
MOVXI R0H, #0xFDB ; R0H = 0x40490FDB

See also

MOVIZF32 RaH, #16FHi
MOVXI RaH, #16FLoHex
MOVIZF32 RaH, #16FHi — Load the Upper 16-bits of a 32-bit Floating-Point Register

MOVIZF32 RaH, #16FHi  
Load the Upper 16-bits of a 32-bit Floating-Point Register

Operands

| RaH        | floating-point register (R0H to R7H) |
| #16FHi     | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

- LSW: 1110 1000 0000 0III
- MSW: IIII IIII IIII Iaaa

Description

Load the upper 16-bits of RaH with the value represented by #16FHi and clear the low 16-bits of RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). #16FHi can be specified in hex or float. That is, -1.5 can be represented as #1.5 or #0x3FFC0.

MOVIZF32 is an alias for the MOVIZ RaH, #16FHiHex instruction. In the case of MOVIZF32 the assembler will accept either a hex or float as the immediate value and encodes it into a MOVIZ instruction. For example, MOVIZF32 RaH, #1.5 will be encoded as MOVIZ RaH, 0xBFC0.

RaH[31:16] = #16FHi
RaH[15:0] = 0

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.

Example

```c
MOVIZF32 R0H, #3.0 ; R0H = 3.0 = 0x40400000
MOVIZF32 R1H, #1.0 ; R1H = 1.0 = 0x3F800000
MOVIZF32 R2H, #2.5 ; R2H = 2.5 = 0x40200000
MOVIZF32 R3H, #-5.5 ; R3H = -5.5 = 0xC0B00000
MOVIZF32 R4H, #0xC0B0 ; R4H = -5.5 = 0xC0B00000

; Load R5H with pi = 3.141593 (0x40490000)
; MOVIZF32 R5H, #3.141593 ; R5H = 3.140625 (0x40490000)
;
; Load R0H with a more accurate pi = 3.141593 (0x40490FDB)
; MOVIZF32 R0H, #0x4049 ; R0H = 0x40490000
MOVXI R0H, #0x0FDB ; R0H = 0x40490FDB
```

See also

- MOVIZ RaH, #16FHiHex
- MOVXI RaH, #16FloHex
MOVST0 FLAG — Load Selected STF Flags into ST0

Operands

<table>
<thead>
<tr>
<th>FLAG</th>
<th>Selected flag</th>
</tr>
</thead>
</table>

Opcode

LSW: 1010 1101 FFFF FFFF

Description

Load selected flags from the STF register into the ST0 register of the 28x CPU where FLAG is one or more of TF, CI, ZI, ZF, NI, NF, LUF or LVF. The specified flag maps to the ST0 register as follows:

- Set OV = 1 if LVF or LUF is set. Otherwise clear OV.
- Set N = 1 if NF or NI is set. Otherwise clear N.
- Set Z = 1 if ZF or ZI is set. Otherwise clear Z.
- Set C = 1 if TF is set. Otherwise clear C.
- Set TC = 1 if TF is set. Otherwise clear TF.

If any STF flag is not specified, then the corresponding ST0 register bit is not modified.

Restrictions

Do not use the MOVST0 instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the MOVST0 operation.

Example

Program flow is controlled by C28x instructions that read status flags in the status register 0 (ST0). If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into ST0 flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVST0 FLAG instruction is used to load the current value of specified STF flags into the respective bits of ST0. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified.

```assembly
Loop:
  MOV32 R0H,*XAR4++
  MOV32 R1H,*XAR3++
  CMPF32 R1H, R0H
  MOVST0 ZF, NF
  BF Loop, GT ; Loop if (R1H > R0H)

See also

MOV32 mem32, STF
MOV32 STF, mem32
```
MOVXI RaH, #16FLoHex — Move Immediate to the Low 16-bits of a Floating-Point Register

**MOVXI RaH, #16FLoHex**  *Move Immediate to the Low 16-bits of a Floating-Point Register*

**Operands**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>floating-point register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits will not be modified.</td>
</tr>
</tbody>
</table>

**Opcode**

| LSW: 1110 1000 0000 1III | MSW: IIII IIII IIII Iaaa |

**Description**

Load the low 16-bits of RaH with the immediate value #16FLoHex. #16FLoHex represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits of RaH will not be modified. MOVXI can be combined with the MOVIZ or MOVIZF32 instruction to initialize all 32-bits of a RaH register.

RaH[15:0] = #16FLoHex
RaH[31:16] = Unchanged

**Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.

**Example**

```plaintext
; Load R0H with pi = 3.141593 (0x40490FDB)
MOVIZ R0H,#0x4049  ; R0H = 0x40490000
MOVXI R0H,#0x0FDB ; R0H = 0x40490FDB
```

**See also**

- MOVIZ RaH, #16FHiHex
- MOVIZF32 RaH, #16FHi
MPYF32 RaH, RbH, RcH  

32-bit Floating-Point Multiply

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0000 0000  
MSW: 0000 000c ccbb baaa  

Description

Multiply the contents of two floating-point registers.

RaH = RbH * RcH

Flags

This instruction modifies the following flags in the STF register:.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF</td>
<td>No</td>
</tr>
<tr>
<td>ZI</td>
<td>No</td>
</tr>
<tr>
<td>NI</td>
<td>No</td>
</tr>
<tr>
<td>ZF</td>
<td>No</td>
</tr>
<tr>
<td>NF</td>
<td>No</td>
</tr>
<tr>
<td>LUF</td>
<td>Yes</td>
</tr>
<tr>
<td>LVF</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

MPYF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)  
NOP ; 1 cycle delay or non-conflicting instruction  
; <-- MPYF32 completes, RaH updated  
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate Y = A * B:

MOV32 R0H, *XAR4 ; Load R0H with A  
MOV32 R1H, *XAR4 ; Load R1H with B  
MPYF32 R0H, R1H, R0H ; Multiply A * B  
MOV32 R0H, #Y  
; <--MPYF32 complete  
MOV32 *XAR4, R0H ; Save the result

See also

MPYF32 RaH, #16FHi, RbH  
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH  
MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32  
MPYF32 RdH, ReH, RfH || MOV32 mem32, RaH  
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH  
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
**MPYF32 RaH, #16FHi, RbH — 32-bit Floating-Point Multiply**

**Operands**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 1000 01II IIII
MSW: IIII IIII IIbb baaa

**Description**

Multiply RbH with the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #1.5 or #0xBFC0.

RaH = RbH * #16FHi:0

This instruction can also be written as MPYF32 RaH, RbH, #16FHi.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

```
MPYF32 RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
      ; <-- MPYF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

**Example 1**

```
MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)
MPYF32 R4H, #3.0, R3H ; R4H = 3.0 * R3H
MOVL XAR1, #0xB006 ; <-- Non conflicting instruction
      ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB006
```

**Example 2**

```
Same as above example but #16FHi is represented in Hex

MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)
MPYF32 R4H, #0x4040, R3H ; R4H = 0x4040 * R3H
      ; 3.0 is represented as 0x40400000 in
      ; IEEE 754 32-bit format
MOVL XAR1, #0xB006 ; <-- Non conflicting instruction
      ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB006
```
See also

- MPYF32 RaH, RbH, #16FHi
- MPYF32 RaH, RbH, RcH
- MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RlH
**MPYF32 RaH, RbH, #16FHi — 32-bit Floating-Point Multiply**

**Operands**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 1000 01II IIII  
MSW: IIII IIII IIbb baaa

**Description**

Multiply RbH with the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value.

RaH = RbH * #16FHi:0

This instruction can also be written as MPYF32 RaH, #16FHi, RbH.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

```assembly
MPYF32 RaH, RbH, #16FHi ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- MPYF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

**Example 1**

```assembly
MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)  
MPYF32 R4H, R3H, #3.0 ; R4H = R3H * 3.0  
MOVL XAR1, #0x0B08 ; <-- Non conflicting instruction
    ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB008
```

**Example 2**

```assembly
; Same as above example but #16FHi is represented in Hex
MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)  
MPYF32 R4H, R3H, #0x4040 ; R4H = R3H * 0x4040
    ; 3.0 is represented as 0x40400000 in IEEE 754 32-bit format
MOVL XAR1, #0xB008 ; <-- Non conflicting instruction
    ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB008
```
See also

- MPYF32 RaH, #16FHi, RbH
- MPYF32 RaH, RbH, RcH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Add

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point source register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RdH</td>
<td>floating-point destination register for ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>ReH</td>
<td>floating-point source register for ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register for ADDF32 (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0100 00ff
MSW: feee dddc cbbb baaa

Description

Multiply the contents of two floating-point registers with parallel addition of two registers.

RaH = RbH * RcH
RdH = ReH + RfH

This instruction can also be written as:
MACF32 RaH, RbH, RcH, RdH, ReH, RfH

Restrictions

The destination register for the MPYF32 and the ADDF32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline

Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

MPYF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)
|| ADDF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
<= MPYF32, ADDF32 complete, RaH, RdH updated
NOP

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.
Example

; Perform 5 multiply and accumulate operations:

; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3

; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0

; R2H = A = X0 * Y0
MPYF32 R2H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1

; R3H = B = X1 * Y1
MPYF32 R3H, R0H, R1H ; In parallel R0H = X2
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2

; R3H = A + B
; R2H = C = X2 * Y2
MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3

; R3H = (A + B) + C
; R2H = D = X3 * Y3
MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X4
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

; R2H = E = X4 * Y4
MPYF32 R2H, R0H, R1H ; in parallel R3H = (A + B + C) + D
|| ADDF32 R3H, R3H, R2H
NOP ; Wait for MPYF32 || ADDF32 to complete

ADDF32 R3H, R3H, R2H ; R3H = (A + B + C + D) + E
NOP ; Wait for ADDF32 to complete

MOV32 @Result, R3H ; Store the result

See also
MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32 — 32-bit Floating-Point Multiply with Parallel Move

Operands

<table>
<thead>
<tr>
<th>RdH</th>
<th>floating-point destination register for the MPYF32 (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReH</td>
<td>floating-point source register for the MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register for the MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point destination register for the MOV32 (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location. This will be the source of the MOV32.</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0011 0000 fffe</td>
<td>eedd daaa mem32</td>
</tr>
</tbody>
</table>

Description

Multiply the contents of two floating-point registers and load another.

\[
RdH = ReH \times RfH \\
RaH = \text{[mem32]}
\]

Restrictions

The destination register for the MPYF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- **LUF** = 1 if MPYF32 generates an underflow condition.
- **LVF** = 1 if MPYF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

\[
\begin{align*}
NF &= RaH(31) \\
ZF &= 0 \\
\text{if}(RaH(30:23) == 0) \{ ZF = 1; NF = 0; \} \\
NI &= RaH(31) \\
ZI &= 0 \\
\text{if}(RaH(31:0) == 0) \{ ZI = 1; \}
\end{align*}
\]

Pipeline

MPYF32 takes 2 pipeline-cycles (2p) and MOV32 takes a single cycle. That is:

```
MPYF32 RdH, ReH, RfH ; 2 pipeline cycles (2p) 
|| MOV32 RaH, mem32 ; 1 cycle 
    ; <-- MOV32 completes, RaH updated 
    NOP ; 1 cycle delay or non-conflicting instruction 
    ; <-- MPYF32 completes, RdH updated 
    NOP
```

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.
Example

Calculate \( Y = M1 \times X1 + B1 \). This example assumes that \( M1, X1, B1 \) and \( Y \) are all on the same data page.

\[
\begin{align*}
\text{MOVW} \quad \text{DP,} \quad #M1 &; \quad \text{Load the data page} \\
\text{MOV32} \quad \text{ROH,} \quad #M1 &; \quad \text{Load ROH with M1} \\
\text{MOV32} \quad \text{R1H,} \quad #X1 &; \quad \text{Load R1H with X1} \\
\text{MPYF32} \quad \text{R1H,} \quad \text{R1H,} \quad \text{ROH} &; \quad \text{Multiply M1} \times \text{X1} \\
\text{||} \quad \text{MOV32} \quad \text{ROH,} \quad #B1 &; \quad \text{and in parallel load ROH with B1} \\
\text{NOP} &; \quad \text{Wait 1 cycle for MPYF32 to complete} \\
\text{ADDF32} \quad \text{R1H,} \quad \text{R1H,} \quad \text{ROH} &; \quad \text{Add M1} \times \text{X1 to B1 and store in R1H} \\
\text{NOP} &; \quad \text{Wait 1 cycle for ADDF32 to complete} \\
\text{MOV32} \quad @Y1, \quad \text{R1H} &; \quad \text{Store the result}
\end{align*}
\]

Calculate \( Y = (A \times B) \times C \):

\[
\begin{align*}
\text{MOV} &; \quad \text{XAR4,} \quad #A \\
\text{MOV32} \quad \text{ROH,} \quad \text{*XAR4} &; \quad \text{Load ROH with A} \\
\text{MOV} &; \quad \text{XAR4,} \quad #B \\
\text{MOV32} \quad \text{R1H,} \quad \text{*XAR4} &; \quad \text{Load R1H with B} \\
\text{MOV} &; \quad \text{XAR4,} \quad #C \\
\text{MPYF32} \quad \text{R1H,} \quad \text{R1H,} \quad \text{ROH} &; \quad \text{Calculate R1H} = A \times B \\
\text{||} \quad \text{MOV32} \quad \text{ROH,} \quad \text{*XAR4} &; \quad \text{and in parallel load R2H with C} \\
\text{MOV} &; \quad \text{XAR4,} \quad #Y \\
\text{||} \quad \text{MPYF32} \quad \text{R2H,} \quad \text{R1H,} \quad \text{ROH} &; \quad \text{Calculate Y} = (A \times B) \times C \\
\text{NOP} &; \quad \text{Wait 1 cycle for MPYF32 to complete} \\
\text{MOV32} \quad \text{ROH,} \quad \text{R1H,} \quad \text{R2H} &; \quad \text{MPYF32 complete}
\end{align*}
\]

See also

MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, RdH, ReH, RfH, *XAR7++
MPYF32 RdH, ReH, RfH || MOV32 mem32, RaH — 32-bit Floating-Point Multiply with Parallel Move

Operands

- **RdH**
  - Floating-point destination register for the MPYF32 (R0H to R7H)
- **ReH**
  - Floating-point source register for the MPYF32 (R0H to R7H)
- **RfH**
  - Floating-point source register for the MPYF32 (R0H to R7H)
- **mem32**
  - Pointer to a 32-bit memory location. This will be the destination of the MOV32.
- **RaH**
  - Floating-point source register for the MOV32 (R0H to R7H)

Opcode

- **LSW**: 1110 0000 0000 fffe
- **MSW**: eedd daaa mem32

Description

Multiply the contents of two floating-point registers and move from memory to register.

\[
RdH = ReH \times RfH, \quad [\text{mem32}] = RaH
\]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- **LUF** = 1 if MPYF32 generates an underflow condition.
- **LVF** = 1 if MPYF32 generates an overflow condition.

Pipeline

MPYF32 takes 2 pipeline-cycles (2p) and MOV32 takes a single cycle. That is:

\[
\text{MPYF32 RdH, ReH, RfH} ; 2 \text{ pipeline cycles (2p)}
\]
\[
\text{|| MOV32 mem32, RaH} ; 1 \text{ cycle}
\]
\[
\text{|| NOP} ; 1 \text{ cycle delay or non-conflicting instruction}
\]
\[
\text{|| NOP} ; \text{ <-- MOV32 completes, mem32 updated}
\]
\[
\text{|| MOV32 *XAR1, R3H} ; \text{ and in parallel store previous R3 value}
\]
\[
\text{|| NOP} ; 1 \text{ cycle delay for MPYF32 to complete}
\]

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

Example

\[
\text{MOV L XAR1, #0xC003} ; \text{XAR1} = 0xC003
\]
\[
\text{MOVIZF32 R3H, #2.0} ; \text{R3H} = 2.0 (0x40000000)
\]
\[
\text{MPYF32 R3H, R3H, #5.0} ; \text{R3H} = \text{R3H} \times 5.0
\]
\[
\text{MOVIZF32 R1H, #5.0} ; \text{R1H} = 5.0 (0x40A00000)
\]
\[
\text{MPYF32 R3H, R1H, R3H} ; \text{R3H} = \text{R1H} \times \text{R3H}
\]
\[
\text{|| MOV32 *XAR1, R3H} ; \text{and in parallel store previous R3 value}
\]
\[
\text{|| NOP} ; 1 \text{ cycle delay for MPYF32 to complete}
\]

See also

- MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32
- MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
- MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
- MACF32 R7H, R3H, mem32, *XAR7++
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH

32-bit Floating-Point Multiply with Parallel Subtract

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point source register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RdH</td>
<td>floating-point destination register for SUBF32 (R0H to R7H)</td>
</tr>
<tr>
<td>ReH</td>
<td>floating-point source register for SUBF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register for SUBF32 (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0111 0101 00ff</td>
<td>feee dddc ccb baaa</td>
</tr>
</tbody>
</table>

Description

Multiply the contents of two floating-point registers with parallel subtraction of two registers.

RaH = RbH * RcH,
RdH = ReH - RfH

Restrictions

The destination register for the MPYF32 and the SUBF32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or SUBF32 generates an underflow condition.
- LVF = 1 if MPYF32 or SUBF32 generates an overflow condition.

Pipeline

MPYF32 and SUBF32 both take 2 pipeline-cycles (2p). That is:

MPYF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)
|| SUBF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- MPYF32, SUBF32 complete. RaH, RdH updated
NOP

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.

Example

MOV1ZF32 R4H, #5.0 ; R4H = 5.0 (0x40A00000)
MOV1ZF32 R5H, #3.0 ; R5H = 3.0 (0x40400000)
MPYF32 R6H, R4H, R5H ; R6H = R4H * R5H
|| SUBF32 R7H, R4H, R5H ; R7H = R4H - R5H NOP
; 1 cycle delay for MPYF32 || SUBF32 to complete
; <-- MPYF32 || SUBF32 complete,
; R6H = 15.0 (0x41700000), R7H = 2.0 (0x40000000)

See also

SUBF32 RaH, RbH, RcH
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH
NEGF32 RaH, RbH{, CNDF} — Conditional Negation

Operand

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>CNDF</td>
<td>condition tested</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1010 CNDF  
MSW: 0000 0000 00bb baaa

Description

if (CNDF == true) {RaH = - RbH }  
else {RaH = RbH }

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode (1)</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.  
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.

Example

MOVIZF32 R0H, #5.0  ; R0H = 5.0  (0x40A00000)  
MOVIZF32 R1H, #4.0  ; R1H = 4.0  (0x40800000)  
MOVIZF32 R2H, #-1.5  ; R2H = -1.5  (0xBFC00000)  
MPYF32 R4H, R1H, R2H ; R4H = -6.0  
MPYF32 R5H, R0H, R1H ; R5H = 20.0  
; <--- R4H valid  
CMPF32 R4H, #0.0 ; NF = 1  
; <--- R5H valid  
NEGF32 R4H, R4H, LT ; if NF = 1, R4H = 6.0  
NEGF32 R5H, #0.0 ; NF = 0  
NEGF32 R5H, R5H, GEQ ; if NF = 0, R4H = -20.0

See also

ABSF32 RaH, RbH
POP RB — Pop the RB Register from the Stack

**Operands**

RB

repeat block register

**Opcode**

LSW: 1111 1111 1111 0001

**Description**

Restore the RB register from stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

**Flags**

This instruction does not affect any flags floating-point Unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.

**Example**

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```assembly
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
Interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Save RB register only if a RPTB block is used in the ISR
    ...;
RPTB #BlockEnd, AL ; Execute the block AL+1 times
    ...;
BlockEnd ; End of block to be repeated
    ...
    ...
    POP RB ; Restore RB register
    ...
    IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```assembly
; Repeat Block within a Low-Priority Interrupt (Interruptible)
Interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Always save RB register
    ...
CLRC INTM ; Enable interrupts only after saving RB
    ...
    ; ISR may or may not include a RPTB block
    ...
SETC INTM ; Disable interrupts before restoring RB
    ...
POP RB ; Always restore RB register
    ...
IRET ; RA = RAS, RAS = 0
```
See also

- PUSH RB
- RPTB label, #RC
- RPTB label, loc16
PUSH RB  

Push the RB Register onto the Stack

Operands

| RB          | repeat block register |

Opcode

LSW: 1111 1111 1111 0000

Description

Save the RB register on the stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

Flags

This instruction does not affect any flags floating-point Unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction for the first iteration, and zero cycles thereafter.

Example

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
_INTERRUPT:
  ; RAS = RA, RA = 0
  ...
  PUSH RB ; Save RB register only if a RPTB block is used in the ISR
  ...
  RPTB #BlockEnd, AL ; Execute the block AL+1 times
  ...
  ...
  BlockEnd ; End of block to be repeated
  ...
  POP RB ; Restore RB register
  ...
  IRET ; RA = RAS, RAS = 0

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

; Repeat Block within a Low-Priority Interrupt (Interruptible)
_INTERRUPT:
  ; RAS = RA, RA = 0
  ...
  PUSH RB ; Always save RB register
  ...
  CLRC INTM ; Enable interrupts only after saving RB
  ...
  ...
  SETC INTM ; Disable interrupts before restoring RB
  ...
  POP RB ; Always restore RB register
  ...
  IRET ; RA = RAS, RAS = 0

See also

POP RB
RPTB label, #RC
RPTB label, loc16
RESTORE — Restore the Floating-Point Registers

Operands

| none | This instruction does not have any operands |

Opcode

LSW: 1110 0101 0110 0010

Description

Restore the floating-point register set (R0H - R7H and STF) from their shadow registers. The SAVE and RESTORE instructions should be used in high-priority interrupts. That is interrupts that cannot themselves be interrupted. In low-priority interrupt routines the floating-point registers should be pushed onto the stack.

Restrictions

The RESTORE instruction cannot be used in any delay slots for pipelined operations. Doing so will yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the RESTORE operation.

; The following is INVALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
RESTORE ; INVALID, do not use RESTORE in a delay slot

; The following is VALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
NOP ; 1 delay cycle, R2H updated after this instruction
RESTORE ; VALID

Flags

Restoring the status register will overwrite all flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
Example

The following example shows a complete context save and restore for a high-priority interrupt. Note that the CPU automatically stores the following registers: ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1 and PC. If an interrupt is low priority (that is it can be interrupted), then push the floating point registers onto the stack instead of using the SAVE and RESTORE operations.

```assembly
; Interrupt Save
.HighestPriorityISR: ; Uninterruptable
    ASP ; Align stack
    PUSH RB ; Save RB register if used in the ISR
    PUSH AR1H:AR0H ; Save other registers if used
    PUSH XAR2
    PUSH XAR3
    PUSH XAR4
    PUSH XAR5
    PUSH XAR6
    PUSH XAR7
    PUSH XT
    SPM 0 ; Set default C28 modes
    CLRC AMODE
    CLRC PAGE0,OVM
    SAVE RNDF32=1 ; Save all FPU registers
    ... ; set default FPU modes
    ... ; Interrupt Restore
    ... ; Restore all FPU registers
    POP XT ; restore other registers
    POP XAR7
    POP XAR6
    POP XAR5
    POP XAR4
    POP XAR3
    POP XAR2
    POP AR1H:AR0H
    POP RB ; restore RB register
    NASP ; un-align stack
    IRET ; return from interrupt
```

See also

SAVE FLAG, VALUE
**RPTB label, loc16 — Repeat A Block of Code**

**Operands**

<table>
<thead>
<tr>
<th><strong>label</strong></th>
<th>This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loc16</strong></td>
<td>16-bit location for the repeat count value.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1011 0101 0bbb bbbb
MSW: 0000 0000 loc16

**Description**

Initialize repeat block loop, repeat count from [loc16]

**Restrictions**

- The maximum block size is ≤127 16-bit words.
- An even aligned block must be ≥ 9 16-bit words.
- An odd aligned block must be ≥ 8 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch, or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

**Flags**

This instruction does not affect any flags in the floating-point unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This instruction takes four cycles on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

**Example**

The minimum size for the repeat block is 9 words if the block is even-aligned and 8 words if the block is odd-aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even-aligned. Since a NOP is a 16-bit instruction the RPTB will be odd-aligned. For blocks of 9 or more words, this is not required.

; Repeat Block of 8 Words (Interruptible)
;
; find the largest element and put its address in XAR6
.align 2

NOP
RPTB VECTOR_MAX_END, AR7 ; Execute the block AR7+1 times
MOV L ACC,XAR0
MOV32 R1H,*XAR0++ ; min size = 8, 9 words
MAXF32 R0H,R1H ; max size = 127 words
MOVS T0 NF,ZF
MOV L XAR6,ACC,LT
VECTOR_MAX_END: ; label indicates the end
   ; RA is cleared

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.
A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```assembly
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
; Interrupt: ; RAS = RA, RA = 0
... PUSH RB ; Save RB register only if a RPTB block is used in the ISR
... ...
RPTB #BlockEnd, AL ; Execute the block AL+1 times
... ...
... BlockEnd ; End of block to be repeated
... ...
POP RB ; Restore RB register
... IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```assembly
; Repeat Block within a Low-Priority Interrupt (Interruptible)
; Interrupt: ; RAS = RA, RA = 0
... PUSH RB ; Always save RB register
... CLRC INTM ; Enable interrupts only after saving RB
... ...
... ; ISR may or may not include a RPTB block
... ...
SETC INTM ; Disable interrupts before restoring RB
... POP RB ; Always restore RB register
... IRET ; RA = RAS, RAS = 0
```

See also

POP RB
PUSH RB
RPTB label, #RC
RPTB label, #RC — Repeat a Block of Code

**Operands**

| label | This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block. |
| #RC | 16-bit location |

**Opcode**

LSW: 1011 0101 1bbb bbbb  
MSW: cccc cccc cccc cccc

**Description**

Repeat a block of code. The repeat count is specified as a immediate value.

**Restrictions**

- The maximum block size is ≤ 127 16-bit words.
- An even aligned block must be ≥ 9 16-bit words.
- An odd aligned block must be ≥ 8 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

**Flags**

This instruction does not affect any flags in the floating-point unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>Ni</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This instruction takes one cycle on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

**Example**

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

```assembly
; Repeat Block (Interruptible)
;
; find the largest element and put its address in XAR6
.align 2

NOP
RPTB VECTOR_MAX_END, #(4-1) ; Execute the block 4 times
MOVL ACC, XAR0
MOV32 R1H, *XAR0++ ; 8 or 9 words block size 127 words
MAXF32 R0H, R1H
MOVST0 NF, ZF
MOVL XAR6, ACC, LT
VECTOR_MAX_END: ; RE indicates the end address
; RA is cleared
```

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.
A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt:  ; RAS = RA, RA = 0
...
PUSH RB  ; Save RB register only if a RPTB block is used in the ISR...
...
RPTB #BlockEnd, #5  ; Execute the block 5+1 times...
...
...
BlockEnd  ; End of block to be repeated...
...
POP RB  ; Restore RB register...
IRET  ; RA = RAS, RAS = 0

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt:  ; RAS = RA, RA = 0
...
PUSH RB  ; Always save RB register...
CLRC INTM  ; Enable interrupts only after saving RB...
...
...
;
isetc INTM  ; Disable interrupts before restoring RB...
POP RB  ; Always restore RB register...
IRET  ; RA = RAS, RAS = 0

See also

POP RB
PUSH RB
RPTB label, loc16
SAVE FLAG, VALUE — Save Register Set to Shadow Registers and Execute SETFLG

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG</td>
<td>11 bit mask indicating which floating-point status flags to change.</td>
</tr>
<tr>
<td>VALUE</td>
<td>11 bit mask indicating the flag value; 0 or 1.</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW: 1110 0110 01FF FFFF</td>
<td>11 bit mask indicating which floating-point status flags to change.</td>
</tr>
<tr>
<td>MSW: FFFF FVVV VVVV VVVV</td>
<td>11 bit mask indicating the flag value; 0 or 1.</td>
</tr>
</tbody>
</table>

Description

This operation copies the current working floating-point register set (R0H to R7H and STF) to the shadow register set and combines the SETFLG FLAG, VALUE operation in a single cycle. The status register is copied to the shadow register before the flag values are changed. The STF[SHDWM] flag is set to 1 when the SAVE command has been executed. The SAVE and RESTORE instructions should be used in high-priority interrupts. That is interrupts that cannot themselves be interrupted. In low-priority interrupt routines the floating-point registers should be pushed onto the stack.

Restrictions

Do not use the SAVE instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the SAVE operation.

Example

The following example shows a complete context save and restore for a high priority interrupt. Note that the CPU automatically stores the following registers: ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1 and PC.
SAVE FLAG, VALUE — Save Register Set to Shadow Registers and Execute SETFLG

```assembly
_HighestPriorityISR:
  ASP          ; Align stack
  PUSH RB      ; Save RB register if used in the ISR
  PUSH AR1H:AR0H ; Save other registers if used
  PUSH XAR2
  PUSH XAR3
  PUSH XAR4
  PUSH XAR5
  PUSH XAR6
  PUSH XAR7
  PUSH XT
  SPM 0        ; Set default C28 modes
  CLRC AMODE
  CLRC PAGE0,OVM
  SAVE RNDF32=0 ; Save all FPU registers
  ...          ; set default FPU modes
  ...          ...
  ...          ...
  RESTORE      ; Restore all FPU registers
  POP XT       ; restore other registers
  POP XAR7
  POP XAR6
  POP XAR5
  POP XAR4
  POP XAR3
  POP XAR2
  POP AR1H:AR0H
  POP RB       ; restore RB register
  NASP         ; un-align stack IRET
  ; return from interrupt

See also
  RESTORE
  SETFLG FLAG, VALUE
```
**SETFLG FLAG, VALUE** — *Set or clear selected floating-point status flags*

**Operands**

<table>
<thead>
<tr>
<th>OPERAND</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG</td>
<td>11 bit mask indicating which floating-point status flags to change.</td>
</tr>
<tr>
<td>VALUE</td>
<td>11 bit mask indicating the flag value; 0 or 1.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 00FF FFFF  
MSW: FFFF FVVV VVVV VVVV

**Description**

The SETFLG instruction is used to set or clear selected floating-point status flags in the STF register. The FLAG field is an 11-bit value that indicates which flags will be changed. That is, if a FLAG bit is set to 1 it indicates that flag will be changed; all other flags will not be modified. The bit mapping of the FLAG field is shown below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>reserved</td>
</tr>
<tr>
<td>9</td>
<td>reserved</td>
</tr>
<tr>
<td>8</td>
<td>reserved</td>
</tr>
<tr>
<td>7</td>
<td>reserved</td>
</tr>
<tr>
<td>6</td>
<td>TF</td>
</tr>
<tr>
<td>5</td>
<td>ZI</td>
</tr>
<tr>
<td>4</td>
<td>NI</td>
</tr>
<tr>
<td>3</td>
<td>ZF</td>
</tr>
<tr>
<td>2</td>
<td>NF</td>
</tr>
<tr>
<td>1</td>
<td>LUF</td>
</tr>
<tr>
<td>0</td>
<td>LVF</td>
</tr>
</tbody>
</table>

The VALUE field indicates the value the flag should be set to; 0 or 1.

**Restrictions**

Do not use the SETFLG instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the SETFLG operation.

```plaintext
; The following is INVALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
SETFLG RNDF32=1 ; INVALID, do not use SETFLG in a delay slot

; The following is VALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
NOP ; 1 delay cycle, R2H updated after this instruction
SETFLG RNDF32=1 ; VALID
```

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Modified</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZI</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NI</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZF</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NF</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUF</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LVF</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Any flag can be modified by this instruction.

**Pipeline**

This is a single-cycle instruction.

**Example**

To make it easier and legible, the assembler will accept a FLAG=VALUE syntax for the SETFLG operation as shown below:

```plaintext
SETFLG RNDF32=0, TF=1, ZF=0 ; FLAG = 01001001000, VALUE = X0XX1XX0XXX
MOVST0 TF, ZF, LUF ; Copy the indicated flags to ST0
; X means this flag is not modified.
; The assembler will set X values to 0
```

**See also**

SAVE FLAG, VALUE
SUBF32 RaH, RbH, RcH  32-bit Floating-Point Subtraction

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R1)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R1)</td>
</tr>
<tr>
<td>RcH</td>
<td>floating-point source register (R0H to R1)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0010 0000
MSW: 0000 000c cbbb baaa

Description

Subtract the contents of two floating-point registers

RaH = RbH - RcH

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

SUBF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- SUBF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

Calculate Y - A + B - C:

MOVL XAR4, #A
    MOV32 R0H, *XAR4 ; Load R0H with A
    MOVL XAR4, #B
    MOV32 R1H, *XAR4 ; Load R1H with B
    MOVL XAR4, #C
    ADDF32 R0H, R1H, R0H ; Add A + B and in parallel
    || MOV32 R2H, *XAR4 ; Load R2H with C

    ; <-- ADDF32 complete
    SUBF32 R0H, R0H, R2H ; Subtract C from (A + B)
    NOP ; <-- SUBF32 completes
    MOV32 *XAR4, R0H ; Store the result

See also

SUBF32 RaH, #16FHi, RbH
SUBF32 RdH, ReH, RiH || MOV32 RaH, mem32
SUBF32 RdH, ReH, RiH || MOV32 mem32, RaH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RiH
SUBF32 RaH, #16FHi, RbH — 32-bit Floating Point Subtraction

**Operands**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R1)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R1)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 1000 11II IIII
MSW: IIII IIII IIbb baaa

**Description**

Subtract RbH from the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBF000000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBF00.

\[
RaH = #16FHi:0 - RbH
\]

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

\[
\text{SUBF32 RaH, #16FHi, RbH} ; 2 \text{ pipeline cycles (2p)}
\]

NOP ; 1 cycle delay or non-conflicting instruction ; <-- SUBF32 completes, RaH updated

NOP

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

**Example**

Calculate \( Y = 2.0 - (A + B) \):

```
MOVL XAR4, #A
MOV32 R0H, *XAR4 ; Load R0H with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
ADDF32 R0H,R1H,R0H ; Add A + B and in parallel
NOP ; <-- ADDF32 complete
SUBF32 R0H,#2.0,R2H ; Subtract (A + B) from 2.0
NOP ; <-- SUBF32 completes
MOV32 *XAR4,R0H ; Store the result
```

**See also**

- SUBF32 RaH, RbH, RcH
- SUBF32 RdH, ReH, RIH || MOV32 RaH, mem32
- SUBF32 RdH, ReH, RIH || MOV32 mem32, RaH
- MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RIH
SUBF32 RdH, ReH, RfH ||MOV32 RaH, mem32  32-bit Floating-Point Subtraction with Parallel Move

**Operands**

<table>
<thead>
<tr>
<th>RdH</th>
<th>floating-point destination register (R0H to R7H) for the SUBF32 operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReH</td>
<td>floating-point source register (R0H to R7H) for the SUBF32 operation</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register (R0H to R7H) for the SUBF32 operation</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point destination register (R0H to R7H) for the MOV32 operation</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to 32-bit source memory location for the MOV32 operation</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0011 0010 fffe  
MSW: eedd daaa mem32

**Description**

Subtract the contents of two floating-point registers and move from memory to a floating-point register.

RdH = ReH - RfH, RaH = [mem32]

**Restrictions**

The destination register for the SUBF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RdH.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if SUBF32 generates an underflow condition.
- LVF = 1 if SUBF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

NF = RaH(31);  
ZF = 0;  
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }  
NI = RaH(31);  
ZI = 0;  
if(RaH(31:0) == 0) ZI = 1;

**Pipeline**

SUBF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

SUBF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)  
|| MOV32 RaH, mem32 ; 1 cycle  
|<-- MOV32 completes, RaH updated  
NOP ; 1 cycle delay or non-conflicting instruction  
|<-- SUBF32 completes, RdH updated  
NOP

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32 — 32-bit Floating-Point Subtraction with Parallel Move

Example

Example

MOVL XAR1, #0xC000 ; XAR1 = 0xC000
SUBF32 R0H, R1H, R2H ; (A) R0H = R1H - R2H
|| MOV32 R3H, *XAR1 ;
; <-- R3H valid
; <-- (A) completes, R0H valid, R4H valid
ADDF32 R5H, R4H, R3H ; (B) R5H = R4H + R3H
|| MOV32 *+XAR1[4], R0H ;
; <-- R0H stored
MOVL XAR2, #0xE000 ;
; <-- (B) completes, R5H valid
MOV32 *XAR2, R5H ;
; <-- R5H stored

See also

SUBF32 RaH, RbH, RcH
SUBF32 RaH, #16FHi, RbH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH — 32-bit Floating-Point Subtraction with Parallel Move

### Operands

<table>
<thead>
<tr>
<th>RdH</th>
<th>floating-point destination register (R0H to R7H) for the SUBF32 operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReH</td>
<td>floating-point source register (R0H to R7H) for the SUBF32 operation</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register (R0H to R7H) for the SUBF32 operation</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to 32-bit destination memory location for the MOV32 operation</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point source register (R0H to R7H) for the MOV32 operation</td>
</tr>
</tbody>
</table>

### Opcode

- **LSW:** 1110 0000 0010 fffe
- **MSW:** eedd daaa mem32

### Description

Subtract the contents of two floating-point registers and move from a floating-point register to memory.

\[
\text{RdH} = \text{ReH} - \text{RfH}, \\
[\text{mem32}] = \text{RaH}
\]

### Flags

This instruction modifies the following flags in the STF register: SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if SUBF32 generates an underflow condition.
- LVF = 1 if SUBF32 generates an overflow condition.

### Pipeline

SUBF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

\[
\text{SUBF32 RdH, ReH, RfH} ; 2 \text{ pipeline cycles (2p)} \\
|| \text{MOV32 mem32, RaH} ; 1 \text{ cycle} \\
\quad ; \text{<-- MOV32 completes, mem32 updated} \\
\quad \text{NOP} ; 1 \text{ cycle delay or non-conflicting instruction} \\
\quad ; \text{<-- ADDF32 completes, RdH updated} \\
\quad \text{NOP}
\]

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

### Example

\[
\begin{align*}
\text{ADDF32} & \quad \text{R3H, R6H, R4H} ; \quad \text{(A) R3H = R6H + R4H and R7H = I3} \\
|| \text{MOV32} & \quad \text{R7H, */SP[2]} ; \\
\quad & \quad ; \text{<-- R7H valid} \\
\text{SUBF32} & \quad \text{R6H, R6H, R4H} ; \quad \text{(B) R6H = R6H - R4H} \\
& \quad ; \text{<-- ADDF32 (A) completes, R3H valid} \\
\text{SUBF32} & \quad \text{R3H, R1H, R7H} ; \quad \text{(C) R3H = R1H - R7H and store R3H (A)} \\
|| \text{MOV32} & \quad \text{*/XAR5[2], R3H} ; \\
\quad & \quad ; \text{<-- SUBF32 (B) completes, R6H valid} \\
\quad & \quad ; \text{<-- MOV32 completes, (A) stored} \\
\text{ADDF32} & \quad \text{R4H, R7H, R1H} ; \quad \text{R4H = D = R7H + R1H and store R6H (B)} \\
|| \text{MOV32} & \quad \text{*/XAR5[6], R6H} ; \\
\quad & \quad ; \text{<-- SUBF32 (C) completes, R3H valid} \\
\quad & \quad ; \text{<-- MOV32 completes, (B) stored} \\
\text{MOV32} & \quad \text{*/XAR5[0], R3H} ; \quad \text{store R3H (C)} \\
\quad & \quad ; \text{<-- MOV32 completes, (C) stored} \\
\quad & \quad ; \text{<-- ADDF32 (D) completes, R4H valid} \\
\text{MOV32} & \quad \text{*/XAR5[4], R4H} ; \quad \text{store R4H (D)} \\
\quad & \quad ; \text{<-- MOV32 completes, (D) stored}
\end{align*}
\]
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH — 32-bit Floating-Point Subtraction with Parallel Move

See also

SUBF32 RaH, RbH, RcH
SUBF32 RaH, #16FHi, RbH
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
SWAPF RaH, RbH{, CNDF}  

**Conditional Swap**

**Operands**

- **RaH**: floating-point register (R0H to R7H)
- **RbH**: floating-point register (R0H to R7H)
- **CNDF**: condition tested

**Opcode**

LSW: 1110 0110 1110 CNDF
MSW: 0000 0000 00bb baaa

**Description**

Conditional swap of RaH and RbH.

\[
\text{if (CNDF == true) swap RaH and RbH}
\]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>UNCF</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected

**Pipeline**

This is a single-cycle instruction.

**Example**

; find the largest element and put it in R1H

MOV L XAR1, #0xB000 ;
MOV32 R1H, *XAR1  ; Initialize R1H
.align 2
NOP
RPTB LOOP_END, #10-1; Execute the block 10 times
MOV32 R2H, *XAR1++ ; Update R2H with next element
CMPP32 R2H, R1H  ; Compare R2H with R1H
SWAPF R1H, R2H, GT ; Swap R1H and R2H if R2 > R1
NOP             ; For minimum repeat block size
NOP             ; For minimum repeat block size
LOOP_END:
TESTTF CNDF — Test STF Register Flag Condition

Operands

<table>
<thead>
<tr>
<th>CNDF</th>
<th>condition to test</th>
</tr>
</thead>
</table>

Opcode

LSW: 1110 0101 1000 CNDF

Description

Test the floating-point condition and if true, set the TF flag. If the condition is false, clear the TF flag. This is useful for temporarily storing a condition for later use.

\[
\text{if (CNDF == true) TF} = 1; \quad \text{else TF} = 0;
\]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF = 0 AND NF = 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF = 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF = 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF = 1 AND NF = 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF = 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF = 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF = 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF = 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.

(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\[
\text{TF} = 0; \quad \text{if (CNDF == true) TF} = 1;
\]

Note: If (CNDF == UNC or UNCF), the TF flag will be set to 1.

Pipeline

This is a single-cycle instruction.

Example

```
CMPF32 R0H, #0.0 ; Compare R0H against 0
TESTTF LT ; Set TF if R0H less than 0 (NF == 0)
ABS R0H, R0H ; Get the absolute value of R0H

; Perform calculations based on ABS R0H
MOVSST0 TF ; Copy TF to TC in ST0
SBF End, NTC ; Branch to end if TF was not set
NEGF32 R0H, R0H
End
```

See also
**UI16TOF32 RaH, mem16**  
*Convert unsigned 16-bit integer to 32-bit floating-point value*

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>pointer to 16-bit source memory location</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0010 1100 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 0aaa mem16</td>
</tr>
</tbody>
</table>

**Description**

RaH = UI16ToF32[mem16]

**Flags**

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

```c
UI16TOF32 RaH, mem16 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- UI16TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

**Example**

```c
; float32 y,m,b;
; AdcRegs.RESULT0 is an unsigned int
; Calculate: y = (float)AdcRegs.RESULT0 * m + b;
;
MOVW DP @0x01C4
UI16TOF32 R0H, @8 ; R0H = (float)AdcRegs.RESULT0
MOV32 R1H, *-SP[6] ; R1H = M
    ; Conversion complete, R0H valid
MPYF32 R0H, R1H, R0H ; R0H = (float)X * M
MOV32 R1H, *-SP[8] ; R1H = B
    ; MPYF32 complete, R0H valid
ADDF32 R0H, R0H, R1H ; R0H = Y = (float)X * M + B
NOP
    ; ADDF32 complete, R0H valid
MOV32 *-[SP], R0H ; Store Y
```

**See also**

F32TOI16 RaH, RbH  
F32TOI16R RaH, RbH  
F32TOUI16 RaH, RbH  
F32TOUI16R RaH, RbH  
I16TOF32 RaH, RbH  
I16TOF32 RaH, mem16  
UI16TOF32 RaH, RbH
UI16TOF32 RaH, RbH — Convert unsigned 16-bit integer to 32-bit floating-point value

### Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

### Opcode

LSW: 1110 0110 1000 1111  
MSW: 0000 0000 00bb baaa

### Description

RaH = UI16ToF32[RbH]

### Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```assembly
UI16TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- UI16TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

### Example

```assembly
MOVXI R5H, #0x800F ; R5H[15:0] = 32783 (0x800F)
UI16TOF32 R6H, R5H ; R6H = UI16TOF32 (R5H[15:0])
NOP ; 1 cycle delay for UI16TOF32 to complete
    ; R6H = 32783.0 (0x47000F00)
```

### See also

- F32TOI16 RaH, RbH
- F32TOI16R RaH, RbH
- F32TOUI16 RaH, RbH
- F32TOUI16R RaH, RbH
- I16TOF32 RaH, RbH
- I16TOF32 RaH, mem16
- UI16TOF32 RaH, mem16
UI32TOF32 RaH, mem32  —  Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value

Operands

- **RaH**
  - floating-point destination register (R0H to R7H)

- **mem32**
  - pointer to 32-bit source memory location

Opcodes

- LSW: 1110 0010 1000 0100
- MSW: 0000 0aaa mem32

Description

- **RaH = UI32ToF32[mem32]**

Flags

- **This instruction does not affect any flags:**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

- This is a 2 pipeline cycle (2p) instruction. That is:
  - UI32TOF32 RaH, mem32 ; 2 pipeline cycles (2p)
  - NOP ; 1 cycle delay non-conflicting instruction
  - <-- UI32TOF32 completes, RaH updated
  - NOP

- Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

```plaintext
; unsigned long X
; float Y, M, B
; ...
; Calculate Y = (float)X * M + B

UI32TOF32 R0H, *-SP[2] ; R0H = (float)X
MOV32 R1H, *-SP[6] ; R1H = M
; <-- Conversion complete, R0H valid
MPYF32 R0H, R1H, R0H ; R0H = (float)X * M
MOV32 R1H, *-SP[8] ; R1H = B
; <-- MPYF32 complete, R0H valid
ADDF32 R0H, R0H, R1H ; R0H = Y = (float)X * M + B
NOP
; <-- ADDF32 complete, R0H valid
MOV32 *-[SP], R0H ; Store Y
```

See also

- F32TOI32 RaH, RbH
- F32TOUI32 RaH, RbH
- I32TOF32 RaH, mem32
- I32TOF32 RaH, RbH
- UI32TOF32 RaH, RbH
UI32TOF32 RaH, RbH — Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1000 1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

Description

RaH = UI32ToF32[RbH]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

UI32TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
     ; <-- UI32TOF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

MOVIZ R3H, #0x8000 ; R3H[31:16] = 0x8000
MOVXI R3H, #0x1111 ; R3H[15:0] = 0x1111
     ; R3H = 2147488017
UI32TOF32 R4H, R3H ; R4H = UI32TOF32 (R3H)
NOP ; 1 cycle delay for UI32TOF32 to complete
     ; R4H = 2147488017.0 (0x4F000011)

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, mem32
I32TOF32 RaH, RbH
UI32TOF32 RaH, mem32
# ZERO RaH

**Zero the Floating-Point Register RaH**

## Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
</table>

## Opcode

LSW: 1110 0101 1001 0aaa

## Description

Zero the indicated floating-point register:

RaH = 0

## Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

## Pipeline

This is a single-cycle instruction.

## Example

```plaintext
;for(i = 0; i < n; i++)
{
    real += (x[2*i] * y[2*i]) - (x[2*i+1] * y[2*i+1]);
    imag += (x[2*i] * y[2*i+1]) + (x[2*i+1] * y[2*i]);
}

;Assume AR7 = n-1
ZERO R4H ; R4H = real = 0
ZERO R5H ; R5H = imag = 0
LOOP
    MOV AL, AR7
    MOV ACC, AL << 2
    MOV AR0, ACC
    MOV32 R0H, *+XAR4[AR0] ; R0H = x[2*i]
    MOV32 R1H, *+XAR5[AR0] ; R1H = y[2*i]
    ADD AR0, #2
    MPYF32 R6H, R0H, R1H; ; R6H = x[2*i] * y[2*i]
    | MOV32 R2H, *+XAR4[AR0] ; R2H = x[2*i+1]
    | MPYF32 R1H, R1H, R2H ; R1H = y[2*i+1] * x[2*i+2]
    | MOV32 R3H, *+XAR5[AR0] ; R3H = y[2*i+1]
    | MPYF32 R2H, R2H, R3H ; R2H = x[2*i+1] * y[2*i+1]
    | ADDF32 R4H, R4H, R6H ; R4H += x[2*i] * y[2*i]
    | MPYF32 R0H, R0H, R3H ; R0H = x[2*i] * y[2*i+1]
    | ADDF32 R5H, R5H, R1H ; R5H += y[2*i] * x[2*i+2]
    | SUBF32 R4H, R4H, R2H ; R4H -= x[2*i+1] * y[2*i+1]
    | ADDF32 R5H, R5H, R0H ; R5H += x[2*i] * y[2*i+1]
    | BANZ LOOP, AR7--
```

## See also

ZEROA
**ZEROA — Zero All Floating-Point Registers**

**Operands**

| none |

**Opcode**

`LSW: 1110 0101 0110 0011`

**Description**

Zero all floating-point registers:

- `R0H = 0`
- `R1H = 0`
- `R2H = 0`
- `R3H = 0`
- `R4H = 0`
- `R5H = 0`
- `R6H = 0`
- `R7H = 0`

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
//for(i = 0; i < n; i++)
{
    real += (x[2*i] * y[2*i]) - (x[2*i+1] * y[2*i+1]);
    imag += (x[2*i] * y[2*i+1]) + (x[2*i+1] * y[2*i]);
}

;Assume AR7 = n-1
ZEROA ; Clear all RaH registers
LOOP
    MOV AL, AR7
    MOV ACC, AL << 2
    MOV R0, ACC
    MOV32 R0H, *+XAR4[AR0] ; R0H = x[2*i]
    MOV32 R1H, *+XAR5[AR0] ; R1H = y[2*i]
    ADD AR0, #2
    MPYF32 R6H, R0H, R1H; ; R6H = x[2*i] * y[2*i]
    | | MOV32 R2H, *+XAR4[AR0] ; R2H = x[2*i+1]
    | | MPYF32 R1H, R1H, R2H ; R1H = y[2*i+1] * x[2*i+2]
    | | MOV32 R3H, *+XAR5[AR0] ; R3H = y[2*i+1]
    | | MPYF32 R2H, R2H, R3H ; R2H = x[2*i+1] * y[2*i+1]
    | | ADDF32 R4H, R4H, R6H ; R4H += x[2*i] * y[2*i]
    | | MPYF32 R0H, R0H, R3H ; R0H = x[2*i] * y[2*i+1]
    | | ADDF32 R5H, R5H, R1H ; R5H += y[2*i] * x[2*i+2]
    | | SUBF32 R4H, R4H, R2H ; R4H -= x[2*i+1] * y[2*i+1]
    | ADDF32 R5H, R5H, R0H ; R5H += x[2*i] * y[2*i+1]
    BANZ LOOP, AR7--
```

**See also**

ZERO RaH
The TMS320C2000™ DSP family consists of fixed-point and floating-point digital signal processors. TMS320C2000™ Digital Signal Processors combine control peripheral integration and ease of use of a microcontroller (MCU) with the processing power and C efficiency of TI's leading DSP technology. This chapter provides an overview of the architectural structure and components of the C28x plus floating-point unit (FPU64) CPU.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Overview</td>
<td>144</td>
</tr>
<tr>
<td>2.2 Components of the C28x plus Floating-Point CPU (FPU64)</td>
<td>145</td>
</tr>
<tr>
<td>2.3 CPU Register Set</td>
<td>148</td>
</tr>
<tr>
<td>2.4 Pipeline</td>
<td>154</td>
</tr>
<tr>
<td>2.5 Floating Point Unit (FPU64) Instruction Set</td>
<td>162</td>
</tr>
</tbody>
</table>
2.1 Overview

The C28x plus floating-point (C28x+FPU64) processor extends the capabilities of the C28x fixed-point CPU by adding registers and instructions to support IEEE single-precision and double-precision floating point operations. This device draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The DSP features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and modified Harvard architecture (usable in Von Neumann mode). The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation. The modified Harvard architecture of the CPU enables instruction and data fetches to be performed in parallel. The CPU can read instructions and data while it writes data simultaneously to maintain the single-cycle instruction operation across the pipeline. The CPU does this over six separate address/data buses.

Throughout this document the following notations are used:

- C28x refers to the C28x fixed-point CPU.
- C28x plus Floating-Point and C28x+FPU both refer to the C28x CPU with enhancements to support IEEE single-precision floating-point operations.
- C28x+FPU64 refer to the C28x CPU with enhancements to support IEEE single-precision and double-precision floating-point operations. FPU64 extensions supports all existing FPU single precision floating point instructions.

2.1.1 Compatibility with the C28x Fixed-Point CPU

No changes have been made to the C28x base set of instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x CPU and C28x CPU + FPU are completely compatible with the C28x CPU + FPU64 and all of the features of the C28x documented in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430) apply to the C28x CPU + FPU64.

Figure 2-1 shows basic functions of the FPU64.

2.1.1.1 Floating-Point Code Development

When developing C28x floating-point code for C28x+FPU64, use Code Composer Studio 8.0, or later. For C28x+FPU64 (double precision), the TI C28x C/C++ Compiler v18.9.0.STS or later is required to generate C28x native floating-point opcodes. To build floating-point code use the compiler switches:-v28 and --float_support=fpu64.

NOTE: In Code Composer Studio 8.0 the float_support option is in the build options under compiler->advanced: floating point support. Without the float_support flag, or with float_support=none, the compiler will generate fixed-point code. These compilers are available via Code Composer Studio update advisor or as a separate download. When building for C28x, using CCS project properties General entry, “Runtime support library <automatic>”, will automatically select the correct RTS library during link. This is just linker option --libc.a. If any are not yet built then the linker will automatically build the necessary RTS library.
2.2 Components of the C28x plus Floating-Point CPU (FPU64)

The C28x+FPU64 contains:

- A central processing unit for generating data and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory.
- A floating-point unit (FPU64) for IEEE single-precision or double-precision floating point operations.
- Emulation logic for monitoring and controlling various parts and functions of the device and for testing device operation. This logic is identical to that on the C28x fixed-point CPU.
- Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts. This logic is identical to the C28x fixed-point CPU.

Some features of the C28x+FPU64 central processing unit are:

- Fixed-Point instructions are pipeline protected. This pipeline for fixed-point instructions is identical to that on the C28x fixed-point CPU. The CPU implements an 8-phase pipeline that prevents a write to and a read from the same location from occurring out of order. See Figure 2-5.
- Some floating-point instructions require pipeline alignment. This alignment is done through software to allow the user to improve performance by taking advantage of required delay slots.
- Independent register space. These registers function as system-control registers, math registers, and data pointers. The system-control registers are accessed by special instructions.
- Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- Floating point unit (FPU64). The 64-bit FPU performs IEEE single-precision and IEEE double-precision floating point operations.
floating-point operations.

- Address register arithmetic unit (ARAU). The ARAU generates data memory addresses and increments or decrements pointers in parallel with ALU operations.
- Barrel shifter. This shifter performs all left and right shifts of fixed-point data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.
- Fixed-Point Multiplier. The multiplier performs 32-bit × 32-bit 2s-complement multiplication with a 64-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.

### 2.2.1 Emulation Logic

The emulation logic is identical to that on the C28x fixed-point CPU. This logic includes the following features:

- Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline.
- A counter for performance benchmarking.
- Multiple debug events. Any of the following debug events can cause a break in program execution:
  - A breakpoint initiated by the ESTOP0 or ESTOP1 instruction.
  - An access to a specified program-space or data-space location.
  - When a debug event causes the C28x to enter the debug-halt state, the event is called a break event.
- Real-time mode of operation.

For more details about these features, refer to the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430).

### 2.2.2 Memory Map

Like the C28x, the C28x+FPU64 uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x+FPU64 designs are uniformly mapped to both program and data space. For specific details about each of the map segments, see the data sheet for your device.

### 2.2.3 On-Chip Program and Data

All C28x+FPU64 based devices contain at least two blocks of single access on-chip memory referred to as M0 and M1. Each of these blocks is 1K words in size. M0 is mapped at addresses 0x0000 – 0x03FF and M1 is mapped at addresses 0x0400 – 0x07FF. Like all other memory blocks on the C28x+FPU64 devices, M0 and M1 are mapped to both program and data space. Therefore, you can use M0 and M1 to execute code or for data variables. At reset, the stack pointer is set to the top of block M1. Depending on the device, it may also have additional random-access memory (RAM), read-only memory (ROM), external interface zones, or flash memory.

### 2.2.4 CPU Interrupt Vectors

The C28x+FPU64 interrupt vectors are identical to those on the C28x CPU. Sixty-four addresses in program space are set aside for a table of 32 CPU interrupt vectors. The CPU vectors can be mapped to the top or bottom of program space by way of the VMAP bit. For more information about the CPU vectors, see *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430). For devices with a peripheral interrupt expansion (PIE) block, the interrupt vectors will reside in the PIE vector table and this memory can be used as program memory.
2.2.5 Memory Interface

The C28x+FPU64 memory interface is identical to that on the C28x. The C28x+FPU64 memory map is accessible outside the CPU by the memory interface, which connects the CPU logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed. The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the C28x+FPU64 supports special byte-access instructions that can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.

2.2.5.1 Address and Data Buses

Like the C28x, the memory interface has three address buses:

- **PAB: Program address bus**
  The PAB carries addresses for reads and writes from program space. PAB is a 22-bit bus.

- **DRAB: Data-read address bus**
  The 32-bit DRAB carries addresses for reads from data space.

- **DWAB: Data-write address bus**
  The 32-bit DWAB carries addresses for writes to data space.

The memory interface also has three data buses:

- **PRDB: Program-read data bus**
  The PRDB carries instructions during reads from program space. PRDB is a 32-bit bus.

- **DRDB: Data-read data bus**
  The DRDB carries data during reads from data space. DRDB is a 32-bit bus.

- **DWDB: Data-/Program-write data bus**
  The 32-bit DWDB carries data during writes to data space or program space.

A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time. This behavior is identical to the C28x CPU.

2.2.5.2 Alignment of 32-Bit Accesses to Even Addresses

The C28x+FPU64 CPU expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU will begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the CPU.

You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space.
2.3 CPU Register Set

The C28x+FPU64 architecture is the same as the C28x CPU with an extended register and instruction set to support IEEE single-precision and double-precision floating point operations. This section describes the extensions to the C28x architecture.

2.3.1 CPU Registers

Devices with the C28x+FPU64 include the standard C28x register set plus an additional set of floating-point unit registers. The additional floating-point unit registers are the following:

- Eight floating-point result registers, RnH (where n = 0 - 7) for single-precision floating point operations
- Eight floating-point result registers, RnH:RnL (where n = 0 - 7) for double-precision floating point operations
- Floating-point Status Register (STF)
- Repeat Block Register (RB)

All of the floating-point registers except the repeat block register are shadowed. This shadowing can be used in high priority interrupts for fast context save and restore of the floating-point registers.

Figure 2-2 shows a diagram of both register sets and Table 2-1 shows a register summary. For information on the standard C28x register set, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430).

![Figure 2-2. C28x With FPU64 Floating-Point Registers](image-url)
### Table 2-1. 28x Plus Floating-Point FPU64 CPU Register Summary

<table>
<thead>
<tr>
<th>Register</th>
<th>C28x CPU</th>
<th>C28x + FPU64</th>
<th>Size</th>
<th>Description</th>
<th>Value After Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Accumulator</td>
<td>0x00000000</td>
</tr>
<tr>
<td>AH</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>High half of ACC</td>
<td>0x0000</td>
</tr>
<tr>
<td>AL</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of ACC</td>
<td>0x0000</td>
</tr>
<tr>
<td>XAR0</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 0</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR1</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR2</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 2</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR3</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 3</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR4</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 4</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR5</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 5</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR6</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 6</td>
<td>0x00000000</td>
</tr>
<tr>
<td>XAR7</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Auxiliary register 7</td>
<td>0x00000000</td>
</tr>
<tr>
<td>AR0</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR0</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR1</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR1</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR2</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR2</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR3</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR3</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR4</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR4</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR5</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR5</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR6</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR6</td>
<td>0x0000</td>
</tr>
<tr>
<td>AR7</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XAR7</td>
<td>0x0000</td>
</tr>
<tr>
<td>DP</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Data-page pointer</td>
<td>0x0000</td>
</tr>
<tr>
<td>IFR</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Interrupt flag register</td>
<td>0x0000</td>
</tr>
<tr>
<td>IER</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Interrupt enable register</td>
<td>0x0000</td>
</tr>
<tr>
<td>DBGIER</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Debug interrupt enable register</td>
<td>0x0000</td>
</tr>
<tr>
<td>P</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Product register</td>
<td>0x00000000</td>
</tr>
<tr>
<td>PH</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>High half of P</td>
<td>0x0000</td>
</tr>
<tr>
<td>PL</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of P</td>
<td>0x0000</td>
</tr>
<tr>
<td>PC</td>
<td>Yes</td>
<td>Yes</td>
<td>22 bits</td>
<td>Program counter</td>
<td>0x3FFFFC0</td>
</tr>
<tr>
<td>RPC</td>
<td>Yes</td>
<td>Yes</td>
<td>22 bits</td>
<td>Return program counter</td>
<td>0x00000000</td>
</tr>
<tr>
<td>SP</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Stack pointer</td>
<td>0x0400</td>
</tr>
<tr>
<td>ST0</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Status register 0</td>
<td>0x0000</td>
</tr>
<tr>
<td>ST1</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Status register 1</td>
<td>0x080B (1)</td>
</tr>
<tr>
<td>XT</td>
<td>Yes</td>
<td>Yes</td>
<td>32 bits</td>
<td>Multiplicand register</td>
<td>0x00000000</td>
</tr>
<tr>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>High half of XT</td>
<td>0x0000</td>
</tr>
<tr>
<td>TL</td>
<td>Yes</td>
<td>Yes</td>
<td>16 bits</td>
<td>Low half of XT</td>
<td>0x0000</td>
</tr>
<tr>
<td>ROH</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 0</td>
<td>0.0</td>
</tr>
<tr>
<td>R1H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 1</td>
<td>0.0</td>
</tr>
<tr>
<td>R2H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 2</td>
<td>0.0</td>
</tr>
<tr>
<td>R3H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 3</td>
<td>0.0</td>
</tr>
<tr>
<td>R4H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 4</td>
<td>0.0</td>
</tr>
<tr>
<td>R5H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 5</td>
<td>0.0</td>
</tr>
<tr>
<td>R6H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 6</td>
<td>0.0</td>
</tr>
</tbody>
</table>

(1) Reset value shown is for devices without the VMAP signal and MOM1MAP signal pinned out. On these devices both of these signals are tied high internal to the device.
### Table 2-1. 28x Plus Floating-Point FPU64 CPU Register Summary (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>C28x CPU</th>
<th>C28x + FPU64</th>
<th>Size</th>
<th>Description</th>
<th>Value After Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7H</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point single / double precision result register 7</td>
<td>0.0</td>
</tr>
<tr>
<td>R0L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 0</td>
<td>0.0</td>
</tr>
<tr>
<td>R1L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 1</td>
<td>0.0</td>
</tr>
<tr>
<td>R2L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 2</td>
<td>0.0</td>
</tr>
<tr>
<td>R3L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 3</td>
<td>0.0</td>
</tr>
<tr>
<td>R4L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 4</td>
<td>0.0</td>
</tr>
<tr>
<td>R5L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 5</td>
<td>0.0</td>
</tr>
<tr>
<td>R6L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 6</td>
<td>0.0</td>
</tr>
<tr>
<td>R7L</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>32 Bits Floating point double precision result register 7</td>
<td>0.0</td>
</tr>
<tr>
<td>STF</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Floating-point status register</td>
<td>0x00000000</td>
</tr>
<tr>
<td>RB</td>
<td>No</td>
<td>Yes</td>
<td>32 bits</td>
<td>Repeat block register</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

#### 2.3.1.1 Floating-Point Status Register (STF)

The floating-point status register (STF) reflects the results of floating-point operations. There are three basic rules for floating point operation flags:

1. Zero and negative flags are set based on moves to registers.
2. Zero and negative flags are set based on the result of compare, minimum, maximum, negative and absolute value operations.
3. Overflow and underflow flags are set by math instructions such as multiply, add, subtract and 1/x. These flags may also be connected to the peripheral interrupt expansion (PIE) block on your device. This can be useful for debugging underflow and overflow conditions within an application.

As on the C28x, program flow is controlled by C28x instructions that read status flags in the status register 0 (ST0). If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into ST0 flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVST0 FLAG instruction is used to load the current value of specified STF flags into the respective bits of ST0. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified.

**Example 2-1. Moving STF Flags to the ST0 Register**

```
Loop:
MOV32 R0H,*XAR4++
MOV32 R1H,*XAR3++
CMPF32 R1H, R0H
MOVST0 ZF, NF ; Move ZF and NF to ST0
BF Loop, GT ; Loop if (R1H > R0H)
```
### Figure 2-3. Floating-point Unit Status Register (STF)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>SHDWS</td>
<td></td>
<td>Shadow Mode Status Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>This bit is forced to 0 by the RESTORE instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>This bit is set to 1 by the SAVE instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bit is not affected by loading the status register either from memory or from the shadow values.</td>
</tr>
<tr>
<td>30 - 11</td>
<td>Reserved</td>
<td></td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>10</td>
<td>RND64</td>
<td></td>
<td>Round 64-bit Floating-Point Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>If this bit is zero, the MPYF64, ADDF64 and SUBF64 instructions will round to zero (truncate).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>If this bit is one, the MPYF64, ADDF64 and SUBF64 instructions will round to the nearest even value.</td>
</tr>
<tr>
<td>9</td>
<td>RND32</td>
<td></td>
<td>Round 32-bit Floating-Point Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>If this bit is zero, the MPYF32, ADDF32 and SUBF32 instructions will round to zero (truncate).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>If this bit is one, the MPYF32, ADDF32 and SUBF32 instructions will round to the nearest even value.</td>
</tr>
<tr>
<td>8 - 7</td>
<td>Reserved</td>
<td></td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>6</td>
<td>TF</td>
<td></td>
<td>Test Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>The condition tested with the TESTTF instruction is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The condition tested with the TESTTF instruction is true.</td>
</tr>
<tr>
<td>5</td>
<td>ZI</td>
<td></td>
<td>Zero Integer Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>The integer value is not zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The integer value is zero.</td>
</tr>
<tr>
<td>4</td>
<td>NI</td>
<td></td>
<td>Negative Integer Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>The integer value is not negative.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The integer value is negative.</td>
</tr>
<tr>
<td>3</td>
<td>ZF</td>
<td></td>
<td>Zero Floating-Point Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>The floating-point value is not zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The floating-point value is zero.</td>
</tr>
</tbody>
</table>

---

#### Table 2-2. Floating-point Unit Status (STF) Register Field Descriptions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>SHDWS</td>
<td>0</td>
<td>Shadow Mode Status Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>This bit is set to 1 by the SAVE instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>10</td>
<td>RND64</td>
<td>0</td>
<td>Round 64-bit Floating-Point Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Round 32-bit Floating-Point Mode</td>
</tr>
<tr>
<td>9</td>
<td>RND32</td>
<td>0</td>
<td>Round 32-bit Floating-Point Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Round 32-bit Floating-Point Mode</td>
</tr>
<tr>
<td>8 - 7</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>6</td>
<td>TF</td>
<td>0</td>
<td>Test Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Test Flag</td>
</tr>
<tr>
<td>5</td>
<td>ZI</td>
<td>0</td>
<td>Zero Integer Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Zero Integer Flag</td>
</tr>
<tr>
<td>4</td>
<td>NI</td>
<td>0</td>
<td>Negative Integer Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Negative Integer Flag</td>
</tr>
<tr>
<td>3</td>
<td>ZF</td>
<td>0</td>
<td>Zero Floating-Point Flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Zero Floating-Point Flag</td>
</tr>
</tbody>
</table>

---

During floating-point operations, a negative zero floating-point value is treated as a positive zero if the ZF and NF flags are configured.

---

(1) A negative zero floating-point value is treated as a positive zero value when configuring the ZF and NF flags.

(2) A DeNorm floating-point value is treated as a positive zero value when configuring the ZF and NF flags.
<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 2    | NF    |       | Negative Floating-Point Flag \(^{[1]} \[2]\)  
      |       | 0     | The floating-point value is not negative.  
      |       | 1     | The floating-point value is negative.  
| 1    | LUF   |       | Latched Underflow Floating-Point Flag  
      |       | 0     | An underflow condition has not been latched. If the MOVST0 instruction is used to copy this bit to ST0, then LUF will be cleared.  
      |       | 1     | An underflow condition has been latched.  
| 0    | LVF   |       | Latched Overflow Floating-Point Flag  
      |       | 0     | An overflow condition has not been latched. If the MOVST0 instruction is used to copy this bit to ST0, then LVF will be cleared.  
      |       | 1     | An overflow condition has been latched.  

The following instructions modify this flag based on the floating-point value stored in the destination register:
- MOV32, MOV32, MOVDD32, ABSF32, NEGF32, ABSF64, NEGF64, CMPF64, MAXF64, and MINF64
- The CMPF32, MAXF32, MINF32, CMPF64, MAXF64, and MINF64 instructions modify this flag based on the result of the operation.
- The SETFLG and SAVE instructions can also be used to modify this flag.

The following instructions will set this flag to 1 if an underflow occurs:
- MPYF32, ADDF32, SUBF32, MACF32, EINV32, EISQRTF32, MPYF64, ADDF64, SUBF64, MACF64, EINV64, EISQRTF64

The following instructions will set this flag to 1 if an overflow occurs:
- MPYF32, ADDF32, SUBF32, MACF32, EINV32, EISQRTF32, MPYF64, ADDF64, SUBF64, MACF64, EINV64, EISQRTF64
2.3.1.2 Repeat Block Register (RB)

The repeat block instruction (RPTB) is a new instruction for C28x+FPU64. This instruction allows you to repeat a block of code as shown in Example 2-2.

Example 2-2. The Repeat Block (RPTB) Instruction uses the RB Register

```assembly
; find the largest element and put its address in XAR6
MOV32 R0H, *XAR0++;
.align 2 ; Aligns the next instruction to an even address
NOP ; Makes RPTB odd aligned - required for a block size of 8
RPTB VECTOR_MAX_END, AR7 ; RA is set to 1
MOVL ACC,XAR0
MOV32 R1H,*XAR0++ ; RSIZE reflects the size of the RPTB block
MAXF32 R0H,R1H ; in this case the block size is 8
MOVST0 NF,ZF
MOVL XAR6,ACC,LT
VECTOR_MAX_END: ; RE indicates the end address. RA is cleared
```

The C28x+FPU64 hardware automatically populates the RB register based on the execution of a RPTB instruction. This register is not normally read by the application and does not accept debugger writes.

![Figure 2-4. Repeat Block Register (RB)](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RAS</td>
<td>0</td>
<td>Repeat Block Active Shadow Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>A repeat block was not active when the interrupt was taken.</td>
</tr>
<tr>
<td>30</td>
<td>RA</td>
<td>0</td>
<td>Repeat Block Active Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>This bit is cleared when the repeat counter, RC, reaches zero.</td>
</tr>
<tr>
<td>29-23</td>
<td>RSIZE</td>
<td>0-7</td>
<td>Repeat Block Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8/9-0x7F</td>
<td>A RPTB block that starts at an even address must include at least 9 16-bit words and a block that starts at an odd address must include at least 8 16-bit words. The maximum block size is 127 16-bit words. The codegen assembler will check for proper block size and alignment.</td>
</tr>
</tbody>
</table>

Table 2-3. Repeat Block (RB) Register Field Descriptions
### Table 2-3. Repeat Block (RB) Register Field Descriptions (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-16</td>
<td>RE</td>
<td></td>
<td>Repeat Block End Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This 7-bit value specifies the end address location of the repeat block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The RE value is calculated by hardware based on the RSIZE field and the PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>value when the RPTB instruction is executed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RE = lower 7 bits of (PC + 1 + RSIZE)</td>
</tr>
<tr>
<td>15-0</td>
<td>RC</td>
<td>0</td>
<td>Repeat Count</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The block will not be repeated; it will be executed only once. In this case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the repeat active, RA, bit will not be set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1-0xFFFF</td>
<td>This 16-bit value determines how many times the block will repeat.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The counter is initialized when the RPTB instruction is executed and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>is decremented when the PC reaches the end of the block. When the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>counter reaches zero, the repeat active bit is cleared and the block will</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>be executed one more time. Therefore the total number of times the block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>is executed is RC+1.</td>
</tr>
</tbody>
</table>

#### 2.4 Pipeline

The pipeline flow for C28x instructions is identical to that of the C28x CPU described in *TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430)*. Some floating-point instructions, however, use additional execution phases and thus require a delay to allow the operation to complete. This pipeline alignment is achieved by inserting NOPs or non-conflicting instructions when required. Software control of delay slots allows you to improve performance of an application by taking advantage of the delay slots and filling them with non-conflicting instructions. This section describes the key characteristics of the pipeline with regards to floating-point instructions. The rules for avoiding pipeline conflicts are small in number and simple to follow and the C28x+FPU64 assembler will help you by issuing errors for conflicts.

#### 2.4.1 Pipeline Overview

The C28x + FPU64 pipeline is identical to the C28x pipeline for all standard C28x instructions. In the decode2 stage (D2), it is determined if an instruction is a C28x instruction or a floating-point unit instruction. The pipeline flow is shown in Figure 2-5. Notice that stalls due to normal C28x pipeline stalls (D2) and memory waitstates (R2 and W) will also stall any C28x FPU64 instruction. Some C28x FPU64 instructions are single cycle and will complete in the FPU E1 or W stage which aligns to the C28x pipeline. Some instructions will take an additional execute cycle (E2,E3). For these instructions you must wait a cycle or two cycles for the result from the instruction to be available. The rest of this section will describe when delay cycles are required. Keep in mind that the assembly tools for the C28x+FPU64 will issue an error if a delay slot has not been handled correctly.

**Figure 2-5. FPU64 Pipeline**

```
F1  F2  D1  D2  R1  R2  E  W  W

D  R  E1  E2  E3

Load
Store
CMP/MIN/MAX/NEG/ABS
MPY/ADD/SUB/MACF32
MPY/ADD/SUBF64
```
2.4.2 General Guidelines for Floating-Point Pipeline Alignment

While the C28x+FPU64 assembler will issue errors for pipeline conflicts, you may still find it useful to understand when software delays are required. This section describes three guidelines you can follow when writing C28x+FPU64 assembly code.

Floating-point instructions that require delay slots have a ‘p’ after their cycle count. For example ‘2p’ stands for 2 pipelined cycles; ‘3p’ stands for 3 pipelined cycles. This means that an instruction can be started every cycle, but the result of the instruction will only be valid one or two instructions later.

There are three general guidelines to determine if an instruction needs a delay slot:
1. Single-precision floating-point math operations (multiply, addition, subtraction, 1/x and MAC) require 1 delay slot.
2. Double-precision Floating-point math operations (multiply, addition, subtraction, 1/x) require 2 delay slots.
4. Double-precision Conversion instructions between integer and floating-point formats require 2 delay slots.
5. Everything else does not require a delay slot. This includes minimum, maximum, compare, load, store, negative and absolute value instructions.

There are two exceptions to these rules. First, moves between the CPU and FPU registers require special pipeline alignment that is described later in this section. These operations are typically infrequent. Second, the MACF32 R7H, R3H, mem32, *XAR7 instruction has special requirements that make it easier to use. Refer to the MACF32 instruction description for details.

An example of the 32-bit ADDF32 instruction is shown in Example 2-3. ADDF32 is a 2p instruction and therefore requires one delay slot. The destination register for the operation, R0H, will be updated one cycle after the instruction for a total of 2 cycles. Therefore, a NOP or instruction that does not use R0H must follow this instruction.

Any memory stall or pipeline stall will also stall the floating-point unit. This keeps the floating-point unit aligned with the C28x pipeline and there is no need to change the code based on the waitstates of a memory block.

Please note that on certain devices instructions make take additional cycles to complete under specific conditions. These exceptions will be documented in the device errata.

Example 2-3. 2p Instruction Pipeline Alignment

```
ADDF32 R0H, #1.5, R1H ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- ADDF32 completes, R0H updated
NOP ; Any instruction
```
2.4.3 Moves from FPU Registers to C28x Registers

When transferring from the floating-point unit registers to the C28x CPU registers, additional pipeline alignment is required as shown in Example 2-4 and Example 2-5.

Example 2-4. Floating-Point to C28x Register Software Pipeline Alignment

```plaintext
; MINF32: 32-bit floating-point minimum: single-cycle operation
; An alignment cycle is required before copying R0H to ACC
MINF32 R0H, R1H ; Single-cycle instruction
    ; <-- R0H is valid
NOP ; Alignment cycle
MOV32 @ACC, R0H ; Copy R0H to ACC
```

For 1-cycle FPU instructions, one delay slot is required between a write to the floating-point register and the transfer instruction as shown in Example 2-4. For 2p FPU instructions, two delay slots are required between a write to the floating-point register and the transfer instruction as shown in Example 2-5. For 3p FPU instructions, three delay slots are required between a write to the floating-point register and the transfer instruction.

Example 2-5. Floating-Point to C28x Register Software Pipeline Alignment

```plaintext
; ADDF32: 32-bit floating-point addition: 2p operation
; An alignment cycle is required before copying R0H to ACC
ADDF32 R0H, R1H, #2 ; R0H = R1H + 2, 2 pipeline cycle instruction
    ; <-- R0H is valid
NOP ; Alignment cycle
NOP ; 3rd NOP
MOV32 @ACC, R0H ; Copy R0H to ACC
```

There is an exception for moves from FPU register to C28x register for the result of ADDF32/SUBF32/MPYF32 instructions. They are 2p cycle instructions but 3 NOPs are needed. This has gone into errata also.

Please refer to page 13 – http://www.ti.com/lit/er/sprz272k/sprz272k.pdf
2.4.4 Moves from C28x Registers to FPU Registers

Transfers from the standard C28x CPU registers to the floating-point registers require four alignment cycles. For the 2833x, 2834x, 2806x, 28M35xx and 28M26xx, the four alignment cycles can be filled with NOPs or any non-conflicting instruction except for F32TOUI32 RaH, RbH, FRACF32 RaH, RbH, UI16TOF32 RaH, mem16 and UI16TOF32 RaH, RbH. These instructions cannot replace any of the four alignment NOPs. On newer devices any non-conflicting instruction can go into the four alignment cycles. Please refer to the device errata for specific exceptions to these rules.

Example 2-6. C28x Register to Floating-Point Register Software Pipeline Alignment

```
; Four alignment cycles are required after copying a standard 28x CPU register to a floating-point register.
; MOV32 R0H, @ACC ; Copy ACC to R0H
NOP
NOP
NOP
NOP ; Wait 4 cycles
ADDF32 R2H, R1H, R0H ; R0H is valid
```

2.4.5 Parallel Instructions

Parallel instructions are single opcodes that perform two operations in parallel. This can be a math operation in parallel with a move operation, or two math operations in parallel. Math operations with a parallel move are referred to as 2p/1 or 3p/1 instructions. The math portion of the operation takes two or three pipelined cycles while the move portion of the operation is single cycle. This means that NOPs or other non-conflicting instructions must be inserted to align the math portion of the operation. An example of an add with parallel move instruction is shown in Example 2-7.

Example 2-7. 2p/1 Parallel Instruction Software Pipeline Alignment

```
; ADDF32 || MOV32 instruction: 32-bit floating-point add with parallel move
; ADDF32 is a 2p operation
; MOV32 is a 1 cycle operation
; ADDF32 R0H, R1H, #2 ; R0H = R1H + 2, 2 pipeline cycle operation
|| MOV32 R1H, @Val ; R1H gets the contents of Val, single cycle operation
NOP ; 1 cycle delay or non-conflicting instruction
|| NOP ; Any instruction
```
Parallel math instructions are referred to as 2p/2p or 3p/3p instructions. Both math operations take 2 or 3 cycles to complete. This means that NOPs or other non conflicting instructions must be inserted to align the both math operations. An example of a multiply with parallel add instruction is shown in Example 2-8.

**Example 2-8. 2p/2p Parallel Instruction Software Pipeline Alignment**

```plaintext
; MPYF32 R0H, R1H, R3H ; R0H = R1H * R3H, 2 pipeline cycle operation
; ADDF32 R1H, R2H, R4H ; R1H = R2H + R4H, 2 pipeline cycle operation
NOP ; 1 cycle delay or non-conflicting instruction
; <-- MPYF32 and ADDF32 complete here (R0H and R1H are valid)
NOP ; Any instruction
```

### 2.4.6 Invalid Delay Instructions

Most instructions can be used in delay slots as long as source and destination register conflicts are avoided. The C28x+FPU64 assembler will issue an error anytime you use an conflicting instruction within a delay slot. The following guidelines can be used to avoid these conflicts.

**NOTE:** Destination register conflicts in delay slots:

Any operation used for pipeline alignment delay must not use the same destination register as the instruction requiring the delay. See Example 2-9.

In Example 2-9 the MPYF32 instruction uses R2H as its destination register. The next instruction should not use R2H as its destination. Since the MOV32 instruction uses the R2H register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than R2H for the MOV32 instruction as shown in Example 2-10.

**Example 2-9. Destination Register Conflict**

```plaintext
; Invalid delay instruction. Both instructions use the same destination register
MPYF32 R2H, R1H, R0H ; 2p instruction
MOV32 R2H, mem32 ; Invalid delay instruction
```

**Example 2-10. Destination Register Conflict Resolved**

```plaintext
; Valid delay instruction
MPYF32 R2H, R1H, R0H ; 2p instruction
MOV32 R3H, mem32 ; Valid delay
; <-- MPYF32 completes, R2H valid
```
NOTE: Instructions in delay slots cannot use the instruction's destination register as a source register.

Any operation used for pipeline alignment delay must not use the destination register of the instruction requiring the delay as a source register as shown in Example 2-11. For parallel instructions, the current value of a register can be used in the parallel operation before it is overwritten as shown in Example 2-13.

In Example 2-11 the MPYF32 instruction again uses R2H as its destination register. The next instruction should not use R2H as its source since the MPYF32 will take an additional cycle to complete. Since the ADDF32 instruction uses the R2H register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than R2H or by inserting a non-conflicting instruction between the MPYF32 and ADDF32 instructions. Since the SUBF32 does not use R2H this instruction can be moved before the ADDF32 as shown in Example 2-12.

Example 2-11. Destination/Source Register Conflict

```plaintext
; Invalid delay instruction. ADDF32 should not use R2H as a source operand
MPYF32 R2H, R1H, R0H ; 2p instruction
ADDF32 R3H, R3H, R2H ; Invalid delay instruction
SUBF32 R4H, R1H, R0H
```

Example 2-12. Destination/Source Register Conflict Resolved

```plaintext
; Valid delay instruction.
MPYF32 R2H, R1H, R0H ; 2p instruction
SUBF32 R4H, R1H, R0H ; Valid delay for MPYF32
ADDF32 R3H, R3H, R2H ; <-- MPYF32 completes, R2H valid
NOP ; <-- SUBF32 completes, R4H valid
```

It should be noted that a source register for the 2nd operation within a parallel instruction can be the same as the destination register of the first operation. This is because the two operations are started at the same time. The 2nd operation is not in the delay slot of the first operation. Consider Example 2-13 where the MPYF32 uses R2H as its destination register. The MOV32 is the 2nd operation in the instruction and can freely use R2H as a source register. The contents of R2H before the multiply will be used by MOV32.

Example 2-13. Parallel Instruction Destination/Source Exception

```plaintext
; Valid parallel operation.
MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32 mem32, R2H ; <-- Uses R2H before the MPYF32
|| ; <-- mem32 updated
NOP ; <-- Delay for MPYF32
; <-- R2H updated
```
Likewise, the source register for the 2nd operation within a parallel instruction can be the same as one of the source registers of the first operation. The MPYF32 operation in Example 2-14 uses the R1H register as one of its sources. This register is also updated by the MOV32 register. The multiplication operation will use the value in R1H before the MOV32 updates it.

**Example 2-14. Parallel Instruction Destination/Source Exception**

```
; Valid parallel instruction
MPYF32 R2H, R1H, ROH ; 2p/1 instruction
|| MOV32 R1H, mem32 ; Valid
NOP ; <-- MOV32 completes, R1H valid
; <-- MPYF32, R2H valid
```

**NOTE:** Operations within parallel instructions cannot use the same destination register.

When two parallel operations have the same destination register, the result is invalid.

For example, see Example 2-15.

If both operations within a parallel instruction try to update the same destination register as shown in Example 2-15 the assembler will issue an error.

**Example 2-15. Invalid Destination Within a Parallel Instruction**

```
; Invalid parallel instruction. Both operations use the same destination register
MPYF32 R2H, R1H, ROH ; 2p/1 instruction
|| MOV32 R2H, mem32 ; Invalid
```

Some instructions access or modify the STF flags. Because the instruction requiring a delay slot will also be accessing the STF flags, these instructions should not be used in delay slots. These instructions are SAVE, SETFLG, RESTORE and MOVST0.

**NOTE:** Do not use SAVE, SETFLG, RESTORE, or the MOVST0 instruction in a delay slot.
2.4.7 Optimizing the Pipeline

The following example shows how delay slots can be used to improve the performance of an algorithm. The example performs two \( Y = MX+B \) operations. In Example 2-16, no optimization has been done. The \( Y = MX+B \) calculations are sequential and each takes 7 cycles to complete. Notice there are NOPs in the delay slots that could be filled with non-conflicting instructions. The only requirement is these instructions must not cause a register conflict or access the STF register flags.

Example 2-16. Floating-Point Code Without Pipeline Optimization

```assembly
; Using NOPs for alignment cycles, calculate the following:
;
; Y1 = M1*X1 + B1
; Y2 = M2*X2 + B2
;
; Calculate Y1

MOV32 R0H,@M1 ; Load R0H with M1 - single cycle
MOV32 R1H,@X1 ; Load R1H with X1 - single cycle
MPYF32 R1H,R1H,R0H ; R1H = M1 * X1 - 2p operation
|| MOV32 R0H,@B1 ; Load R0H with B1 - single cycle
NOP ; Wait for MPYF32 to complete
|<-- MPYF32 completes, R1H is valid
ADDF32 R1H,R1H,R0H ; R1H = R1H + R0H - 2p operation
NOP ; Wait for ADDF32 to complete
|<-- ADDF32 completes, R1H is valid
MOV32 @Y1,R1H ; Save R1H in Y1 - single cycle

; Calculate Y2

MOV32 R0H,@M2 ; Load R0H with M2 - single cycle
MOV32 R1H,@X2 ; Load R1H with X2 - single cycle
MPYF32 R1H,R1H,R0H ; R1H = M2 * X2 - 2p operation
|| MOV32 R0H,@B2 ; Load R0H with B2 - single cycle
NOP ; Wait for MPYF32 to complete
|<-- MPYF32 completes, R1H is valid
ADDF32 R1H,R1H,R0H ; R1H = R1H + R0H
NOP ; Wait for ADDF32 to complete
|<-- ADDF32 completes, R1H is valid
MOV32 @Y1,R1H ; Save R1H in Y2

; 14 cycles
; 48 bytes
```

The code shown in Example 2-17 was generated by the C28x+FPU64 compiler with optimization enabled. Notice that the NOPs in the first example have now been filled with other instructions. The code for the two \( Y = MX+B \) calculations are now interleaved and both calculations complete in only nine cycles.
Example 2-17. Floating-Point Code With Pipeline Optimization

; Using non-conflicting instructions for alignment cycles,
; calculate the following:
; Y1 = M1*X1 + B1
; Y2 = M2*X2 + B2
;
MOV32 R2H, @X1 ; Load R2H with X1 - single cycle
MOV32 R1H, @M1 ; Load R1H with M1 - single cycle
MPYF32 R3H, R2H, R1H ; R3H = M1 * X1 - 2p operation
MOV32 ROH, @M2 ; Load ROH with M2 - single cycle
MOV32 R1H, @X2 ; Load R1H with X2 - single cycle
; <-- MPYF32 completes, R3H is valid
MOV32 ROH, R1H, ROH ; ROH = M2 * X2 - 2p operation
|| MOV32 R4H, @B1 ; Load R4H with B1 - single cycle
; <-- MOV32 completes, R4H is valid
ADDF32 R1H, R4H, R3H ; R1H = B1 + M1*X1 - 2p operation
|| MOV32 R2H, @B2 ; Load R2H with B2 - single cycle
; <-- MPYF32 completes, ROH is valid
ADDF32 ROH, R2H, ROH ; ROH = B2 + M2*X2 - 2p operation
; <-- ADDF32 completes, R1H is valid
MOV32 @Y1, R1H ; Store Y1
; <-- ADDF32 completes, ROH is valid
MOV32 @Y2, ROH ; Store Y2
;
; 9 cycles
; 36 bytes

2.5 Floating Point Unit (FPU64) Instruction Set

This chapter describes the assembly language instructions of the TMS320C28x plus floating-point processor FPU64. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are an extension to the standard C28x instruction set. For information on standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430).

2.5.1 Instruction Descriptions

This section gives detailed information on the instruction set. This section lists all the single precision floating point instructions and note that these are identical to the instructions available in C28x + FPU. Each instruction may present the following information:

• Operands
• Opcode
• Description
• Exceptions
• Pipeline
• Examples
• See also

The example INSTRUCTION is shown to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information. On the C28x+FPU64 instructions, follow the same format as the C28x. The source operand(s) are always on the right and the destination operand(s) are on the left.

The explanations for the syntax of the operands used in the instruction descriptions for the TMS320C28x plus floating-point processor are given in Table 2-4. For information on the operands of standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430).
### Table 2-4. Operand Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHHi</td>
<td>16-bit immediate (hex or float) value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FHHiHex</td>
<td>16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32Fhex</td>
<td>32-bit immediate value that represents an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32F</td>
<td>Immediate float value represented in floating-point representation</td>
</tr>
<tr>
<td>#0.0</td>
<td>Immediate zero</td>
</tr>
<tr>
<td>#RC</td>
<td>16-bit immediate value for the repeat count</td>
</tr>
<tr>
<td>*(0:16bitAddr)</td>
<td>16-bit immediate address, zero extended</td>
</tr>
<tr>
<td>CNDF</td>
<td>Condition to test the flags in the STF register</td>
</tr>
<tr>
<td>FLAG</td>
<td>Selected flags from STF register (OR) 11 bit mask indicating which floating-point status flags to change</td>
</tr>
<tr>
<td>label</td>
<td>Label representing the end of the repeat block</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
<tr>
<td>RaH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RbH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RcH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RdH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>ReH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RiH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RaL</td>
<td>R0L to R7L registers</td>
</tr>
<tr>
<td>RbL</td>
<td>R0L to R7L registers</td>
</tr>
<tr>
<td>RcL</td>
<td>R0L to R7L registers</td>
</tr>
<tr>
<td>RdL</td>
<td>R0L to R7L registers</td>
</tr>
<tr>
<td>ReL</td>
<td>R0L to R7L registers</td>
</tr>
<tr>
<td>RfL</td>
<td>R0L to R7L registers</td>
</tr>
<tr>
<td>RB</td>
<td>Repeat Block Register</td>
</tr>
<tr>
<td>STF</td>
<td>FPU Status Register</td>
</tr>
<tr>
<td>VALUE</td>
<td>Flag value of 0 or 1 for selected flag (OR) 11 bit mask indicating the flag value; 0 or 1</td>
</tr>
</tbody>
</table>
INSTRUCTION dest1, source1, source2 — Short Description

Operands

<table>
<thead>
<tr>
<th></th>
<th>Description for the 1st operand for the instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest1</td>
<td>description for the 1st operand for the instruction</td>
</tr>
<tr>
<td>source1</td>
<td>description for the 2nd operand for the instruction</td>
</tr>
<tr>
<td>source2</td>
<td>description for the 3rd operand for the instruction</td>
</tr>
</tbody>
</table>

Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).

Opcode

This section shows the opcode for the instruction.

Description

Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.

Restrictions

Any constraints on the operands or use of the instruction imposed by the processor are discussed.

Pipeline

This section describes the instruction in terms of pipeline cycles as described in Section 2.4.

Example

Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution. All examples assume the device is running with the OBJMODE set to 1. Normally the boot ROM or the c-code initialization will set this bit.

See Also

Lists related instructions.
2.5.2 Instructions

The instructions are listed alphabetically, preceded by a summary.

Table 2-5. Summary of Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSF32 RaH, RbH — 32-bit Floating-Point Absolute Value</td>
<td>169</td>
</tr>
<tr>
<td>ADDF32 RaH, #16FHi, RbH — 32-bit Floating-Point Addition</td>
<td>170</td>
</tr>
<tr>
<td>ADDF32 RaH, RbH, #16FHi — 32-bit Floating-Point Addition</td>
<td>172</td>
</tr>
<tr>
<td>ADDF32 RaH, RbH, RcH — 32-bit Floating-Point Addition</td>
<td>174</td>
</tr>
<tr>
<td>ADDF32 RdH, ReH, RFH</td>
<td>MOV32 mem32, RaH — 32-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>ADDF32 RdH, ReH, RFH</td>
<td>MOV32 RaH, mem32 — 32-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>CMPF32 RaH, RbH — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>180</td>
</tr>
<tr>
<td>CMPF32 RaH, #16FHi — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>181</td>
</tr>
<tr>
<td>CMPF32 RaH, #0.0 — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>183</td>
</tr>
<tr>
<td>EINVF32 RaH, RbH — 32-bit Floating-Point Reciprocal Approximation</td>
<td>184</td>
</tr>
<tr>
<td>EISQRTF32 RaH, RbH — 32-bit Floating-Point Square-Root Reciprocal Approximation</td>
<td>186</td>
</tr>
<tr>
<td>F32TOI16 RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Integer</td>
<td>188</td>
</tr>
<tr>
<td>F32TOI16R RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Integer and Round</td>
<td>189</td>
</tr>
<tr>
<td>F32TOI32 RaH, RbH — Convert 32-bit Floating-Point Value to 32-bit Integer</td>
<td>190</td>
</tr>
<tr>
<td>F32TOUI16 RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer</td>
<td>191</td>
</tr>
<tr>
<td>F32TOUI16R RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round</td>
<td>192</td>
</tr>
<tr>
<td>FRACF32 RaH, RbH — Fractional Portion of a 32-bit Floating-Point Value</td>
<td>193</td>
</tr>
<tr>
<td>I16TOF32 RaH, RbH — Convert 16-bit Integer to 32-bit Floating-Point Value</td>
<td>194</td>
</tr>
<tr>
<td>I16TOF32 RaH, mem16 — Convert 16-bit Integer to 32-bit Floating-Point Value</td>
<td>195</td>
</tr>
<tr>
<td>I32TOF32 RaH, RbH — Convert 32-bit Integer to 32-bit Floating-Point Value</td>
<td>196</td>
</tr>
<tr>
<td>I32TOF32 RaH, mem32 — Convert 32-bit Integer to 32-bit Floating-Point Value</td>
<td>197</td>
</tr>
<tr>
<td>MACF32 R3H, R2H, RdH, ReH, RFH — 32-bit Floating-Point Multiply with Parallel Add</td>
<td>198</td>
</tr>
<tr>
<td>MACF32 R3H, R2H, RdH, ReH, RFH</td>
<td>MOV32 RaH, mem32 — 32-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MACF32 R7H, R3H, mem32, *XAR7++ — 32-bit Floating-Point Multiply and Accumulate</td>
<td>201</td>
</tr>
<tr>
<td>MACF32 R7H, R6H, RdH, ReH, RFH — 32-bit Floating-Point Multiply with Parallel Add</td>
<td>202</td>
</tr>
<tr>
<td>MACF32 R7H, R6H, RdH, ReH, RFH</td>
<td>MOV32 RaH, mem32 — 32-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MAXF32 RaH, RbH — 32-bit Floating-Point Maximum</td>
<td>204</td>
</tr>
<tr>
<td>MAXF32 RaH, #16FHi — 32-bit Floating-Point Maximum</td>
<td>205</td>
</tr>
<tr>
<td>MAXF32 RaH, RbH</td>
<td>MOV32 Rch, RdH — 32-bit Floating-Point Maximum with Parallel Move</td>
</tr>
<tr>
<td>MINF32 RaH, RbH — 32-bit Floating-Point Minimum</td>
<td>207</td>
</tr>
<tr>
<td>MINF32 RaH, #16FHi — 32-bit Floating-Point Minimum</td>
<td>208</td>
</tr>
<tr>
<td>MINF32 RaH, RbH</td>
<td>MOV32 Rch, RdH — 32-bit Floating-Point Minimum with Parallel Move</td>
</tr>
<tr>
<td>MOV16 mem16, RaH — Move 16-bit Floating-Point Register Contents to Memory</td>
<td>210</td>
</tr>
<tr>
<td>MOV32 *(0:16bitAddr), loc32 — Move the Contents of loc32 to Memory</td>
<td>211</td>
</tr>
<tr>
<td>MOV32 ACC, RaH — Move 32-bit Floating-Point Register Contents to ACC</td>
<td>212</td>
</tr>
<tr>
<td>MOV32 loc32, *(0:16bitAddr) — Move 32-bit Value from Memory to loc32</td>
<td>213</td>
</tr>
<tr>
<td>MOV32 loc32, RaH — Move 32-bit Floating-Point Register Contents to Memory</td>
<td>214</td>
</tr>
<tr>
<td>MOV32 mem32, STF — Move 32-bit STF Register to Memory</td>
<td>215</td>
</tr>
<tr>
<td>MOV32 P, RaH — Move 32-bit Floating-Point Register Contents to P</td>
<td>216</td>
</tr>
<tr>
<td>MOV32 RaH, ACC — Move the Contents of ACC to a 32-bit Floating-Point Register</td>
<td>217</td>
</tr>
<tr>
<td>MOV32 RaH, mem32 (, CNDF) — Conditional 32-bit Move</td>
<td>218</td>
</tr>
</tbody>
</table>

Copyright © 2014–2019, Texas Instruments Incorporated
### Table 2-5. Summary of Instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV32 RaH, P</td>
<td>Move the Contents of P to a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MOV32 RaH, RbH (CNDF)</td>
<td>Conditional 32-bit Move</td>
</tr>
<tr>
<td>MOV32 RaH, XARn</td>
<td>Move the Contents of XARn to a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MOV32 RaH, XT</td>
<td>Move the Contents of XT to a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MOV32 STF, mem32</td>
<td>Move 32-bit Value from Memory to the STF Register</td>
</tr>
<tr>
<td>MOV32 XARn, RaH</td>
<td>Move 32-bit Floating-Point Register Contents to XARn</td>
</tr>
<tr>
<td>MOV32 XT, RaH</td>
<td>Move 32-bit Floating-Point Register Contents to XT</td>
</tr>
<tr>
<td>MOV32 RaH, mem32</td>
<td>Move 32-bit Value from Memory with Data Copy</td>
</tr>
<tr>
<td>MOV32 RaH, #32F</td>
<td>Load the 32-bits of a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MOV32 RaH, #32FHex</td>
<td>Load the 32-bits of a 32-bit Floating-Point Register with the immediate</td>
</tr>
<tr>
<td>MOVIZ RaH, #16FHiHex</td>
<td>Load the Upper 16-bits of a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MOVIZF32 RaH, #16FHi</td>
<td>Load the Upper 16-bits of a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MOVST0 FLAG</td>
<td>Load Selected STF Flags into ST0</td>
</tr>
<tr>
<td>MOVXI RaH, #16FloHex</td>
<td>Move Immediate to the Low 16-bits of a Floating-Point Register</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RcH</td>
<td>32-bit Floating-Point Multiply</td>
</tr>
<tr>
<td>MPYF32 RaH, #16FHi, RbH</td>
<td>32-bit Floating-Point Multiply</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, #16FHi</td>
<td>32-bit Floating-Point Multiply</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RCH</td>
<td>32-bit Floating-Point Multiply with Parallel Add</td>
</tr>
<tr>
<td>MPYF32 RdH, ReH, RfH</td>
<td>32-bit Floating-Point Multiply with Parallel Move</td>
</tr>
<tr>
<td>MPYF32 RdH, ReH, RfH [MOV32 RaH, mem32]</td>
<td>32-bit Floating-Point Multiply with Parallel Move</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RCH [SUBF32 RdH, ReH, RfH]</td>
<td>32-bit Floating-Point Multiply with Parallel Subtract</td>
</tr>
<tr>
<td>NEG32 RaH, RbH (CNDF)</td>
<td>Conditional Negation</td>
</tr>
<tr>
<td>POP RB</td>
<td>Pop the RB Register from the Stack</td>
</tr>
<tr>
<td>PUSH RB</td>
<td>Push the RB Register onto the Stack</td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore the Floating-Point Registers</td>
</tr>
<tr>
<td>RPTB label, loc16</td>
<td>Repeat A Block of Code</td>
</tr>
<tr>
<td>RPTB label, #RC</td>
<td>Repeat a Block of Code</td>
</tr>
<tr>
<td>SAVE FLAG, VALUE</td>
<td>Save Register Set to Shadow Registers and Execute SETFLG</td>
</tr>
<tr>
<td>SETFLG FLAG, VALUE</td>
<td>Set or clear selected floating-point status flags</td>
</tr>
<tr>
<td>SUBF32 RaH, RbH, RcH</td>
<td>32-bit Floating-Point Subtraction</td>
</tr>
<tr>
<td>SUBF32 RaH, #16FHi, RbH</td>
<td>32-bit Floating Point Subtraction</td>
</tr>
<tr>
<td>SUBF32 RdH, ReH, RfH [MOV32 RaH, mem32]</td>
<td>32-bit Floating-Point Subtraction with Parallel Move</td>
</tr>
<tr>
<td>SUBF32 RdH, ReH, RfH [MOV32 mem32, RaH]</td>
<td>32-bit Floating-Point Subtraction with Parallel Move</td>
</tr>
<tr>
<td>SWAPF RaH, RbH (CNDF)</td>
<td>Conditional Swap</td>
</tr>
<tr>
<td>TESTTF CNDF</td>
<td>Test STF Register Flag Condition</td>
</tr>
<tr>
<td>UI16TOF32 RaH, mem16</td>
<td>Convert unsigned 16-bit integer to 32-bit floating-point value</td>
</tr>
<tr>
<td>UI16TOF32 RaH, RbH</td>
<td>Convert unsigned 16-bit integer to 32-bit floating-point value</td>
</tr>
<tr>
<td>UI32TOF32 RaH, mem32</td>
<td>Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value</td>
</tr>
<tr>
<td>UI32TOF32 RaH, RbH</td>
<td>Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value</td>
</tr>
<tr>
<td>ZERO RaH</td>
<td>Zero the Floating-Point Register RaH</td>
</tr>
<tr>
<td>ZEROA</td>
<td>Zero All Floating-Point Registers</td>
</tr>
<tr>
<td>MOV32 RaL, mem32 (CNDF)</td>
<td>Conditional 32-bit Move</td>
</tr>
<tr>
<td>MOVDD32 RaL, mem32</td>
<td>Move From Register To Memory 32-bit Move</td>
</tr>
<tr>
<td>MOVDD32 RaH, mem32</td>
<td>Move From Register To Memory 32-bit Move</td>
</tr>
<tr>
<td>MOV32 mem32, RaL</td>
<td>Move From Memory to Register 32-bit Move</td>
</tr>
<tr>
<td>MOVIX RaL, #16</td>
<td>Load the Upper 16-bits of a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MOVXI RaL, #16</td>
<td>Load the Lower 16-bits of a 32-bit Floating-Point Register</td>
</tr>
<tr>
<td>MPYF64 Rd, Re, Rf [MOV32 RaL, mem32]</td>
<td>64-bit Floating-Point Multiply with Parallel Move</td>
</tr>
<tr>
<td>MPYF64 Rd, Re, Rf [MOV32 mem32, RaL]</td>
<td>64-bit Floating-Point Multiply with Parallel Move</td>
</tr>
</tbody>
</table>
### Floating Point Unit (FPU64) Instruction Set

Table 2-5. Summary of Instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaL,mem32 — 64-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>ADDF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 mem32, RaL — 64-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>SUBF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaL,mem32 — 64-bit Floating-Point Subtraction with Parallel Move</td>
</tr>
<tr>
<td>SUBF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 mem32, RaL — 64-bit Floating-Point Subtraction with Parallel Move</td>
</tr>
<tr>
<td>MACF64 R3,R2,Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaL,mem32 — 64-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MACF64 R7,R6,Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaL,mem32 — 64-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MPYF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaH,mem32 — 64-bit Floating-Point Multiply with Parallel Move</td>
</tr>
<tr>
<td>MPYF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 mem32, RaH — 64-bit Floating-Point Multiply with Parallel Move</td>
</tr>
<tr>
<td>ADDF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaH,mem32 — 64-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>ADDF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 mem32, RaH — 64-bit Floating-Point Addition with Parallel Move</td>
</tr>
<tr>
<td>SUBF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaH,mem32 — 64-bit Floating-Point Subtraction with Parallel Move</td>
</tr>
<tr>
<td>SUBF64 Rd,Re,Rf</td>
<td></td>
<td>MOV32 mem32, RaH — 64-bit Floating-Point Subtraction with Parallel Move</td>
</tr>
<tr>
<td>MACF64 R3,R2,Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaH,mem32 — 64-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MACF64 R7,R6,Rd,Re,Rf</td>
<td></td>
<td>MOV32 RaH,mem32 — 64-bit Floating-Point Multiply and Accumulate with Parallel Move</td>
</tr>
<tr>
<td>MPYF64 Ra,Rb,Rc</td>
<td></td>
<td>ADD64 Rd,Re,Rf — 64-bit Floating-Point Multiply with Parallel Addition</td>
</tr>
<tr>
<td>MPYF64 Ra,Rb,Rc</td>
<td></td>
<td>SUB64 Rd,Re,Rf — 64-bit Floating-Point Multiply with Parallel Subtraction</td>
</tr>
<tr>
<td>MPYF64 Ra,Rb,Rc</td>
<td></td>
<td>MOV32 RaH,mem32</td>
</tr>
<tr>
<td>ADDF64 Ra,Rb,Rc — 64-bit Floating-Point Addition</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td>SUBF64 Ra,Rb,Rc — 64-bit Floating-Point Subtraction</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>MPYF64 Ra,Rb,Rc</td>
<td></td>
<td>MOV32 RaH,mem32 — 64-bit Floating-Point Multiply with Parallel Move</td>
</tr>
<tr>
<td>ADDF64 Ra,Rb,#16F OR ADDF64 Ra,#16F, Rb — 64-bit Floating-Point Addition</td>
<td>306</td>
<td></td>
</tr>
<tr>
<td>SUBF64 Ra,#16F,Rb — 64-bit Floating-Point Subtraction</td>
<td>307</td>
<td></td>
</tr>
<tr>
<td>CMPF64 Ra, Rb — 64-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>308</td>
<td></td>
</tr>
<tr>
<td>CMPF64 Ra,#16F — 64-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>309</td>
<td></td>
</tr>
<tr>
<td>CMPF64 Ra,#0.0 — 64-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>310</td>
<td></td>
</tr>
<tr>
<td>MAXF64 Ra, Rb — 64-bit Floating-Point Maximum</td>
<td>311</td>
<td></td>
</tr>
<tr>
<td>MAXF64 Ra, Rb</td>
<td></td>
<td>MOV32 Rc,Rd — 64-bit Floating-Point Maximum with Parallel Move</td>
</tr>
<tr>
<td>MAXF64 Ra, #16F — 64-bit Floating-Point Maximum</td>
<td>313</td>
<td></td>
</tr>
<tr>
<td>MINF64 Ra, Rb — 64-bit Floating-Point Minimum</td>
<td>314</td>
<td></td>
</tr>
<tr>
<td>MINF64 Ra, Rb</td>
<td></td>
<td>MOV32 Rc,Rd — 64-bit Floating-Point Minimum with Parallel Move</td>
</tr>
<tr>
<td>MINF64 Ra, #16F — 64-bit Floating-Point Minimum</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>F64TOI32 RaH,Re — Convert 64-bit Floating-Point Value to 32-bit Integer</td>
<td>317</td>
<td></td>
</tr>
<tr>
<td>F64TOUI32 RaH,Rb — Convert 64-bit Floating-Point Value to 32-bit Unsigned Integer</td>
<td>318</td>
<td></td>
</tr>
<tr>
<td>I32TOF64 Ra,mem32 — Convert 32-bit Integer to 64-bit Floating-Point Value</td>
<td>319</td>
<td></td>
</tr>
<tr>
<td>I32TOF64 Ra,RbH — Convert 32-bit Integer to 64-bit Floating-Point Value</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>UI32TOF64 Ra,mem32 — Convert unsigned 32-bit Integer to 64-bit Floating-Point Value</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td>F64TOI64 Ra,Rb — Convert 64-bit Floating-Point Value to 64-bit Integer</td>
<td>322</td>
<td></td>
</tr>
<tr>
<td>F64TOUI64 Ra,Rb — Convert 64-bit Floating-Point Value to 64-bit unsigned Integer</td>
<td>323</td>
<td></td>
</tr>
<tr>
<td>I64TOF64 Ra,Rb — Convert 64-bit Integer to 64-bit Floating-Point Value</td>
<td>324</td>
<td></td>
</tr>
<tr>
<td>UI64TOF64 Ra,Rb — Convert 64-bit unsigned Integer to 64-bit Floating-Point Value</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>I64TOF64 Ra,Rb — Convert 64-bit Integer to 64-bit Floating-Point Value</td>
<td>326</td>
<td></td>
</tr>
<tr>
<td>UI64TOF64 Ra,Rb — Convert 64-bit unsigned Integer to 64-bit Floating-Point Value</td>
<td>327</td>
<td></td>
</tr>
<tr>
<td>FRACF64 Ra,Rb — Fractional Portion of a 64-bit Floating-Point Value</td>
<td>328</td>
<td></td>
</tr>
<tr>
<td>F64TOF32 RaH,Rb — Convert 64-bit Floating-Point Value to 32-bit Floating-Point Value</td>
<td>329</td>
<td></td>
</tr>
<tr>
<td>F32TOF64 Ra,RbH — Convert 32-bit Floating-Point Value to 64-bit Floating-Point Value</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td>F32TOF64 Ra,mem32 — Convert 32-bit Floating-Point Value to 64-bit Floating-Point Value</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>F32TOF64 Ra,mem32 — Convert 32-bit Floating-Point Value to 64-bit Floating-Point Value</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>ABSF64 Ra, Rb — 64-bit Floating-Point Absolute Value</td>
<td>333</td>
<td></td>
</tr>
<tr>
<td>NEGF64 Ra, Rb( &amp;CNDF) — Conditional Negation</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>MOV64 Ra, Rb{, CNDF}</td>
<td>Conditional 64-bit Move</td>
<td>335</td>
</tr>
<tr>
<td>EISQRTF64 Ra, Rb</td>
<td>64-bit Floating-Point Square-Root Reciprocal Approximation</td>
<td>336</td>
</tr>
<tr>
<td>EINVF64 Ra, Rb</td>
<td>64-bit Floating-Point Reciprocal Approximation</td>
<td>337</td>
</tr>
</tbody>
</table>
ABSF32 RaH, RbH  

32-bit Floating-Point Absolute Value

**Operands**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1001 0101
MSW: 0000 0000 00bb baaa

**Description**

The absolute value of RbH is loaded into RaH. Only the sign bit of the operand is modified by the ABSF32 instruction.

\[
\text{if (RbH < 0)} \{ \text{RaH} = -\text{RbH} \}
\]

\[
\text{else} \{ \text{RaH} = \text{RbH} \}
\]

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

\[
\text{NF} = 0;
\]

\[
\text{ZF} = 0;
\]

\[
\text{if ( RaH[30:23] == 0)} \text{ ZF} = 1;
\]

**Pipeline**

This is a single-cycle instruction.

**Example**

- `MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)`
- `ABSF32 R1H, R1H ; R1H = 2.0 (0x40000000), ZF = NF = 0`
- `MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)`
- `ABSF32 R0H, R0H ; R0H = 5.0 (0x40A00000), ZF = NF = 0`
- `MOVIZF32 R0H, #0.0 ; R0H = 0.0`
- `ABSF32 R1H, R0H ; R1H = 0.0 ZF = 1, NF = 0`

**See also**

NEGF32 RaH, RbH, (CNDF)
ADDF32 RaH, #16FHi, RbH — 32-bit Floating-Point Addition

ADDF32 RaH, #16FHi, RbH 32-bit Floating-Point Addition

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 10II IIII
MSW: IIII IIII IIbb baaa

Description

Add RbH to the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

RaH = RbH + #16FHi:0

This instruction can also be written as ADDF32 RaH, RbH, #16FHi.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

ADDF32 RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- ADDF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

; Add to R1H the value 2.0 in 32-bit floating-point format
ADDF32 ROH, #2.0, R1H ; ROH = 2.0 + R1H
NOP ; Delay for ADDF32 to complete
; <-- ADDF32 completes, ROH updated
NOP

; Add to R3H the value -2.5 in 32-bit floating-point format
ADDF32 R2H, #-2.5, R3H ; R2H = -2.5 + R3H
NOP ; Delay for ADDF32 to complete
; <-- ADDF32 completes, R2H updated
NOP

; Add to R5H the value 0x3FC00000 (1.5)
ADDF32 R5H, #0x3FC0, R5H ; R5H = 1.5 + R5H
NOP ; Delay for ADDF32 to complete
; <-- ADDF32 completes, R5H updated
See also

ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
ADDF32 RaH, RbH, #16FHi — 32-bit Floating-Point Addition

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 10II IIII

MSW: IIII IIII IIbb baaa

Description

Add RbH to the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

RaH = RbH + #16FHi:0

This instruction can also be written as ADDF32 RaH, #16FHi, RbH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

```
ADDF32 RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- ADDF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

; Add to R1H the value 2.0 in 32-bit floating-point format
ADDF32 R0H, R1H, #2.0 ; R0H = R1H + 2.0
NOP ; Delay for ADDF32 to complete
    ; <-- ADDF32 completes, R0H updated
NOP

; Add to R3H the value -2.5 in 32-bit floating-point format
ADDF32 R2H, R3H, #(-2.5) ; R2H = R3H + (-2.5)
NOP ; Delay for ADDF32 to complete
    ; <-- ADDF32 completes, R2H updated
NOP

; Add to R5H the value 0x3FC00000 (1.5)
ADDF32 R5H, R5H, #0x3FC0 ; R5H = R5H + 1.5
NOP ; Delay for ADDF32 to complete
    ; <-- ADDF32 completes, R5H updated

172 Floating Point Unit (FPU64)
ADD32 RaH, RbH, #16FHi — 32-bit Floating-Point Addition

See also

ADD32 RaH, RbH, #16FHi
ADD32 RaH, RbH, RcH
ADD32 RdH, ReH, RfH || MOV32 RaH, mem32
ADD32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
ADDF32 RaH, RbH, RcH — 32-bit Floating-Point Addition

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0001 0000
MSW: 0000 000c ccbb baaa

Description

Add the contents of RcH to the contents of RbH and load the result into RaH.

RaH = RbH + RcH

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

ADDF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- ADDF32 completes, RaH updated

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate Y = M1*X1 + B1. This example assumes that M1, X1, B1 and Y are all on the same data page.

MOVW DP, #M1 ; Load the data page
MOV32 R0H, @M1 ; Load R0H with M1
MOV32 R1H, @X1 ; Load R1H with X1
MPYF32 R1H, R1H, R0H ; Multiply M1*X1
|| MOV32 R0H, @B1 ; and in parallel load R0H with B1
NOP ; <-- MOV32 complete
; <-- MPYF32 complete
ADDF32 R1H, R1H, R0H ; Add M*X1 to B1 and store in R1H
NOP ; <-- ADDF32 complete
MOV32 @Y1, R1H ; Store the result

Calculate Y = A + B

MOVL XAR4, #A
MOV32 R0H, *XAR4 ; Load R0H with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
ADDF32 R0H, R1H, R0H ; Add A + B R0H+R0H+R1H
MOVL XAR4, #Y
; < -- ADDF32 complete
MOV32 *XAR4, R0H ; Store the result

See also

ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, #16F, RbH
ADDF32 RdH, ReH, RIH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RiH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RiH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RiH
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH — 32-bit Floating-Point Addition with Parallel Move

**Operands**

- **RdH**: Floating-point destination register for the ADDF32 (R0H to R7H)
- **ReH**: Floating-point source register for the ADDF32 (R0H to R7H)
- **RfH**: Floating-point source register for the ADDF32 (R0H to R7H)
- **mem32**: Pointer to a 32-bit memory location. This will be the destination of the MOV32.
- **RaH**: Floating-point source register for the MOV32 (R0H to R7H)

**Opcode**

| LSW: | 1110 0000 0001 fffe |
| MSW: | eedd daaa mem32 |

**Description**

Perform an ADDF32 and a MOV32 in parallel. Add RfH to the contents of ReH and store the result in RdH. In parallel move the contents of RaH to the 32-bit location pointed to by mem32. mem32 addresses memory using any of the direct or indirect addressing modes supported by the C28x CPU.

\[
RdH = ReH + RfH, \\
[mem32] = RaH
\]

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- **LUF** = 1 if ADDF32 generates an underflow condition.
- **LVF** = 1 if ADDF32 generates an overflow condition.

**Pipeline**

ADDF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

\[
\begin{align*}
\text{ADDF32 RdH, ReH, RfH} & \quad \text{; 2 pipeline cycles (2p)} \\
|| \text{MOV32 mem32, RaH} & \quad \text{; 1 cycle} \\
\text{NOP} & \quad \text{; 1 cycle delay or non-conflicting instruction} \\
\text{NOP} & \quad \text{; 1 cycle delay or non-conflicting instruction}
\end{align*}
\]

Any instruction in the delay slot must not use RdH as a destination register or use RdH as a source operand.

**Example**

\[
\begin{align*}
\text{ADDF32 R3H, R6H, R4H} & \quad \text{; (A) R3H = R6H + R4H and R7H = I3} \\
|| \text{MOV32 R7H, *-SP[2]} & \quad ; \text{R7H valid} \\
\text{SUBF32 R6H, R6H, R4H} & \quad \text{; (B) R6H = R6H - R4H} \\
\text{|| MOV32 *+XAR5[2], R3H} & \quad ; \text{ADDF32 (A) completes, R3H valid} \\
\text{SUBF32 R3H, R1H, R7H} & \quad \text{; (C) R3H = R1H - R7H and store R3H (A)} \\
\text{|| MOV32 *+XAR5[2], R3H} & \quad ; \text{SUBF32 (B) completes, R6H valid} \\
\text{ADDF32 R4H, R7H, R1H} & \quad \text{; (D) completes, (A) stored} \\
\text{|| MOV32 *+XAR5[6], R6H} & \quad ; \text{R4H = D = R7H + R1H and store R6H (B)} \\
\text{MOV32 *+XAR5[0], R3H} & \quad ; \text{store R3H (C)} \\
\text{|| ADDF32 (D) completes, R4H valid} \\
\text{MOV32 *+XAR5[4], R4H} & \quad ; \text{store R4H (D)}
\end{align*}
\]
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH — 32-bit Floating-Point Addition with Parallel Move

; <-- MOV32 completes, (D) stored

See also
ADDF32 RaH, #16FHi, RbH
ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32 — 32-bit Floating-Point Addition with Parallel Move

Operands

| RdH | Floating-point destination register for the ADDF32 (R0H to R7H). RdH cannot be the same register as RaH. |
| ReH | Floating-point source register for the ADDF32 (R0H to R7H) |
| RfH | Floating-point source register for the ADDF32 (R0H to R7H) |
| RaH | Floating-point destination register for the MOV32 (R0H to R7H). RaH cannot be the same register as RdH. |
| mem32 | pointer to a 32-bit memory location. This is the source for the MOV32. |

Opcode

LSW: 1110 0011 0001 fffe
MSW: eedd daaa mem32

Description

Perform an ADDF32 and a MOV32 operation in parallel. Add RfH to the contents of ReH and store the result in RdH. In parallel move the contents of the 32-bit location pointed to by mem32 to RaH. mem32 addresses memory using any of the direct or indirect addressing modes supported by the C28x CPU.

RdH = ReH + RfH,
RaH = [mem32]

Restrictions

The destination register for the ADDF32 and the MOV32 must be unique. That is, RaH and RdH cannot be the same register.

Any instruction in the delay slot must not use RdH as a destination register or use RdH as a source operand.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;

Pipeline

The ADDF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

ADDF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
|| MOV32 RaH, mem32 ; 1 cycle
; <-- MOV32 completes, RaH updated NOP
; 1 cycle delay or non-conflicting instruction
; <-- ADDF32 completes, RdH updated
NOP
Example

Calculate \( Y = A + B - C \):

```
MOVL XAR4, #A
MOV32 ROH, *XAR4 ; Load ROH with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
MOVL XAR4, #C
ADDF32 R0H,R1H,R0H ; Add A + B and in parallel
|| MOV32 R0H, *XAR4 ; Load R0H with C
   ; <-- MOV32 complete
MOV32 R0H, *XAR4 ; Store the result
```

See also

- `ADDF32 RaH, #16FHi, RbH`
- `ADDF32 RaH, RbH, #16FHi`
- `ADDF32 RaH, RbH, RcH`
- `ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH`
- `MACF32 R3H, R2H, RdH, ReH, RfH`
- `MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH`
CMPF32 RaH, RbH — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than

### Operands
- **RaH**: Floating-point source register (R0H to R7H)
- **RbH**: Floating-point source register (R0H to R7H)

### Opcode
- **LSW**: 1110 0110 1001 0100
- **MSW**: 0000 0000 00bb baaa

### Description
Set ZF and NF flags on the result of RaH - RbH. The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE format offsetting the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:
- Negative zero will be treated as positive zero.
- A denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

### Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:
- If(RaH == RbH) \{ZF=1, NF=0\}
- If(RaH > RbH) \{ZF=0, NF=0\}
- If(RaH < RbH) \{ZF=0, NF=1\}

### Pipeline
This is a single-cycle instruction.

### Example
; Behavior of ZF and NF flags for different comparisons

```assembly
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
CMPF32 R1H, R0H ; ZF = 0, NF = 1
CMPF32 R0H, R1H ; ZF = 0, NF = 0
CMPF32 R0H, R0H ; ZF = 1, NF = 0

; Using the result of a compare for loop control
Loop:
  MOV32 R0H,*XAR4++ ; Load R0H
  MOV32 R1H,*XAR3++ ; Load R1H
  CMPF32 R1H, R0H ; Set/clear ZF and NF
  MOVST0 ZF, NF ; Copy ZF and NF to ST0 Z and N bits
  BF Loop, GT ; Loop if R1H > R0H
```

### See also
- CMPF32 RaH, #16FHi
- CMPF32 RaH, #0.0
- MAXF32 RaH, #16FHi
- MAXF32 RaH, RbH
- MINF32 RaH, #16FHi
- MINF32 RaH, RbH
CMPF32 RaH, #16FHi

32-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point source register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 0001 0III
MSW: IIII IIII IIII Iaaa

Description

Compare the value in RaH with the floating-point value represented by the immediate operand. Set the ZF and NF flags on (RaH - #16FHi:0).

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

- Negative zero will be treated as positive zero.
- Denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

If(RaH == #16FHi:0) {ZF=1, NF=0}
If(RaH > #16FHi:0) {ZF=0, NF=0}
If(RaH < #16FHi:0) {ZF=0, NF=1}

Pipeline

This is a single-cycle instruction

Example

; Behavior of ZF and NF flags for different comparisons
MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
MOVIZF32 R0H, #5.0 ; ROH = 5.0 (0x40A00000)
CMPF32 R1H, #-2.2 ; ZF = 0, NF = 0
CMPF32 ROH, #6.5 ; ZF = 0, NF = 1
CMPF32 ROH, #5.0 ; ZF = 1, NF = 0

; Using the result of a compare for loop control
Loop:
    MOV32 R1H,*XAR3++ ; Load R1H
    CMPF32 R1H, #2.0 ; Set/clear ZF and NF
    MOVST0 ZF, NF ; Copy ZF and NF to ST0 Z and N bits
    BF Loop, GT ; Loop if R1H > #2.0

See also

CMPF32 RaH, #0.0
CMPF32 RaH, RbH
MAXF32 RaH, #16FHi
CMPF32 RaH, #16FHi — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than

```
MAXF32 RaH, RbH
MINF32 RaH, #16FHi
MINF32 RaH, RbH
```
CMPF32 RaH, #0.0  32-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point source register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0.0</td>
<td>zero</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1010 0aaa

Description

Set the ZF and NF flags on (RaH - #0.0). The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

- Negative zero will be treated as positive zero.
- Denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

If(RaH == #0.0) {ZF=1, NF=0}
If(RaH > #0.0) {ZF=0, NF=0}
If(RaH < #0.0) {ZF=0, NF=1}

Pipeline

This is a single-cycle instruction.

Example

; Behavior of ZF and NF flags for different comparisons
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
MOVIZF32 R2H, #0.0 ; R2H = 0.0 (0x00000000)
CMPF32 R0H, #0.0 ; ZF = 0, NF = 0
CMPF32 R1H, #0.0 ; ZF = 0, NF = 1
CMPF32 R2H, #0.0 ; ZF = 0, NF = 0

; Using the result of a compare for loop control
Loop:
MOV32 R1H,*XAR3++ ; Load R1H
CMPF32 R1H, #0.0 ; Set/clear ZF and NF
MOVST0 ZF, NF ; Copy ZF and NF to ST0 Z and N bits
BF Loop, GT ; Loop if R1H > #0.0

See also

CMPF32 RaH, #0.0
CMPF32 RaH, #16FHi
MAXF32 RaH, #16FHi
MAXF32 RaH, RbH
MINF32 RaH, #16FHi
MINF32 RaH, RbH
EINVF32 RaH, RbH — 32-bit Floating-Point Reciprocal Approximation

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1001 0011
MSW: 0000 0000 00bb baaa

**Description**

This operation generates an estimate of 1/X in 32-bit floating-point format accurate to approximately 8 bits. This value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

\[
Y_e = \text{Estimate}(1/X);
Y_e = Y_e \times (2.0 - Y_e \times X)
\]

After two iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to the 32-bit floating-point format. On each iteration the mantissa bit accuracy approximately doubles. The EINVF32 operation will not generate a negative zero, DeNorm or NaN value.

RaH = Estimate of 1/RbH

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if EINVF32 generates an underflow condition.
- LVF = 1 if EINVF32 generates an overflow condition.

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

EINVF32 RaH, RbH ; 2p
NOP ; 1 cycle delay or non-conflicting instruction
NOP ; <-- EINVF32 completes, RaH updated

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.
Example

Calculate \( Y = A/B \). A fast division routine similar to that shown below can be found in the C28x FPU Fast RTS Library (SPRC664).

```assembly
MOVL XAR4, #A
MOV32 R0H, *XAR4 ; Load R0H with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
LCR DIV ; Calculate R0H = R0H / R1H
MOV32 *XAR4, R0H ;

....

DIV:
EINVF32 R2H, R1H ; R2H = Ye = Estimate(1/B)
CMPF32 R0H, #0.0 ; Check if A == 0
MPYF32 R3H, R2H, R1H ; R3H = Ye*B
NOP
SUBF32 R3H, #2.0, R3H ; R3H = 2.0 - Ye*B
NOP
MPYF32 R2H, R2H, R3H ; R2H = Ye = Ye*(2.0 - Ye*B)
NOP
MPYF32 R3H, R2H, R1H ; R3H = Ye*B
CMPF32 R1H, #0.0 ; Check if B == 0.0
SUBF32 R3H, #2.0, R3H ; R3H = 2.0 - Ye*B
NEGF32 R0H, R0H, EQ ; Fixes sign for A/0.0
MPYF32 R2H, R2H, R3H ; R2H = Ye = Ye*(2.0 - Ye*B)
NOP
MPYF32 R0H, R0H, R2H ; R0H = Y = A*Ye = A/B
LRETR
```

See also

EISQRTF32 RaH, RbH
EISQRTF32 RaH, RbH — 32-bit Floating-Point Square-Root Reciprocal Approximation

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1001 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

Description

This operation generates an estimate of $1/\sqrt{X}$ in 32-bit floating-point format accurate to approximately 8 bits. This value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

$$Ye = \text{Estimate}(1/\sqrt{X});$$
$$Ye = Ye*(1.5 - Ye*Ye*X/2.0)$$
$$Ye = Ye*(1.5 - Ye*Ye*X/2.0)$$

After 2 iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to the 32-bit floating-point format. On each iteration the mantissa bit accuracy approximately doubles. The EISQRTF32 operation will not generate a negative zero, DeNorm or NaN value.

$$RaH = \text{Estimate of } 1/\sqrt{\text{RbH}}$$

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- $LUF = 1$ if EISQRTF32 generates an underflow condition.
- $LVF = 1$ if EISQRTF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
EINVF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
      ; <-- EISQRTF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.
Example

Calculate the square root of $X$. A square-root routine similar to that shown below can be found in the C28x FPU Fast RTS Library (SPRC664).

```assembly
; $Y = \text{sqrt}(X)$
; $Ye = \text{Estimate}(1/\text{sqrt}(X))$
; $Ye = Ye*(1.5 - Ye*Ye*X*0.5)$
; $Y = X*Ye$

_sqrt:
; R0H = X on entry
EISQRTF32 R1H, R0H ; R1H = Ye = Estimate(1/sqrt(X))
MPYF32 R2H, R0H, #0.5 ; R2H = X*0.5
MPYF32 R3H, R1H, R1H ; R3H = Ye*Ye
NOP
MPYF32 R3H, R3H, R2H ; R3H = Ye*Ye*X*0.5
NOP
SUBF32 R3H, #1.5, R3H ; R3H = 1.5 - Ye*Ye*X*0.5
NOP
MPYF32 R1H, R1H, R3H ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
NOP
MPYF32 R3H, R1H, R2H ; R3H = Ye*X*0.5
NOP
MPYF32 R3H, R1H, R3H ; R3H = Ye*Ye*X*0.5
NOP
SUBF32 R3H, #1.5, R3H ; R3H = 1.5 - Ye*Ye*X*0.5
CMPF32 R0H, #0.0 ; Check if X == 0
MPYF32 R1H, R1H, R3H ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
NOP
MOV32 R1H, R0H, EQ ; If X is zero, change the Ye estimate to 0
MPYF32 R0H, R0H, R1H ; R0H = Y = X*Ye = sqrt(X)
LRETR
```

See also

EINVF32 RaH, RbH
F32TOI16 RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Integer

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

| LSW: 1110 0110 1000 1100 |
| MSW: 0000 0000 00bb baaa |

Description

Convert a 32-bit floating point value in RbH to a 16-bit integer and truncate. The result will be stored in RaH.

RaH(15:0) = F32TOI16(RbH)
RaH(31:16) = sign extension of RaH(15)

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

F32TOI16 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

| MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000) |
| F32TOI16 R1H, R0H ; R1H(15:0) = F32TOI16(R0H) |
| ; R1H(31:16) = Sign extension of R1H(15) |
| MOVIZF32 R2H, #-5.0 ; R2H = -5.0 (0xC0A00000) |
| ; <-- F32TOI16 complete, R1H(15:0) = 5 (0x0005) |
| ; R1H(31:16) = 0 (0x0000) |
| F32TOI16 R3H, R2H ; R3H(15:0) = F32TOI16(R2H) |
| ; R3H(31:16) = Sign extension of R3H(15) |
| NOP ; 1 Cycle delay for F32TOI16 to complete |
| ; <-- F32TOI16 complete, R3H(15:0) = -5 (0xFFFFB) |
| ; R3H(31:16) = (0xFFFF) |

See also

F32TOI16R RaH, RbH
F32TOU16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH
F32TOI16R RaH, RbH  

--- Convert 32-bit Floating-Point Value to 16-bit Integer and Round ---

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

| LSW       | 1110 0110 1000 1100 |
| MSW       | 1000 0000 00bb baaa |

**Description**

Convert the 32-bit floating point value in RbH to a 16-bit integer and round to the nearest even value. The result is stored in RaH.

RaH(15:0) = F32ToI16round(RbH)  
RaH(31:16) = sign extension of RaH(15)

**Flags**

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

F32TOI16R RaH, RbH  ; 2 pipeline cycles (2p)
NOP  ; 1 cycle delay or non-conflicting instruction
      ; <-- F32TOI16R completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

**Example**

```
MOVIZ R0H, #0x3FD9 ; R0H [31:16] = 0x3FD9
MO VXI R0H, #0x999A ; R0H [15:0] = 0x999A
                    ; R0H = 1.7 (0x3FD99999)
F32TOI16R R1H, R0H ; R1H(15:0) = F32TOI16Round(R0H)
                    ; R1H(31:16) = Sign extension of R1H(15)
MOV F32 R2H, #-1.7 ; R2H = -1.7 (0xBFDB9999)
                  ; <- F32TOI16R complete, R1H(15:0) = 2 (0x0002)
                  ; R1H(31:16) = 0 (0x0000)
F32TOI16R R3H, R2H ; R3H(15:0) = F32TOI16Round(R2H)
                    ; R3H(31:16) = Sign extension of R2H(15)
NOP  ; 1 cycle delay for F32TOI16R to complete
      ; <-- F32TOI16R complete, R1H(15:0) = -2 (0xFFFE)
      ; R1H(31:16) = (0xFFFF)
```

**See also**

- F32TOI16 RaH, RbH
- F32TOUI16 RaH, RbH
- F32TOUI16R RaH, RbH
- I16TOF32 RaH, RbH
- I16TOF32 RaH, mem16
- UI16TOF32 RaH, mem16
- UI16TOF32 RaH, RbH
F32TOI32 RaH, RbH — Convert 32-bit Floating-Point Value to 32-bit Integer

F32TOI32 RaH, RbH  Convert 32-bit Floating-Point Value to 32-bit Integer

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1000
MSW: 0000 0000 00bb baaa

Description
Convert the 32-bit floating-point value in RbH to a 32-bit integer value and truncate. Store the result in RaH.

RaH = F32TOI32(RbH)

Flags
This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline
This is a 2 pipeline cycle (2p) instruction. That is:

F32TOI32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- F32TOI32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVF32 R2H, #11204005.0 ; R2H = 11204005.0 (0x4B2AF5A5)
F32TOI32 R3H, R2H ; R3H = F32TOI32 (R2H)
MOVF32 R4H, #-11204005.0 ; R4H = -11204005.0 (0xCB2AF5A5)
    ; <-- F32TOI32 complete,
    ; R3H = 11204005 (0x00AAF5A5)
F32TOI32 R5H, R4H ; R5H = F32TOI32 (R4H)
NOP ; 1 Cycle delay for F32TOI32 to complete
    ; <-- F32TOI32 complete,
    ; R5H = -11204005 (0xFF550A5B)

See also
F32TOUI32 RaH, RbH
I32TOF32 RaH, RbH
I32TOF32 RaH, mem32
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32
F32TOUI16 RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1110
MSW: 0000 0000 00bb baaa

Description

Convert the 32-bit floating point value in RbH to an unsigned 16-bit integer value and truncate to zero. The result will be stored in RaH. To instead round the integer to the nearest even value use the F32TOUI16R instruction. The instruction will saturate the float to what can fit in 16-bit integer and then convert to 16-bit. For example 300000 will be saturated to 65535.

\[ RaH(15:0) = \text{F32ToUI16}(RbH) \quad RaH(31:16) = 0x0000 \]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

\[
\begin{align*}
\text{F32TOUI16} & \quad \text{RaH, RbH} \quad ; \text{2 pipeline cycles (2p)} \\
\text{NOP} & \quad ; \text{1 cycle delay or non-conflicting instruction} \\
\text{NOP} & \quad ; \langle\text{-- F32TOUI16 completes, RaH updated}\rangle \\
\end{align*}
\]

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

\[
\begin{align*}
\text{MOVIZF32} & \quad \text{R4H, #9.0} \quad ; \text{R4H} = 9.0 \quad (0x41100000) \\
\text{F32TOUI16} & \quad \text{R5H, R4H} \quad ; \text{R5H (15:0) = F32TOUI16 (R4H)} \\
& \quad ; \text{R5H (31:16) = 0x0000} \\
\text{MOVIZF32} & \quad \text{R6H, #-9.0} \quad ; \text{R6H} = -9.0 \quad (0xC1100000) \\
& \quad ; \langle\text{-- F32TOUI16 complete, R5H (15:0) = 9.0} \quad (0x0000)\rangle \\
& \quad ; \text{R5H (31:16) = 0.0} \quad (0x0000) \\
\text{F32TOUI16} & \quad \text{R7H, R6H} \quad ; \text{R7H (15:0) = F32TOUI16 (R6H)} \\
& \quad ; \text{R7H (31:16) = 0x0000} \\
\text{NOP} & \quad ; \text{1 Cycle delay for F32TOUI16 to complete} \\
& \quad ; \langle\text{-- F32TOUI16 complete, R7H (15:0) = 0.0} \quad (0x0000)\rangle \\
& \quad ; \text{R7H (31:16) = 0.0} \quad (0x0000) \\
\end{align*}
\]

See also

- F32TOI16 RaH, RbH
- F32TOUI16R RaH, RbH
- F32TOUI16R RaH, RbH
- I16TOF32 RaH, RbH
- I16TOF32 RaH, mem16
- UI16TOF32 RaH, mem16
- UI16TOF32 RaH, RbH
F32TOUI16R RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round

Operands

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1110
MSW: 1000 0000 00bb baaa

Description

Convert the 32-bit floating-point value in RbH to an unsigned 16-bit integer and round to the closest even value. The result will be stored in RaH. To instead truncate the converted value, use the F32TOI16 instruction. The instruction will saturate the float to what can fit in 16bit integer and then convert to 16bit. For example 300000 will be saturated to 65535.

RaH(15:0) = F32ToUI16round(RbH)
RaH(31:16) = 0x0000

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

F32TOUI16R RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- F32TOUI16R completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVIZ R5H, #0x412C ; R5H = 0x412C
MOVX1 R5H, #0xCCCD ; R5H = 0xCCCD
; R5H = 10.8 (0x412CCCCD)
F32TOUI16R R6H, R5H ; R6H (15:0) = F32TOUI16round (R5H)
; R6H (31:16) = 0x0000
MOVF32 R7H, #-10.8 ; R7H = -10.8 (0x00C12CCCCD)
; <-- F32TOUI16R complete,
; R6H (15:0) = 11.0 (0x0000B)
; R6H (31:16) = 0.0 (0x0000)
F32TOUI16R R0H, R7H ; R0H (15:0) = F32TOUI16round (R7H)
; R0H (31:16) = 0x0000
NOP ; 1 Cycle delay for F32TOUI16R to complete
; <-- F32TOUI16R complete,
; R0H (15:0) = 0.0 (0x0000)
; R0H (31:16) = 0.0 (0x0000)

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH
F32TOUI32 RaH, RbH  

Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1010
MSW: 0000 0000 00bb baaa

Description

Convert the 32-bit floating-point value in RbH to an unsigned 32-bit integer and store the result in RaH.

RaH = F32ToUI32(RbH)

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

F32TOUI32 RaH, RbH ; 2 pipeline cycles (2p)
NOP  ; 1 cycle delay or non-conflicting instruction
      ; <-- F32TOUI32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVIZF32 R6H, #12.5 ; R6H = 12.5 (0x41480000)
F32TOUI32 R7H, R6H ; R7H = F32TOUI32 (R6H)
MOVIZF32 R1H, #−6.5 ; R1H = −6.5 (0xC0D00000)
      ; <-- F32TOUI32 complete, R7H = 12.0 (0x0000000C)
F32TOUI32 R2H, R1H ; R2H = F32TOUI32 (R1H)
NOP  ; 1 Cycle delay for F32TOUI32 to complete
      ; <-- F32TOUI32 complete, R2H = 0.0 (0x00000000)

See also

F32TOI32 RaH, RbH
I32TOF32 RaH, RbH
I32TOF32 RaH, mem32
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32
FRACF32 RaH, RbH — Fractional Portion of a 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1111 0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

Description

Returns in RaH the fractional portion of the 32-bit floating-point value in RbH.

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
FRACF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- FRACF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZF32 R2H, #19.625 ; R2H = 19.625 (0x419D0000)
FRACF32 R3H, R2H ; R3H = FRACF32 (R2H)
NOP ; 1 cycle delay for FRACF32 to complete
; <-- FRACF32 complete, R3H = 0.625 (0x3F200000)
```

See also
I16TOF32 RaH, RbH — Convert 16-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1101
MSW: 0000 0000 00bb baaa

Description

Convert the 16-bit signed integer in RbH to a 32-bit floating point value and store the result in RaH.

RaH = I16ToF32 RbH

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

I16TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- I16TOF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVIZ R0H, #0x0000 ; R0H[31:16] = 0.0 (0x0000)
MOVX1 R0H, #0x0004 ; R0H[15:0] = 4.0 (0x0004)
I16TOF32 R1H, R0H ; R1H - I16TOF32 (R0H)
MOVIZ R2H, #0x0000 ; R2H[31:16] = 0.0 (0x0000)
; <--I16TOF32 complete, R1H = 4.0 (0x40800000)
MOVX1 R2H, #0xFFFC ; R2H[15:0] = -4.0 (0xFFFC) I16TOF32 R3H, R2H ; R3H = I16TOF32 (R2H)
NOP ; 1 Cycle delay for I16TOF32 to complete
; <-- I16TOF32 complete, R3H = -4.0 (0xC0800000)

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOU16 RaH, RbH
F32TOU16R RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH
I16TOF32 RaH, mem16 — Convert 16-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem316</td>
<td>16-bit source memory location to be converted</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1100 1000
MSW: 0000 0aaa mem16

Description

Convert the 16-bit signed integer indicated by the mem16 pointer to a 32-bit floating-point value and store the result in RaH.

RaH = I16ToF32[mem16]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>Ni</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
I16TOF32 RaH, mem16 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- I16TOF32 completes, RaH updated
NOP
```

Example

```
MOVW DP, #0x0280 ; DP = 0x0280
MOV @0, #0x0004 ; [0x00A000] = 4.0 (0x0004)
I16TOF32 R0H, @0 ; R0H = I16TOF32 [0x00A000]
MOV @1, #0xFFFC ; [0x00A001] = -4.0 (0xFFFC)
    ; <--I16TOF32 complete, R0H = 4.0 (0x40800000)
I16TOF32 R1H, @1 ; R1H = I16TOF32 [0x00A001]
NOP ; 1 cycle delay for I16TOF32 to complete
    ; <-- I16TOF32 complete, R1H = -4.0 (0xC0800000)
```

See also

- F32TOI16 RaH, RbH
- F32TOI16R RaH, RbH
- F32TOUI16 RaH, RbH
- F32TOUI16R RaH, RbH
- I16TOF32 RaH, RbH
- UI16TOF32 RaH, mem16
- UI16TOF32 RaH, RbH
I32TOF32 RaH, mem32  

Convert 32-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>32-bit source for the MOV32 operation. mem32 means that the operation can only address memory using any of the direct or indirect addressing modes supported by the C28x CPU</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 1000
MSW: 0000 0aaa mem32

Description

Convert the 32-bit signed integer indicated by the mem32 pointer to a 32-bit floating point value and store the result in RaH.

RaH = I32ToF32[mem32]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

I32TOF32 RaH, mem32 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
NOP ; <-- I32TOF32 completes, RaH updated

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

MOVW DP, #0x0280 ; DP = 0x0280
MOV @0, #0x1111 ; [0x000A000] = 4369 (0x1111)
MOV @1, #0x1111 ; [0x000A001] = 4369 (0x1111)
; Value of the 32 bit signed integer present in
; 0x00A001 and 0x00A000 is +286331153 (0x11111111)
I32TOF32 R1H, @0 ; R1H = I32TOF32 (0x11111111)
NOP ; 1 Cycle delay for I32TOF32 to complete
; <-- I32TOF32 complete, R1H = 286331153 (0x4D888888)

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, RbH
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32
**I32TOF32 RaH, RbH — Convert 32-bit Integer to 32-bit Floating-Point Value**

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

```
LSW: 1110 0110 1000 1001
MSW: 0000 0000 00bb baaa
```

**Description**

Convert the signed 32-bit integer in RbH to a 32-bit floating-point value and store the result in RaH.

RaH = I32ToF32(RbH)

**Flags**

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

```
I32TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
I32TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

**Example**

```
MOVIZ R2H, #0x1111 ; R2H[31:16] = 4369 (0x1111)
MOVXI R2H, #0x1111 ; R2H[15:0] = 4369 (0x1111)
; Value of the 32 bit signed integer present
; in R2H is +286331153 (0x11111111)
I32TOF32 R3H, R2H ; R3H = I32TOF32 (R2H)
NOP ; 1 Cycle delay for I32TOF32 to complete
; <-- I32TOF32 complete, R3H = 286331153 (0x4D888888)
```

**See also**

- F32TOI32 RaH, RbH
- F32TOUI32 RaH, RbH
- I32TOF32 RaH, mem32
- UI32TOF32 RaH, RbH
- UI32TOF32 RaH, mem32
MACF32 R3H, R2H, RdH, ReH, RfH  32-bit Floating-Point Multiply with Parallel Add

Operands

This instruction is an alias for the parallel multiply and add instruction. The operands are translated by the assembler such that the instruction becomes:

MPYF32 RdH, ReH, RfH
|| ADDF32 R3H, R3H, R2H

- **R3H** — floating-point destination and source register for the ADDF32
- **R2H** — Floating-point source register for the ADDF32 operation (R0H to R7H)
- **RdH** — Floating-point destination register for MPYF32 operation (R0H to R7H)
- **ReH** — Floating-point source register for MPYF32 operation (R0H to R7H)
- **RfH** — Floating-point source register for MPYF32 operation (R0H to R7H)

Opcode

LSW: 1110 0111 0100 00ff
MSW: feee dddc ccbf baaa

Description

This instruction is an alias for the parallel multiply and add, MACF32 || ADDF32, instruction.

RdH = ReH * RfH
R3H = R3H + R2H

Restrictions

The destination register for the MPYF32 and the ADDF32 must be unique. That is, RdH cannot be R3H.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:
- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline

Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

MPYF32 RaH, RbH, Rch ; 2 pipeline cycles (2p)
|| ADDF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction

NOP ; <-- MPYF32, ADDF32 complete, RaH, RdH updated

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.
Example

; Perform 5 multiply and accumulate operations:

; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3

; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0

MPYF32 R2H, R0H, R1H ; in parallel R0H = X1
||

MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1
; R2H = A = X0 * Y0

MPYF32 R3H, R0H, R1H ; in parallel R0H = X2
||

MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2
; R3H = B = X1 * Y1
; R2H = C = X2 * Y2

MACF32 R3H, R2H, R2H, R0H, R1H ; in parallel R0H = X3
||

MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3
; R3H = A + B
; R2H = C = X2 * Y2

MACF32 R3H, R2H, R2H, R0H, R1H ; in parallel R0H = X4
||

MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

; The next MACF32 is an alias for
; MPYF32 || ADDF32

MACF32 R3H, R2H, R2H, R0H, R1H ; in parallel R3H = (A + B + C) + D
NOP ; wait for MPYF32 || ADDF32 to complete

ADDF32 R3H, R3H, R2H ; R3H = (A + B + C + D) + E
NOP ; wait for ADDF32 to complete
MOV32 @Result, R3H ; store the result

See also

MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32

32-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H</td>
<td>32-bit floating-point destination/source register R3H for the add operation</td>
</tr>
<tr>
<td>R2H</td>
<td>32-bit floating-point source register R2H for the add operation</td>
</tr>
<tr>
<td>RdH</td>
<td>32-bit floating-point destination register (R0H to R7H) for the multiply operation</td>
</tr>
<tr>
<td>ReH</td>
<td>32-bit floating-point source register (R0H to R7H) for the multiply operation</td>
</tr>
<tr>
<td>RfH</td>
<td>32-bit floating-point source register (R0H to R7H) for the multiply operation</td>
</tr>
<tr>
<td>RaH</td>
<td>32-bit floating-point destination register for the MOV32 operation (R0H to R7H).</td>
</tr>
<tr>
<td>mem32</td>
<td>32-bit source for the MOV32 operation</td>
</tr>
</tbody>
</table>

Opcode

| LSW       | 1110 0011 0011 fffe |
| MSW       | eedd daaa mem32 |

Description

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF32.

R3H = R3H + R2H,
RdH = ReH * RfH,
RaH = [mem32]

Restrictions

The destination registers for the MACF32 and the MOV32 must be unique. That is, RaH cannot be R3H and RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MACF32 (add or multiply) generates an underflow condition.
- LVF = 1 if MACF32 (add or multiply) generates an overflow condition.

MOV32 sets the NF, ZF, NI and ZI flags as follows:

- NF = RaH(31);
- ZF = 0;
- if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
- NI = RaH(31);
- ZI = 0;
- if(RaH(31:0) == 0) ZI = 1;

Pipeline

The MACF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

```
MACF32 R3H, R2H, RdH, ReH, RfH ; 2 pipeline cycles (2p)

|| MOV32 RaH, mem32 ; 1 cycle
  ;<--- MOV32 completes, RaH updated

NOP ; 1 cycle delay for MACF32

NOP ;<--- MACF32 completes, R3H, RdH updated

NOP
```

Any instruction in the delay slot for this version of MACF32 must not use R3H or RdH as a destination register or R3H or RdH as a source operand.
Example

; Perform 5 multiply and accumulate operations:

; 1st multiply: A = X0 \times Y0
; 2nd multiply: B = X1 \times Y1
; 3rd multiply: C = X2 \times Y2
; 4th multiply: D = X3 \times Y3
; 5th multiply: E = X3 \times Y3

; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0

MPYF32 R2H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1

MPYF32 R3H, R0H, R1H ; In parallel R0H = X2
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2

MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3

MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X4
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

MPYF32 R2H, R0H, R1H ; in parallel R3H = (A + B) + C + D
|| ADDF32 R3H, R3H, R2H
NOP ; Wait for MPYF32 || ADDF32 to complete

ADDF32 R3H, R3H, R2H ; R3H = (A + B + C + D) + E
NOP ; Wait for ADDF32 to complete

MOV32 @Result, R3H ; Store the result

See also
MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MACF32 R7H, R3H, mem32, *XAR7++ 32-bit Floating-Point Multiply and Accumulate

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7H</td>
<td>Floating-point destination register</td>
</tr>
<tr>
<td>R3H</td>
<td>Floating-point destination register</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit source location</td>
</tr>
<tr>
<td>*XAR7++</td>
<td>32-bit location pointed to by auxiliary register 7, XAR7 is post incremented.</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 0101 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0001 1111 mem32</td>
</tr>
</tbody>
</table>

Description

Perform a multiply and accumulate operation. When used as a standalone operation, the MACF32 will perform a single multiply as shown below:

Cycle 1: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]

This instruction is the only floating-point instruction that can be repeated using the single repeat instruction (RPT |). When repeated, the destination of the accumulate will alternate between R3H and R7H on each cycle and R2H and R6H are used as temporary storage for each multiply.

Cycle 1: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
Cycle 2: R7H = R7H + R6H, R6H = [mem32] * [XAR7++]
Cycle 3: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
Cycle 4: R7H = R7H + R6H, R6H = [mem32] * [XAR7++]

etc...

Restrictions

R2H and R6H will be used as temporary storage by this instruction.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MACF32 generates an underflow condition.
- LVF = 1 if MACF32 generates an overflow condition.

Pipeline

When repeated the MACF32 takes 3 + N cycles where N is the number of times the instruction is repeated. When repeated, this instruction has the following pipeline restrictions:

```
<instruction1>
<instruction2>
RPT #(N-1)
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
<instruction3>
```

- No restriction
- Cannot be a 2p instruction that writes to R2H, R3H, R6H or R7H
- Execute N times, where N is even
- No restrictions.
- Can read R2H, R3H, R6H and R7H
MACF32 can also be used standalone. In this case, the instruction takes 2 cycles and the following pipeline restrictions apply:

```
<instruction1> ; No restriction
<instruction2> ; Cannot be a 2p instruction that writes
to R2H, R3H, R6H or R7H
MACF32 R7H, R3H, *XAR6, *XAR7 ; R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
; <--
R2H and R3H are valid (note: no delay required)
NOP
```

Example

```
ZERO R2H ; Zero the accumulation registers
ZERO R3H ; and temporary multiply storage
ZERO R6H
ZERO R7H
RPT #3 ; Repeat MACF32 N+1 (4) times
| MACF32 R7H, R3H, *XAR6++, *XAR7++
| ADDF32 R7H, R7H, R3H ; Final accumulate
| NOP ; <-- ADDF32 completes, R7H valid
| NOP
```

Cascading of RPT || MACF32 is allowed as long as the first and subsequent counts are even. Cascading is useful for creating interruptible windows so that interrupts are not delayed too long by the RPT instruction. For example:

```
ZERO R2H ; Zero the accumulation registers
ZERO R3H ; and temporary multiply storage
ZERO R6H
ZERO R7H
RPT #3 ; Execute MACF32 N+1 (4) times
| MACF32 R7H, R3H, *XAR6++, *XAR7++ RPT #5 ; Execute MACF32 N+1 (6) times
| | MACF32 R7H, R3H, *XAR6++, *XAR7++ RPT #N ; Repeat MACF32 N+1 times where N+1
| is even
| | MACF32 R7H, R3H, *XAR6++, *XAR7++
| | ADDF32 R7H, R7H, R3H ; Final accumulate
| | NOP ; <-- ADDF32 completes, R7H valid
```

See also

MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Add

Operands
This instruction is an alias for the parallel multiply and add instruction. The operands are translated by the assembler such that the instruction becomes:

\[ \text{MPYF32 RdH, RaH, RbH || ADDF32 R7H, R7H, R6H} \]

- **R7H**: floating-point destination and source register for the ADDF32
- **R6H**: Floating-point source register for the ADDF32 operation (R0H to R7H)
- **RdH**: Floating-point destination register for MPYF32 operation (R0H to R7H)
- **RdH** cannot be R3H
- **ReH**: Floating-point source register for MPYF32 operation (R0H to R7H)
- **RfH**: Floating-point source register for MPYF32 operation (R0H to R7H)

Opcode
- **LSW**: 1110 0111 0100 00ff
- **MSW**: feee dddc cbbb baaa

Description
This instruction is an alias for the parallel multiply and add, MACF32 || ADDF32, instruction.

- \( \text{RdH} = \text{RaH} \times \text{RbH} \)
- \( \text{R7H} = \text{R6H} + \text{R6H} \)

Restrictions
The destination register for the MPYF32 and the ADDF32 must be unique. That is, RdH cannot be R7H.

Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- \( \text{LUF} = 1 \) if MPYF32 or ADDF32 generates an underflow condition.
- \( \text{LVF} = 1 \) if MPYF32 or ADDF32 generates an overflow condition.

Pipeline
Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

\[ \text{MPYF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)} \]
\[ \text{|| ADDF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)} \]
\[ \text{NOP ; 1 cycle delay or non-conflicting instruction} \]
\[ ; \text{<-- MPYF32, ADDF32 complete, RaH, RdH updated} \]

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.
Example

; Perform 5 multiply and accumulate operations:

; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3

; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0
MPYF32 R6H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1
; R7H = B = X1 * Y1
MPYF32 R7H, R0H, R1H ; In parallel R0H = X2
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2
; R7H = A + B
; R6H = C = X2 * Y2
MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3
; R7H = (A + B) + C
; R6H = D = X3 * Y3
MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

; Next MACF32 is an alias for
; MPYF32 || ADDF32
MACF32 R7H, R6H, R6H, R0H, R1H ; R6H = E = X4 * Y4
|| in parallel R7H = (A + B + C) + D
NOP ; Wait for MPYF32 || ADDF32 to complete
ADDF32 R7H, R7H, R6H ; R7H = (A + B + C + D) + E
NOP ; Wait for ADDF32 to complete
MOV32 @Result, R7H ; Store the result

See also

MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, Rch || ADDF32 RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH | MOV32 RaH, mem32  32-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7H</td>
<td>Floating-point destination/source register R7H for the add operation</td>
</tr>
<tr>
<td>R6H</td>
<td>Floating-point source register R6H for the add operation</td>
</tr>
<tr>
<td>RdH</td>
<td>Floating-point destination register (R0H to R7H) for the multiply operation. RdH cannot be the same register as RaH.</td>
</tr>
<tr>
<td>ReH</td>
<td>Floating-point source register (R0H to R7H) for the multiply operation</td>
</tr>
<tr>
<td>RfH</td>
<td>Floating-point source register (R0H to R7H) for the multiply operation</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point destination register for the MOV32 operation (R0H to R7H). RaH cannot be R3H or the same as RdH.</td>
</tr>
<tr>
<td>mem32</td>
<td>32-bit source for the MOV32 operation</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0011 1100 fffe</td>
<td>eedd daaa mem32</td>
</tr>
</tbody>
</table>

Description

Multiply/accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF32.

\[
R7H = R7H + R6H \\
RdH = ReH \times RfH, \\
RaH = [\text{mem32}] 
\]

Restrictions

The destination registers for the MACF32 and the MOV32 must be unique. That is, RaH cannot be R7H and RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- \( LUF = 1 \) if MACF32 (add or multiply) generates an underflow condition.
- \( LVF = 1 \) if MACF32 (add or multiply) generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

\[
\begin{align*}
NF &= RaH(31); \\
ZF &= 0; \\
& \text{if}(RaH(30:23) == 0) \ {ZF} = 1; \\
NF &= 0; \ {NI} = RaH(31); \\
ZI &= 0; \\
& \text{if}(RaH(31:0) == 0) \ {ZI} = 1;
\end{align*}
\]

Pipeline

The MACF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

\[
\begin{align*}
\text{MACF32} &\ \ R7H, \ R6H, \ RdH, \ ReH, \ RfH \ ; \ 2 \ \text{pipeline cycles (2p)} \\
\ || \ MOV32 &\ \ RaH, \ \ \text{mem32} \\
\ || \ NO &\ \ \text{NOP} \\
\ || \ NO &\ \ \text{NOP}
\end{align*}
\]

\<-- \text{MOV32 completes, RaH updated} \\
\<-- \text{MACF32 completes, R7H, RdH updated}
Example

Perform 5 multiply and accumulate operations:

1. 1st multiply: \( A = X_0 \times Y_0 \)
2. 2nd multiply: \( B = X_1 \times Y_1 \)
3. 3rd multiply: \( C = X_2 \times Y_2 \)
4. 4th multiply: \( D = X_3 \times Y_3 \)
5. 5th multiply: \( E = X_3 \times Y_3 \)

Result = \( A + B + C + D + E \)

\[
\begin{align*}
\text{MOV32} & \text{ R0H, *XAR4++} ; \quad \text{R0H} = X_0 \\
\text{MOV32} & \text{ R1H, *XAR5++} ; \quad \text{R1H} = Y_0 \\
\text{MPYF32} & \text{ R6H, R0H, R1H} ; \quad \text{In parallel R0H} = X_1 \\
\text{MACF32} & \text{ R7H, R6H, R6H, R6H, R1H} ; \quad \text{In parallel R0H} = X_2 \\
\text{MACF32} & \text{ R7H, R6H, R6H, R6H, R1H} ; \quad \text{In parallel R0H} = X_3 \\
\text{MACF32} & \text{ R7H, R6H, R6H, R6H, R1H} ; \quad \text{In parallel R0H} = X_4 \\
\text{MPYF32} & \text{ R6H, R0H, R1H} ; \quad \text{In parallel R7H} = (A + B + C) + D \\
\text{ADDF32} & \text{ R7H, R7H, R6H} ; \quad \text{Wait for MPYF32 || ADDF32 to complete} \\
\text{ADDF32} & \text{ R7H, R7H, R6H} ; \quad \text{R7H} = (A + B + C + D) + E \\
\text{NOP} & ; \quad \text{Wait for ADDF32 to complete} \\
\text{MOV32} & \text{ @Result, R7H} ; \quad \text{Store the result}
\end{align*}
\]

See also

\[
\begin{align*}
\text{MACF32} & \text{ R7H, R3H, mem32, *XAR7++} \\
\text{MACF32} & \text{ R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32} \\
\text{MPYF32} & \text{ RaH, RbH, RcH || ADDF32 RdH, ReH, RfH}
\end{align*}
\]
MAXF32 RaH, RbH  

32-bit Floating-Point Maximum

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1001 0110  
MSW: 0000 0000 00bb baaa

**Description**

if(RaH < RbH) RaH = RbH

Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

if(RaH == RbH) ZF=1, NF=0
if(RaH > RbH) ZF=0, NF=0
if(RaH < RbH) ZF=0, NF=1

**Pipeline**

This is a single-cycle instruction.

**Example**

MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #2.0 ; R1H = -2.0 (0xC0000000)
MOVIZF32 R2H, #1.5 ; R2H = -1.5 (0xBFC00000)
MAXF32 R2H, R1H ; R2H = -1.5, ZF = NF = 0
MAXF32 R1H, R2H ; R1H = -1.5, ZF = 0, NF = 1
MAXF32 R2H, R0H ; R2H = 5.0, ZF = 0, NF = 1
MAXF32 R0H, R2H ; R2H = 5.0, ZF = 1, NF = 0

**See also**

CMPF32 RaH, RbH  
CMPF32 RaH, #16FHi  
CMPF32 RaH, #0.0  
MAXF32 RaH, RbH || MOV32 RcH, RdH  
MAXF32 RaH, #16FHi  
MINF32 RaH, RbH  
MINF32 RaH, #16FHi
MAXF32 RaH, #16FHi — 32-bit Floating-Point Maximum

Operands

GaH floating-point source/destination register (R0H to R7H)
#16FHi A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

LSW: 1110 1000 0010 0III
MSW: IIII IIII IIII Iaaa

Description

Compare RaH with the floating-point value represented by the immediate operand. If the immediate value is larger, then load it into RaH.

\[
\text{if}(\text{RaH} < \#16FHi:0) \text{ RaH} = \#16FHi:0
\]

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

\[
\text{if}(\text{RaH} == \#16FHi:0) \{ZF=1, \text{NF}=0\}
\]

\[
\text{if}(\text{RaH} > \#16FHi:0) \{ZF=0, \text{NF}=0\}
\]

\[
\text{if}(\text{RaH} < \#16FHi:0) \{ZF=0, \text{NF}=1\}
\]

Pipeline

This is a single-cycle instruction.

Example

| MOVIZF32 | R0H, #5.0 ; R0H = 5.0 (0x40A00000) |
| MOVIZF32 | R1H, #4.0 ; R1H = 4.0 (0x40800000) |
| MOVIZF32 | R2H, #1.5 ; R2H = -1.5 (0xBFC00000) |
| MAXF32   | R0H, #5.5 ; R0H = 5.5, ZF = 0, NF = 1 |
| MAXF32   | R1H, #2.5 ; R1H = 4.0, ZF = 0, NF = 0 |
| MAXF32   | R2H, #1.0 ; R2H = -1.0, ZF = 0, NF = 1 |
| MAXF32   | R2H, #1.0 ; R2H = -1.5, ZF = 1, NF = 0 |

See also

MAXF32 RaH, RbH
MAXF32 RaH, RbH || MOV32 RcH, RdH
MINF32 RaH, RbH
MINF32 RaH, #16FHi
MAXF32 RaH, RbH | MOV32 RcH, RdH  32-bit Floating-Point Maximum with Parallel Move

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register for the MAXF32 operation (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register for the MAXF32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>Floating-point destination register for the MOV32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>RdH</td>
<td>Floating-point source register for the MOV32 operation (R0H to R7H)</td>
</tr>
</tbody>
</table>

RaH cannot be the same register as RbH and RcH cannot be the same register as RaH.

Opcode

LSW: 1110 0110 1001 1100
MSW: 0000 dddc cccb baaa

Description

If RaH is less than RbH, then load RaH with RbH. Thus RaH will always have the maximum value. If RaH is less than RbH, then, in parallel, also load RcH with the contents of RdH.

```
if(RaH < RbH) { RaH = RbH; RcH = RdH; }
```

The MAXF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Restrictions

The destination register for the MAXF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RbH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == RbH) { ZF=1, NF=0; }
if(RaH > RbH) { ZF=0, NF=0; }
if(RaH < RbH) { ZF=0, NF=1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #4.0 ; R1H = 4.0 (0x40800000)
MOVIZF32 R2H, #1.5 ; R2H = -1.5 (0xBFCD0000)
MOVIZF32 R3H, #2.0 ; R3H = -2.0 (0xC0000000)
MAXF32 R0H, R1H ; R0H = 5.0, R3H = -1.5, ZF = 0, NF = 0
|| MOV32 R3H, R2H
MAXF32 R1H, R0H ; R1H = 5.0, R3H = -1.5, ZF = 0, NF = 1
|| MOV32 R3H, R2H
MAXF32 R0H, R1H ; R0H = 5.0, R2H = -1.5, ZF = 1, NF = 0
|| MOV32 R2H, R1H
```

See also

MAXF32 RaH, RbH
MAXF32 RaH, #16FHi
MINF32 RaH, RbH — 32-bit Floating-Point Minimum

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point source/destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0110 1001 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

**Description**

if(RaH > RbH) RaH = RbH

Special cases for the output from the MINF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

if(RaH == RbH) {ZF=1, NF=0}  
if(RaH > RbH) {ZF=0, NF=0}  
if(RaH < RbH) {ZF=0, NF=1}

**Pipeline**

This is a single-cycle instruction.

**Example**

MOVIZF32 ROH, #5.0 ; ROH = 5.0 (0x40A00000)  
MOVIZF32 RH1, #4.0 ; RH1 = 4.0 (0x40800000)  
MOVIZF32 RH2, #-1.5 ; RH2 = -1.5 (0x8FC00000)  
MINF32 ROH, RH1 ; ROH = 4.0, ZF = 0, NF = 0  
MINF32 RH1, RH2 ; RH1 = -1.5, ZF = 0, NF = 0  
MINF32 RH2, RH1 ; RH2 = -1.5, ZF = 1, NF = 0  
MINF32 RH1, ROH ; RH1 = -1.5, ZF = 0, NF = 1

**See also**

MAXF32 RaH, RbH  
MAXF32 RaH, #16FHi  
MINF32 RaH, #16FHi  
MINF32 RaH, RbH || MOV32 RcH, RdH
MINF32 RaH, #16FHi  

**32-bit Floating-Point Minimum**

**Operands**

- **RaH**: floating-point source/destination register (R0H to R7H)
- **#16FHi**: A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

**Opcode**

- **LSW**: 1110 1000 0011 0III
- **MSW**: IIII IIII IIII Iaaa

**Description**

Compare RaH with the floating-point value represented by the immediate operand. If the immediate value is smaller, then load it into RaH.

```
if(RaH > #16FHi:0) RaH = #16FHi:0
```

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F800000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as # -1.5 or #0xBFC0.

Special cases for the output from the MINF32 operation:
- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == #16FHi:0) {ZF=1, NF=0}
if(RaH > #16FHi:0) {ZF=0, NF=0}
if(RaH < #16FHi:0) {ZF=0, NF=1}
```

**Pipeline**

This is a single-cycle instruction.

**Example**

- MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
- MOVIZF32 R1H, #4.0 ; R1H = 4.0 (0x40800000)
- MOVIZF32 R2H, #-1.5 ; R2H = -1.5 (0x0BFC00000)
- MINF32 R0H, #5.5 ; R0H = 5.0, ZF = 0, NF = 1
- MINF32 R1H, #2.5 ; R1H = 2.5, ZF = 0, NF = 0
- MINF32 R2H, #-1.0 ; R2H = -1.5, ZF = 0, NF = 1
- MINF32 R2H, #1.5 ; R2H = -1.5, ZF = 1, NF = 0

**See also**

- MAXF32 RaH, #16FHi
- MAXF32 RaH, RbH
- MINF32 RaH, RbH
- MINF32 RaH, RbH || MOV32 RcH, RdH
MINF32 RaH, RbH | MOV32 RcH, RdH — 32-bit Floating-Point Minimum with Parallel Move

Operands

RaH floating-point source/destination register for the MIN32 operation (R0H to R7H)
RaH cannot be the same register as RcH

RbH Floating-point source register for the MIN32 operation (R0H to R7H)

RcH Floating-point destination register for the MOV32 operation (R0H to R7H)
RcH cannot be the same register as RaH

RdH Floating-point source register for the MOV32 operation (R0H to R7H)

Opcode

LSW: 1110 0110 1001 1101
MSW: 0000 dddc cbbb baaa

Description

if(RaH > RbH) { RaH = RbH; RcH = RdH; }

Special cases for the output from the MINF32 operation:

• NaN output will be converted to infinity
• A denormalized output will be converted to positive zero.

Restrictions

The destination register for the MINF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RcH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

• if(RaH == RbH) {ZF=1, NF=0}
• if(RaH > RbH) {ZF=0, NF=0}
• if(RaH < RbH) {ZF=0, NF=1}

Pipeline

This is a single-cycle instruction.

Example

```assembly
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #4.0 ; R1H = 4.0 (0x40800000)
MOVIZF32 R2H, #1.5 ; R2H = 1.5 (0x4BC00000)
MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x4C000000)
MINF32 R0H, R1H ; R0H = 4.0, R3H = -1.5, ZF = 0, NF = 0
|| MOV32 R3H, R2H
MINF32 R1H, R0H ; R1H = 4.0, R3H = -1.5, ZF = 1, NF = 0
|| MOV32 R3H, R2H
MINF32 R2H, R1H ; R2H = -1.5, R1H = 4.0, ZF = 1, NF = 1
|| MOV32 R1H, R3H
```

See also

MINF32 RaH, RbH
MINF32 RaH, #16FHi
MOV16 mem16, RaH — Move 16-bit Floating-Point Register Contents to Memory

Operands

<table>
<thead>
<tr>
<th>mem16</th>
<th>points to the 16-bit destination memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0001 0011
MSW: 0000 0aaa mem16

Description

Move 16-bit value from the lower 16-bits of the floating-point register (RaH[15:0]) to the location pointed to by mem16.

\[ \text{[mem16]} = \text{RaH}[15:0] \]

Flags

No flags STF flags are affected.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.

Example

MOVW DP, #0x02C0 ; DP = 0x02C0
MOVXI R4H, #0x0003 ; R4H = 3.0 (0x0003)
MOV16 @0, R4H ; [0x00B000] = 3.0 (0x0003)

See also

MOVIZ RaH, #16FHiHex
MOVIZF32 RaH, #16FHi
MOVXI RaH, #16FLoHex
MOV32 *(0:16bitAddr), loc32 — Move the Contents of loc32 to Memory

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:16bitAddr</td>
<td>16-bit immediate address, zero extended</td>
</tr>
<tr>
<td>loc32</td>
<td>32-bit source location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1101  loc32
MSW:  I IIII IIII IIII

Description

Move the 32-bit value in loc32 to the memory location addressed by 0:16bitAddr. The EALLOW bit in the ST1 register is ignored by this operation.

\[0:16bitAddr] = \{loc32\]

Flags

This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Modified</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a two-cycle instruction.

Example

MOVIZ  R5H, #0x1234 ; R5H[31:16] = 0x1234
MOVXI  R5H, #0xABCD ; R5H[15:0] = 0xABCD
NOP ; 1 Alignment Cycle
MOV32 ACC, R5H ; ACC = 0x1234ABCD
MOV32 *(0xA000), @ACC ; [0x00A000] = ACC NOP
; 1 Cycle delay for MOV32 to complete
; \(---> MOV32 *(0:16bitAddr), loc32 complete,
; \; [0x00A000] = 0xABCD, [0x00A001] = 0x1234

See also

MOV32 mem32, RaH
MOV32 mem32, STF
MOV32 loc32, *(0:16bitAddr)
MOV32 ACC, RaH  

Move 32-bit Floating-Point Register Contents to ACC

**Operands**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>28x accumulator</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

- LSW: 1011 1111 loc32
- MSW: IIII IIII IIII IIII

**Description**

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

ACC = RaH

**Flags**

No STF flags are affected.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Z and N flag in status register zero (ST0) of the 28x CPU are affected.

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```assembly
MINF32 R0H, R1H ; Single-cycle instruction
NOP ; 1 alignment cycle
MOV32 @ACC, R0H ; Copy R0H to ACC
NOP ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```assembly
ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP ; 1 cycle delay for ADDF32 to complete
; <-- ADDF32 completes, R2H is valid
NOP ; 1 alignment cycle MOV32 ACC, R2H
; copy R2H into ACC, takes 2 cycles
; <-- MOV32 completes, ACC is valid
NOP ; Any instruction
```

**Example**

```assembly
ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP ; 1 cycle delay for ADDF32 to complete
; <-- ADDF32 completes, R2H is valid
NOP ; 1 alignment cycle
MOV32 ACC, R2H ; copy R2H into ACC, takes 2 cycles
; <-- MOV32 completes, ACC is valid
NOP ; Any instruction
MOV1ZF32 R0H, #2.5 ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H ; Delay for conversion instruction
; <-- Conversion complete, R0H valid
NOP ; Alignment cycle
MOV32 P, R0H ; P = 2 = 0x00000002
```

**See also**

- MOV32 P, RaH
- MOV32 XARn, RaH
- MOV32 XT, RaH
MOV32 loc32, *(0:16bitAddr) — Move 32-bit Value from Memory to loc32

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc32</td>
<td>destination location</td>
</tr>
<tr>
<td>0:16bitAddr</td>
<td>16-bit address of the 32-bit source value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1111 loc32  
MSW: IIII IIII IIII IIII

Description

Copy the 32-bit value referenced by 0:16bitAddr to the location indicated by loc32.

[loc32] = [0:16bitAddr]

Flags

No STF flags are affected. If loc32 is the ACC register, then the Z and N flag in status register zero (ST0) of the 28x CPU are affected.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 cycle instruction.

Example

MOVW DP, #0x0300 ; DP = 0x0300  
MOV @0, #0xFFFF ; [0x00C000] = 0xFFFF;  
MOV @1, #0x1111 ; [0x00C001] = 0x1111;  
MOV32 @ACC, *(0xC000) ; AL = [0x00C000], AH = [0x00C001]  
NOP ; 1 Cycle delay for MOV32 to complete  
; <-- MOV32 complete, AL = 0xFFFF, AH = 0x1111

See also

MOV32 RaH, mem32{, CNDF}  
MOV32 *(0:16bitAddr), loc32  
MOV32 STF, mem32  
MOVD32 RaH, mem32
MOV32 mem32, RaH — Move 32-bit Floating-Point Register Contents to Memory

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>points to the 32-bit destination memory</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 0011
MSW: 0000 0aaa mem32

Description

Move from memory to STF.

[mem32] = RaH

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

Pipeline

This is a single-cycle instruction.

Example

; Perform 5 multiply and accumulate operations:
; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3
;
; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0

| || MPYF32 R6H, R0H, R1H ; In parallel R0H = X1
MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1

| || MPYF32 R7H, R0H, R1H ; In parallel R0H = X2
MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2

; R7H = A + B
; R6H = C + X2 * Y2
| || MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3
MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3

; R3H = (A + B) + C
; R6H = D + X3 * Y3
| || MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4
MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

; R6H = E = X4 * Y4
| || ADDF32 R7H, R7H, R2H
NOP ; Wait for MPYF32 || ADDF32 to complete
MOV32 mem32, RaH — Move 32-bit Floating-Point Register Contents to Memory

ADDF32 R7H, R7H, R6H ; R7H = (A + B + C + D) + E
NOP
MOV32 @Result, R7H ; Wait for ADDF32 to complete
; Store the result

See also
MOV32 *(0:16bitAddr), loc32
MOV32 mem32, STF
MOV32 mem32, STF  

--- Move 32-bit STF Register to Memory ---

**Operands**

<table>
<thead>
<tr>
<th>STF</th>
<th>floating-point status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>points to the 32-bit destination memory</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 0000 0000  
MSW: 0000 0000 mem32

**Description**

Copy the floating-point status register, STF, to memory.  
[mem32] = STF

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

**Pipeline**

This is a single-cycle instruction.

**Example 1**

MOVW  DP, #0x0280 ; DP = 0x0280  
MOVI2F32 R0H, #2.0  ; R0H = 2.0 (0x40000000)  
MOVI2F32 R1H, #3.0  ; R1H = 3.0 (0x40400000)  
CMPP32  R0H, R1H  ; ZF = 0, NF = 1, STF = 0x00000004  
MOV32  @0, STF  ; [0x00A000] = 0x00000004

**Example 2**

MOV32  *SP++, STF  ; Store STF in stack  
MOVF32  R2H, #3.0  ; R2H = 3.0 (0x40400000)  
MOVF32  R3H, #5.0  ; R3H = 5.0 (0x40A00000)  
CMPP32  R2H, R3H  ; ZF = 0, NF = 1, STF = 0x00000004  
MOV32  R3H, R2H, LT  ; R3H = 3.0 (0x40400000)  
MOV32  STF, *--SP  ; Restore STF from stack

**See also**

MOV32 mem32, RaH  
MOV32 *(0:16bitAddr), loc32  
MOVST0 FLAG
MOV32 P, RaH — Move 32-bit Floating-Point Register Contents to P

**Operands**

<table>
<thead>
<tr>
<th>P</th>
<th>28x product register P</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1011 1111 loc32
MSW: IIII IIII IIII IIII

**Description**

Move the 32-bit value in RaH to the 28x product register P.

P = RaH

**Flags**

No flags affected in floating-point unit.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

MINF32 R0H,R1H ; Single-cycle instruction
NOP         ; 1 alignment cycle
MOV32 @ACC,R0H ; Copy R0H to ACC
NOP         ; Any instruction

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP                 ; 1 cycle delay for ADDF32 to complete
                    ; --- ADDF32 completes, R2H is valid
NOP                 ; 1 alignment cycle
MOV32 ACC, R2H     ; copy R2H into ACC, takes 1 cycle
                    ; --- MOV32 completes, ACC is valid NOP ; Any instruction

**Example**

MOV2IZF32 ROH, #2.5 ; ROH = 2.5 = 0x40200000
F32TOUI32 ROH, ROH
NOP                 ; Delay for conversion instruction
                    ; --- Conversion complete, ROH valid
NOP                 ; Alignment cycle
MOV32 P, ROH       ; P = 2 = 0x00000002

**See also**

MOV32 ACC, RaH
MOV32 XARn, RaH
MOV32 XT, RaH
MOV32 RaH, ACC — Move the Contents of ACC to a 32-bit Floating-Point Register

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>accumulator</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1011 1101 loc32  
MSW: IIII IIII IIII IIII  

**Description**

Move the 32-bit value in ACC to the floating-point register RaH.  
RaH = ACC

**Flags**

This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

MOV32 R0H, @ACC ; Copy ACC to R0H  
NOP ; Wait 4 cycles  
NOP ; Do not use FRACF32, UI16TOF32  
NOP ; I16TOF32, F32TOUI32 or F32TOI32  
NOP ; <-- R0H is valid

**Example**

MOV AH, #0x0000  
MOV AL, #0x0200 ; ACC = 512  
MOV32 R0H, ACC  
NOP  
NOP  
NOP UI32TOF32 R0H, R0H ; R0H = 512.0 (0x44000000)

**See also**

MOV32 RaH, P  
MOV32 RaH, XARn  
MOV32 RaH, XT
MOV32 RaH, mem32 {, CNDF}  

Conditional 32-bit Move

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to the 32-bit source memory location</td>
</tr>
<tr>
<td>CNDF</td>
<td>optional condition.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1010 CNDF  
MSW: 0000 0aaa mem32

Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

\[
\text{if (CNDF == TRUE)} \RaH = \text{[mem32]}
\]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.  
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\[
\text{if(CNDF == UNCF)} \\
\text{\{ } \\
\text{\quad NF = RaH[31]; ZF = 0; } \\
\text{\quad if(RaH[30:23] == 0) \{ ZF = 1; NF = 0; \} NI = RaH[31]; ZI = 0; } \\
\text{\quad if(RaH[31:0] == 0) ZI = 1; } \\
\text{\}} \\
\text{\quad else No flags modified;}
\]

Pipeline

This is a single-cycle instruction.
MOV32 RaH, mem32 {, CNDF} — Conditional 32-bit Move

Example

```
MOVW DP, #0x0300 ; DP = 0x0300
MOV @0, #0x5555 ; [0x00C000] = 0x5555
MOV @1, #0x5555 ; [0x00C001] = 0x5555
MOVZF32 R3H, #7.0 ; R3H = 7.0 (0x40E00000)
MOVZF32 R4H, #7.0 ; R4H = 7.0 (0x40E00000)
MAXF32 R3H, R4H ; ZF = 1, NF = 0
MOV32 R1H, @0, EQ ; R1H = 0x55555555
```

See also

MOV32 RaH, RbH{, CNDF}
MOVD32 RaH, mem32
**MOV32 RaH, P**  
*Move the Contents of P to a 32-bit Floating-Point Register*

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>product register</td>
</tr>
</tbody>
</table>

**Opcode**

- **LSW:** 1011 1101 loc32
- **MSW:** IIII IIII IIII IIII

**Description**

Move the 32-bit value in the product register, P, to the floating-point register RaH.

RaH = P

**Flags**

This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>Ni</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

```assembly
MOV32 R0H,@P ; Copy P to R0H
NOP ; Wait 4 alignment cycles
NOP ; Do not use FRACF32, UI16TOF32
NOP ; I16TOF32, F32TOUI32 or F32TOI32
NOP ;
; <-- R0H is valid
; Instruction can use R0H as a source
```

**Example**

```assembly
MOV PH, #0x0000
MOV PL, #0x0200 ; P = 512
MOV32 R0H, P
NOP
NOP
NOP
NOP
UI32TOF32 R0H, R0H ; R0H = 512.0 (0x44000000)
```

**See also**

- MOV32 RaH, ACC
- MOV32 RaH, XARn
- MOV32 RaH, XT
MOV32 RaH, RbH {, CNDF}  Conditional 32-bit Move

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>CNDF</td>
<td>optional condition.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1100 CNDF
MSW: 0000 0000 00bb baaa

Description
If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

\[
\text{if (CNDF == TRUE) RaH = RbH}
\]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\[
\text{if (CNDF == UNCF) \{ NF = RaH(31); ZF = 0;}
\]
\[
\text{if (RaH[30:23] == 0 \{ ZF = 1; NF = 0;\} NI = RaH(31); ZI = 0;}
\]
\[
\text{if (RaH[31:0] == 0 \{ ZI = 1; \} else No flags modified;}
\]

Pipeline
This is a single-cycle instruction.

Example

MOV32 RaH, #8.0 ; RaH = 8.0 (0x41000000)
MOV32 RaH, #7.0 ; RaH = 7.0 (0x40E00000)
MAXF32 R3H, R4H ; ZF = 0, NF = 0
MOV32R1H, R3H, GT ; R1H = 8.0 (0x41000000)

See also
MOV32 RaH, mem32{, CNDF}
MOV32 RaH, XARn — Move the Contents of XARn to a 32-bit Floating-Point Register

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XARn</td>
<td>auxiliary register (XAR0 - XAR7)</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>1011 1101 loc32</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>IIII IIII IIII IIII</td>
</tr>
</tbody>
</table>

**Description**

Move the 32-bit value in the auxiliary register XARn to the floating point register RaH.

RaH = XARn

**Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

MOV32 R0H, @XAR7 ; Copy XAR7 to R0H
NOP ; Wait 4 alignment cycles
NOP ; Do not use FRACF32, UI16TOF32
NOP ; I16TOF32, F32TOUI32 or F32TOI32
NOP ;
; <--- R0H is valid
ADDF32 R2H, R1H, R0H ; Instruction can use R0H as a source

**Example**

MOV32 R0H, XAR1

MOV32 R0H, XAR1
NOP
NOP
NOP
NOP

UI32TOF32 R0H, R0H ; R0H = 512.0 (0x44000000)

**See also**

MOV32 RaH, ACC
MOV32 RaH, P
MOV32 RaH, XT
MOV32 RaH, XT — Move the Contents of XT to a 32-bit Floating-Point Register

Operands

| RaH       | floating-point register (R0H to R7H) |
| XT        | auxiliary register (XAR0 - XAR7)     |

Opcode

LSW: 1011 1101 loc32
MSW: IIII IIII IIII IIII

Description
Move the 32-bit value in temporary register, XT, to the floating-point register RaH.

RaH = XT

Flags
This instruction does not modify any STF register flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline
While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

MOV32 R0H, XT ; Copy XT to R0H
NOP ; Wait 4 alignment cycles
NOP ; Do not use FRACF32, UI16TOF32
NOP ; I16TOF32, F32TOUI32 or F32TOI32
NOP ;
<= R0H is valid
ADD F32 R2H,R1H,R0H ; Instruction can use R0H as a source

Example

MOVIZF32 R6H, #5.0 ; R6H = 5.0 (0x40A00000)
NOP ; 1 Alignment cycle
MOV32 XT, R6H ; XT = 5.0 (0x40A00000)
MOV32 R1H, XT ; R1H = 5.0 (0x40A00000)

See also
MOV32 RaH, ACC
MOV32 RaH, P
MOV32 RaH, XARn
**MOV32 STF, mem32 — Move 32-bit Value from Memory to the STF Register**

**Operands**

<table>
<thead>
<tr>
<th>STF</th>
<th>floating-point unit status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>pointer to the 32-bit source memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 1000 0000  
MSW: 0000 0000 mem32

**Description**

Move from memory to the floating-point unit's status register STF.

STF = [mem32]

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>Ni</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Restoring status register will overwrite all flags.

**Pipeline**

This is a single-cycle instruction.

**Example 1**

MOVW DP, #0x0300 ; DP = 0x0300  
MOV @2, #0x020C ; [0x00C002] = 0x020C  
MOV @3, #0x0000 ; [0x00C003] = 0x0000  
MOV32 STF, @2 ; STF = 0x0000020C

**Example 2**

MOV32 *SP++, STF ; Store STF in stack  
MOVF32 R2H, #3.0 ; R2H = 3.0 (0x40400000)  
MOVF32 R3H, #5.0 ; R3H = 5.0 (0x40A00000)  
CMFF32 R2H, R3H ; ZF = 0, NF = 1, STF = 0x00000004  
MOV32 R3H, R2H, LT ; R3H = 3.0 (0x40400000)  
MOV32 STF, *--SP ; Restore STF from stack

**See also**

MOV32 mem32, STF  
MOVST0 FLAG
MOV32 XARn, RaH  —  Move 32-bit Floating-Point Register Contents to XARn

**Operands**

- **XARn**: 28x auxiliary register (XAR0 - XAR7)
- **RaH**: Floating-point source register (R0H to R7H)

**Opcode**

- **LSW**: 1011 1111 loc32
- **MSW**: IIII IIII IIII IIII

**Description**

Move the 32-bit value from the floating-point register RaH to the auxiliary register XARn.

- **XARn = RaH**

**Flags**

No flags affected in floating-point unit.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H, R1H ; Single-cycle instruction
NOP ; 1 alignment cycle
MOV32 @ACC, R0H ; Copy R0H to ACC
NOP ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP ; 1 cycle delay for ADDF32 to complete
NOP ; 1 alignment cycle
MOV32 ACC, R2H ; copy R2H into ACC, takes 1 cycle
NOP ; <-- MOV32 completes, ACC is valid
NOP ; Any instruction
```

**Example**

```
MOVIZF32 R0H, $2.5 ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP ; Delay for conversion instruction
NOP ; <-- Conversion complete, R0H valid
NOP ; Alignment cycle
MOV32 XAR0, R0H ; XAR0 = 2 = 0x00000002
```

**See also**

- MOV32 ACC, RaH
- MOV32 P, RaH
- MOV32 XT, RaH
MOV32 XT, RaH — Move 32-bit Floating-Point Register Contents to XT

Operands

<table>
<thead>
<tr>
<th>XT</th>
<th>temporary register</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1111 loc32  
MSW: I IIII IIII IIII IIII

Description

Move the 32-bit value in RaH to the temporary register XT.  
XT = RaH

Flags

No flags affected in floating-point unit.

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>N1</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H ; Single-cycle instruction
NOP ; 1 cycle delay for ADDF32 to complete
MOV32 @XT,R0H ; Copy R0H to ACC NOP
; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H ; 2 pipeline instruction (2p)
NOP ; 1 cycle delay for ADDF32 to complete
NOP ; 1 alignment cycle
MOV32 XT, R2H ; copy R2H into ACC, takes 1 cycle
; <-- MOV32 completes, ACC is valid
NOP ; Any instruction
```

Example

```
MOV1ZF32 R0H, #2.5 ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP ; Delay for conversion instruction
; <-- Conversion complete, R0H valid
NOP ; Alignment cycle
MOV32 XT, R0H ; XT = 0x00000002
```

See also

MOV32 ACC, RaH  
MOV32 P, RaH  
MOV32 XARn, RaH
MOVD32 RaH, mem32  Move 32-bit Value from Memory with Data Copy

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>pointer to the 32-bit source memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0010 0011
MSW: 0000 0aaa mem32

Description

Move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

\[ RaH = [\text{mem32}][\text{mem32}+2] = [\text{mem32}] \]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\[ NF = RaH[31]; \]
\[ ZF = 0; \]
\[ \text{if}(RaH[30:23] == 0) \{ ZF = 1; NF = 0; \} \]
\[ NI = RaH[31]; \]
\[ ZI = 0; \]
\[ \text{if}(RaH[31:0] == 0) ZI = 1; \]

Pipeline

This is a single-cycle instruction.

Example

MOVW DP, #0x02C0 ; DP = 0x02C0
MOV @2, #0x0000 ; [0x00B002] = 0x0000
MOV @3, #0x4110 ; [0x00B003] = 0x4110
MOVD32 R7H, @2 ; R7H = 0x41100000,
                 ; [0x00B004] = 0x0000, [0x00B005] = 0x4110

See also

MOV32 RaH, mem32 {,CNDF}
MOVF32 RaH, #32F  —  Load the 32-bits of a 32-bit Floating-Point Register

MOVF32 RaH, #32F  

Load the 32-bits of a 32-bit Floating-Point Register

Operands

This instruction is an alias for MOVIZ and MOVXI instructions. The second operand is translated by the assembler such that the instruction becomes:

- MOVIZ RaH, #16FHiHex
- MOVXI RaH, #16FLoHex

| RaH | Floating-point destination register (R0H to R7H) |
| #32F | Immediate float value represented in floating-point representation |

Opcode

- LSW: 1110 1000 0000 0III (opcode of MOVIZ RaH, #16FHiHex)
- MSW: IIII IIII IIII Iaaa
- LSW: 1110 1000 0000 1III (opcode of MOVXI RaH, #16FLoHex)
- MSW: IIII IIII IIII Iaaa

Description

Note: This instruction accepts the immediate operand only in floating-point representation. To specify the immediate value as a hex value (IEEE 32-bit floating-point format) use the MOVI32 RaH, #32FHex instruction.

Load the 32-bits of RaH with the immediate float value represented by #32F.

#32F is a float value represented in floating-point representation. The assembler will only accept a float value represented in floating-point representation. That is, 3.0 can only be represented as #3.0. #0x40400000 will result in an error.

RaH = #32F

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

Depending on #32FH, this instruction takes one or two cycles. If all of the lower 16-bits of the IEEE 32-bit floating-point format of #32F are zeros, then the assembler will convert MOVF32 into only MOVIZ instruction. If the lower 16-bits of the IEEE 32-bit floating-point format of #32F are not zeros, then the assembler will convert MOVF32 into MOVIZ and MOVXI instructions.

Example

- MOVF32 R1H, #3.0 ; R1H = 3.0 (0x40400000)
  - Assembler converts this instruction as
  - MOVIZ R1H, #0x4040
- MOVF32 R2H, #0.0 ; R2H = 0.0 (0x00000000)
  - Assembler converts this instruction as
  - MOVIZ R2H, #0x0
- MOVF32 R3H, #12.265 ; R3H = 12.625 (0x41443D71)
  - Assembler converts this instruction as
  - MOVIZ R3H, #0x4144
  - MOVXI R3H, #0x3D71

See also

- MOVIZ RaH, #16FHiHex
- MOVXI RaH, #16FLoHex
- MOV32 RaH, #32FHex
- MOVIZF32 RaH, #16FHi
MOVI32 RaH, #32FHex  —  Load the 32-bits of a 32-bit Floating-Point Register with the immediate

**Operands**

This instruction is an alias for MOVIZ and MOVXI instructions. The second operand is translated by the assembler such that the instruction becomes:

```plaintext
MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex
```

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#32FHex</td>
<td>A 32-bit immediate value that represents an IEEE 32-bit floating-point value.</td>
</tr>
</tbody>
</table>

**Opcode**

```
LSW: 1110 1000 0000 0III (opcode of MOVIZ RaH, #16FHiHex)  
MSW: IIII IIII IIII Iaaa

LSW: 1110 1000 0000 1III (opcode of MOVXI RaH, #16FLoHex)  
MSW: IIII IIII IIII Iaaa
```

**Description**

Note: This instruction only accepts a hex value as the immediate operand. To specify the immediate value with a floating-point representation use the MOVF32 RaH, #32F instruction.

Load the 32-bits of RaH with the immediate 32-bit hex value represented by #32Fhex. #32Fhex is a 32-bit immediate hex value that represents the IEEE 32-bit floating-point value of a floating-point number. The assembler will only accept a hex immediate value. That is, 3.0 can only be represented as #0x40400000. #3.0 will result in an error.

```
RaH = #32FHex
```

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

Depending on #32FHex, this instruction takes one or two cycles. If all of the lower 16-bits of #32FHex are zeros, then assembler will convert MOVI32 to the MOVIZ instruction. If the lower 16-bits of #32FHex are not zeros, then assembler will convert MOVI32 to a MOVIZ and a MOVXI instruction.

**Example**

```
MOVI32 R1H, #0x40400000 ; R1H = 0x40400000  
    ; Assembler converts this instruction as  
    ; MOVIZ R1H, #0x4040

MOVI32 R2H, #0x00000000 ; R2H = 0x00000000  
    ; Assembler converts this instruction as  
    ; MOVIZ R2H, #0x0

MOVI32 R3H, #0x40004001 ; R3H = 0x40004001  
    ; Assembler converts this instruction as  
    ; MOVIZ R3H, #0x4000 ; MOVXI R3H, #0x4001

MOVI32 R4H, #0x00000404 ; R4H = 0x00000404  
    ; Assembler converts this instruction as  
    ; MOVIZ R4H, #0x0000 ; MOVXI R4H, #0x4040
```

**See also**

- MOVIZ RaH, #16FHiHex
- MOVXI RaH, #16FLoHex
- MOVF32 RaH, #32F
- MOVIZF32 RaH, #16FHi
MOVIZ RaH, #16FHiHex — Load the Upper 16-bits of a 32-bit Floating-Point Register

**Operands**

| RaH | floating-point register (R0H to R7H) |
| #16FHiHex | A 16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

**Opcode**

| LSW: 1110 1000 0000 0III |
| MSW: IIII IIII IIII Iaaa |

**Description**

Note: This instruction only accepts a hex value as the immediate operand. To specify the immediate value with a floating-point representation use the MOVIZF32 pseudo instruction.

Load the upper 16-bits of RaH with the immediate value #16FHiHex and clear the low 16-bits of RaH.

#16FHiHex is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. The assembler will only accept a hex immediate value. That is, -1.5 can only be represented as #0xBFC0. #1.5 will result in an error.

By itself, MOVIZ is useful for loading a floating-point register with a constant in which the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). If a constant requires all 32-bits of a floating-point register to be initialized, then use MOVIZ along with the MOVXI instruction.

RaH[31:16] = #16FHiHex
RaH[15:0] = 0

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.

**Example**

; Load R0H with -1.5 (0xBFC00000)
MOVIZ R0H, #0xBFC0 ; R0H = 0xBFC00000

; Load R0H with pi = 3.141593 (0x40490FDB)
MOVIZ R0H, #0x4049 ; R0H = 0x40490000
MOVXI R0H, #0x0FDB ; R0H = 0x40490FDB

**See also**

MOVIZF32 RaH, #16FHi
MOVXI RaH, #16FLoHex
MOVIZF32 RaH, #16FHi — Load the Upper 16-bits of a 32-bit Floating-Point Register

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 0000 0III
MSW: IIII IIII IIII Iaaa

Description

Load the upper 16-bits of RaH with the value represented by #16FHi and clear the low 16-bits of RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). #16FHi can be specified in hex or float. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

MOVIZF32 is an alias for the MOVIZ RaH, #16FHiHex instruction. In the case of MOVIZF32 the assembler will accept either a hex or float as the immediate value and encodes it into a MOVIZ instruction. For example, MOVIZF32 RaH, #-1.5 will be encoded as MOVIZ RaH, 0xBFC0.

RaH[31:16] = #16FHi
RaH[15:0] = 0

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.

Example

```assembly
MOVIZF32 R0H, #3.0         ; R0H = 3.0 = 0x40400000
MOVIZF32 R1H, #1.0         ; R1H = 1.0 = 0x3F800000
MOVIZF32 R2H, #2.5         ; R2H = 2.5 = 0x40200000
MOVIZF32 R3H, #-5.5        ; R3H = -5.5 = 0xC0B00000
MOVIZF32 R4H, #0xC0B0      ; R4H = -5.5 = 0xC0B00000

; Load R5H with pi = 3.141593 (0x40490000)
;    MOVIZF32 R5H, #3.141593 ; R5H = 3.140625 (0x40490000)
;
; Load R0H with a more accurate pi = 3.141593 (0x40490FDB)
;    MOVIZF32 R0H, #0x4049 ; R0H = 0x40490000
    MOVXI R0H, #0x0FDB ; R0H = 0x40490FDB
```

See also

MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex
**MOVST0 FLAG — Load Selected STF Flags into ST0**

**Operands**

<table>
<thead>
<tr>
<th>FLAG</th>
<th>Selected flag</th>
</tr>
</thead>
</table>

**Opcode**

LSW: 1010 1101 FFFF FFFF

**Description**

Load selected flags from the STF register into the ST0 register of the 28x CPU where FLAG is one or more of TF, CI, ZI, ZF, NI, NF, LUF or LVF. The specified flag maps to the ST0 register as follows:

- Set OV = 1 if LVF or LUF is set. Otherwise clear OV.
- Set N = 1 if NF or NI is set. Otherwise clear N.
- Set Z = 1 if ZF or ZI is set. Otherwise clear Z.
- Set C = 1 if TF is set. Otherwise clear C.
- Set TC = 1 if TF is set. Otherwise clear TF.

If any STF flag is not specified, then the corresponding ST0 register bit is not modified.

**Restrictions**

Do not use the MOVST0 instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the MOVST0 operation.

```plaintext
; The following is INVALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
MOVST0 TF ; INVALID, do not use MOVST0 in a delay slot

; The following is VALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
NOP ; 1 delay cycle, R2H updated after this instruction
MOVST0 TF ; VALID
```

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

When the flags are moved to the C28x ST0 register, the LUF or LVF flags are automatically cleared if selected.

**Pipeline**

This is a single-cycle instruction.

**Example**

Program flow is controlled by C28x instructions that read status flags in the status register 0 (ST0). If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into ST0 flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVST0 FLAG instruction is used to load the current value of specified STF flags into the respective bits of ST0. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified.

Loop:

```
MOV32 R0H,*XAR4++
MOV32 R1H,*XAR3++
CMPF32 R1H, R0H
MOVST0 ZF, NF
BF Loop, GT ; Loop if (R1H > R0H)
```

**See also**

- MOV32 mem32, STF
- MOV32 STF, mem32
MOVXI RaH, #16FLoHex — Move Immediate to the Low 16-bits of a Floating-Point Register

**Operands**

<table>
<thead>
<tr>
<th>Ra</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits will not be modified.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 1000 0000 1III  
MSW: IIII IIII IIII Iaaa

**Description**

Load the low 16-bits of RaH with the immediate value #16FLoHex. #16FLoHex represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits of RaH will not be modified. MOVXI can be combined with the MOVIZ or MOVIZF32 instruction to initialize all 32-bits of a RaH register.

RaH[15:0] = #16FLoHex  
RaH[31:16] = Unchanged

**Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
; Load R0H with pi = 3.141593 (0x40490FDB)  
MOVIZ R0H,#0x4049 ; R0H = 0x40490000  
MOVXI R0H,#0x0FDB ; R0H = 0x40490FDB  
```

**See also**

MOVIZ RaH, #16FHiHex
MOVIZF32 RaH, #16FHi
MPYF32 RaH, RbH, RcH  32-bit Floating-Point Multiply

Operands

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0000 0000
MSW: 0000 000c cbbb baaa

Description

Multiply the contents of two floating-point registers.

RaH = RbH * RcH

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

MPYF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- MPYF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate Y = A * B:

```
MOVL XAR4, #A
MOV32 R0H, *XAR4 ; Load R0H with A
MOVL XAR4, # B
MOV32 R1H, *XAR4 ; Load R1H with B
MPYF32 R0H,R1H,R0H ; Multiply A * B
MOVL XAR4, #Y
; <--MPYF32 complete
MOV32 *XAR4,R0H ; Save the result
```

See also

MPYF32 RaH, #16FHi, RbH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RdH, ReH, RfH || MOV32 mem32, RaH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, #16FHi, RbH  32-bit Floating-Point Multiply

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
<tr>
<td>RcH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1000 01II IIII
MSW: IIII IIII IIbb baaa

Description

Multiply RbH with the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

RaH = RbH * #16FHi:0

This instruction can also be written as MPYF32 RaH, RbH, #16FHi.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
MPYF32 RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- MPYF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example 1

MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)
MPYF32 R4H, #3.0, R3H ; R4H = 3.0 * R3H
MOVL XAR1, #0xB006 ; <-- Non conflicting instruction
; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB006

Example 2

; Same as above example but #16FHi is represented in Hex

MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)
MPYF32 R4H, #0x4040, R3H ; R4H = 0x4040 * R3H ; 3.0 is represented as 0x40400000 in IEEE 754 32-bit format
MOVL XAR1, #0xB006 ; <-- Non conflicting instruction
; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB006
MPYF32 RaH, #16FHi, RbH — 32-bit Floating-Point Multiply

See also
MPYF32 RaH, RbH, #16FHi
MPYF32 RaH, RbH, RcH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RlH
**MPYF32 RaH, RbH, #16FHi — 32-bit Floating-Point Multiply**

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 1000 01II IIII  
MSW: IIII IIII IIbb baaa

**Description**

Multiply RbH with the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

RaH = RbH * #16FHi:0

This instruction can also be written as MPYF32 RaH, #16FHi, RbH.

**Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:
- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

```
MPYF32 RaH, RbH, #16FHi ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- MPYF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

**Example 1**

```
MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)
MPYF32 R4H, R3H, #3.0 ; R4H = R3H * 3.0
MOVL XAR1, #0xB008 ; <-- Non conflicting instruction
    ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB008
```

**Example 2**

```
Same as above example but #16FHi is represented in Hex
MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)
MPYF32 R4H, R3H, #0x4040 ; R4H = R3H * 0x4040
    ; 3.0 is represented as 0x40400000 in
    ; IEEE 754 32-bit format
MOVL XAR1, #0xB008 ; <-- Non conflicting instruction
    ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
MOV32 *XAR1, R4H ; Save the result in memory location 0xB008
```
MPYF32 RaH, RbH, #16FHi — 32-bit Floating-Point Multiply

See also

MPYF32 RaH, #16FHi, RbH
MPYF32 RaH, RbH, RcH
MPYF32 RaH, RbH, RcH \| ADDF32 RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Add

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register for MPYF32 (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RaH cannot be the same register as RdH</td>
</tr>
<tr>
<td>RbH</td>
<td>Floating-point source register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>Floating-point source register for MPYF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RdH</td>
<td>Floating-point destination register for ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td></td>
<td>RdH cannot be the same register as RaH</td>
</tr>
<tr>
<td>ReH</td>
<td>Floating-point source register for ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RfH</td>
<td>Floating-point source register for ADDF32 (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 0111 0100 00ff
- MSW: feee dddc ccbf baa

Description

Multiply the contents of two floating-point registers with parallel addition of two registers.

\[
RaH = RbH \times RcH \\
RdH = ReH + RfH
\]

This instruction can also be written as:

MACF32 RaH, RbH, RcH, RdH, ReH, RfH

Restrictions

The destination register for the MPYF32 and the ADDF32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:
- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline

Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

\[
\begin{align*}
\text{MPYF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)} \\
\mid \mid \text{ADDF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)} \\
\text{NOP ; 1 cycle delay or non-conflicting instruction} \\
\text{; <-- MPYF32, ADDF32 complete, RaH, RdH updated} \\
\text{NOP}
\end{align*}
\]

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.
Example

; Perform 5 multiply and accumulate operations:
;
; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3
;
; Result = A + B + C + D + E

MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0

; R2H = A = X0 * Y0
MPYF32 R2H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y1

; R3H = B = X1 * Y1
MPYF32 R3H, R0H, R1H ; In parallel R0H = X2
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2

; R3H = A + B
; R2H = C = X2 * Y2
MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3

; R3H = (A + B) + C
; R2H = D = X3 * Y3
MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X4
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4

; R2H = E = X4 * Y4
MPYF32 R2H, R0H, R1H ; In parallel R3H = (A + B + C) + D
|| ADDF32 R3H, R3H, R2H NOP ; Wait for MPYF32 || ADDF32 to complete

ADDF32 R3H, R3H, R2H ; R3H = (A + B + C + D) + E NOP

; Wait for ADDF32 to complete
MOV32 @Result, R3H ; Store the result

See also

MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32 32-bit Floating-Point Multiply with Parallel Move

Operands

| RdH | Floating-point destination register for the MPYF32 (R0H to R7H) RdH cannot be the same register as RaH |
| ReH | Floating-point source register for the MPYF32 (R0H to R7H) |
| RfH | Floating-point source register for the MPYF32 (R0H to R7H) |
| RaH | Floating-point destination register for the MOV32 (R0H to R7H) RaH cannot be the same register as RdH |
| mem32 | pointer to a 32-bit memory location. This will be the source of the MOV32. |

Opcode

LSW: 1110 0011 0000 fffe
MSW: eedd daaa mem32

Description

Multiply the contents of two floating-point registers and load another.

RdH = ReH * RfH
RaH = [mem32]

Restrictions

The destination register for the MPYF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;

Pipeline

MPYF32 takes 2 pipeline-cycles (2p) and MOV32 takes a single cycle. That is:

MPYF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
|| MOV32 RaH, mem32 ; 1 cycle
||| NOP ; 1 cycle delay or non-conflicting instruction
|<-- MOV32 completes, RaH updated
|<-- MPYF32 completes, RdH updated

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.
Example

Calculate $Y = M1 \times X1 + B1$. This example assumes that $M1$, $X1$, $B1$ and $Y1$ are all on the same data page.

```assembly
MOVW DP, #M1 ; Load the data page
MOV32 ROH, #M1 ; Load ROH with M1
MOV32 R1H, #X1 ; Load R1H with X1
MPYF32 R1H, R1H, ROH ; Multiply M1\times X1
|| MOV32 ROH, #B1 ; and in parallel load ROH with B1
; <-- MOV32 complete
NOP ; Wait 1 cycle for MPYF32 to complete
; <-- MPYF32 complete
ADDF32 R1H, R1H, ROH ; Add M\times X1 to B1 and store in R1H
NOP ; Wait 1 cycle for ADDF32 to complete
; <-- ADDF32 complete
MOV32 @Y1, R1H ; Store the result
```

Calculate $Y = (A \times B) \times C$:

```assembly
MOVL XAR4, #A
MOV32 ROH, *XAR4 ; Load ROH with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
MOVL XAR4, #C
MPYF32 R1H, R1H, ROH ; Calculate R1H = A \times B
|| MOV32 ROH, *XAR4 ; and in parallel load R2H with C
; <-- MOV32 complete
MOVL XAR4, #Y
; <-- MPYF32 complete
MPYF32 R2H, R1H, ROH ; Calculate $Y = (A \times B) \times C$
NOP ; Wait 1 cycle for MPYF32 to complete
; MPYF32 complete
MOV32 *XAR4, R2H
```

See also

- $\text{MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32}$
- $\text{MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32}$
- $\text{MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32}$
- $\text{MACF32 R7H, R3H, mem32, *XAR7++}$
### Operands

| RdH | Floating-point destination register for the MPYF32 (R0H to R7H) |
| ReH | Floating-point source register for the MPYF32 (R0H to R7H) |
| RfH | Floating-point source register for the MPYF32 (R0H to R7H) |
| mem32 | pointer to a 32-bit memory location. This will be the destination of the MOV32. |
| RaH | Floating-point source register for the MOV32 (R0H to R7H) |

### Opcode

- **LSW**: 1110 0000 0000 fffe
- **MSW**: eedd daaa mem32

### Description

Multiply the contents of two floating-point registers and move from memory to register.  
\[ \text{RdH} = \text{ReH} \times \text{RfH}, \ [\text{mem32}] = \text{RaH} \]

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:
- \( \text{LUF} = 1 \) if MPYF32 generates an underflow condition.
- \( \text{LVF} = 1 \) if MPYF32 generates an overflow condition.

### Pipeline

MPYF32 takes 2 pipeline-cycles (2p) and MOV32 takes a single cycle. That is:

\[
\begin{align*}
\text{MPYF32 RdH, ReH, RfH} & ; 2 \text{ pipeline cycles (2p)} \\
|| \text{MOV32 mem32, RaH} & ; 1 \text{ cycle} \\
& ; \text{--- MOV32 completes, mem32 updated} \\
\text{NOP} & ; 1 \text{ cycle delay or non-conflicting instruction} \\
& ; \text{--- MPYF32 completes, RdH updated} \\
\text{NOP} & \text{--- MOV32 completes, R3 value} \\
\end{align*}
\]

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

### Example

\[
\begin{align*}
\text{MOVL XAR1, #0xC003} & ; \text{XAR1} = 0xC003 \\
\text{MOVIZF32 R3H, #2.0} & ; \text{R3H} = 2.0 (0x40000000) \\
\text{MPYF32 R3H, R3H, #5.0} & ; \text{R3H} = \text{R3H} \times 5.0 \\
\text{MOVIZF32 R1H, #5.0} & ; \text{R1H} = 5.0 (0x40A00000) \\
\text{MPYF32 R3H, R1H, R3H} & ; \text{--- MPYF32 complete, R3H} = 10.0 (0x41200000) \\
|| \text{MOV32 *XAR1, R3H} & ; \text{and in parallel store previous R3 value} \\
& ; \text{MOV32 complete, [0xC003] = 0x4120,} \\
& ; \text{[0xC002] = 0x0000} \\
\text{NOP} & ; 1 \text{ cycle delay for MPYF32 to complete} \\
& ; \text{--- MPYF32 completes, R3H} = 50.0 (0x42480000) \\
\end{align*}
\]

### See also

MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32  
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32  
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32  
MACF32 R7H, R3H, mem32, *XAR7++
**MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Subtract**

### Operands

- **RaH**: Floating-point destination register for MPYF32 (R0H to R7H)
- **RbH**: Floating-point source register for MPYF32 (R0H to R7H)
- **RcH**: Floating-point source register for MPYF32 (R0H to R7H)
- **RdH**: Floating-point destination register for SUBF32 (R0H to R7H)
- **ReH**: Floating-point source register for SUBF32 (R0H to R7H)
- **RfH**: Floating-point source register for SUBF32 (R0H to R7H)

**Restrictions**

The destination register for the MPYF32 and the SUBF32 must be unique. That is, RaH cannot be the same register as RdH.

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- **LUF** = 1 if MPYF32 or SUBF32 generates an underflow condition.
- **LVF** = 1 if MPYF32 or SUBF32 generates an overflow condition.

### Pipeline

MPYF32 and SUBF32 both take 2 pipeline-cycles (2p). That is:

\[
\text{MPYF32 } \text{RaH, RbH, RcH ; 2 pipeline cycles (2p)}
\]

\[
\text{|| SUBF32 } \text{RdH, ReH, RfH ; 2 pipeline cycles (2p)}
\]

\[
\text{NOP ; 1 cycle delay or non-conflicting instruction}
\]

\[
\text{; <-- MPYF32, SUBF32 complete. RaH, RdH updated}
\]

\[
\text{NOP}
\]

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.

### Example

\[
\text{MOVIZF32 R4H, #5.0 ; R4H = 5.0 (0x40A00000)}
\]

\[
\text{MOVIZF32 R5H, #3.0 ; R5H = 3.0 (0x40400000)}
\]

\[
\text{MPYF32 R6H, R4H, R5H ; R6H = R4H * R5H}
\]

\[
\text{|| SUBF32 R7H, R4H, R5H ; R7H = R4H - R5H NOP}
\]

\[
\text{; 1 cycle delay for MPYF32 || SUBF32 to complete}
\]

\[
\text{; <-- MPYF32 || SUBF32 complete,}
\]

\[
\text{R6H = 15.0 (0x41700000), R7H = 2.0 (0x40000000)}
\]

### See also

- SUBF32 RaH, RbH, RcH
- SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
- SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH
NEGF32 RaH, RbH{, CNDF}  

Conditional Negation

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>CNDF</td>
<td>condition tested</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1010 CNDF  
MSW: 0000 0000 00bb baaa

**Description**

if (CNDF == true) {RaH = - RbH }  
else {RaH = RbH }

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.  
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.

**Example**

MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)  
MOVIZF32 R1H, #4.0 ; R1H = 4.0 (0x40800000)  
MOVIZF32 R2H, #-1.5 ; R2H = -1.5 (0x0BFC00000)  
MPYF32 R4H, R1H, R2H ; R4H = -6.0  
MPYF32 R5H, R0H, R1H ; R5H = 20.0  
; <-- R4H valid  
CMPF32 R4H, #0.0 ; NF = 1  
; <-- R5H valid  
NEGF32 R4H, R4H, LT ; if NF = 1, R4H = 6.0  
CMPF32 R5H, #0.0 ; NF = 0  
NEGF32 R5H, R5H, GEQ ; if NF = 0, R4H = -20.0

**See also**

ABSF32 RaH, RbH
**POP RB — Pop the RB Register from the Stack**

**Operands**

| RB         | repeat block register |

**Opcode**

LSW: 1111 1111 1111 0001

**Description**

Restore the RB register from stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

**Flags**

This instruction does not affect any flags floating-point Unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.

**Example**

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
中断:  ; RAS = RA, RA = 0
    ...
    PUSH RB ; Save RB register only if a RPTB block is used in the
    ISR    ...
    ...
    RPTB #BlockEnd, AL ; Execute the block AL+1 times
    ...
    ...
    BlockEnd ; End of block to be repeated
    ...
    ...
    POP RB ; Restore RB register
    ...
    IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```
; Repeat Block within a Low-Priority Interrupt (Interruptible)
中断:  ; RAS = RA, RA = 0
    ...
    PUSH RB ; Always save RB register
    ...
    CLR INTM ; Enable interrupts only after saving RB
    ...
    ...
    ; ISR may or may not include a RPTB block
    ...
    SETC INTM ; Disable interrupts before restoring RB
    ...
    POP RB ; Always restore RB register
    ...
    IRET ; RA = RAS, RAS = 0
```
POP RB — Pop the RB Register from the Stack

See also

PUSH RB
RPTB label, #RC
RPTB label, loc16
PUSH RB — Push the RB Register onto the Stack

PUSH RB

**Push the RB Register onto the Stack**

Operands

| RB | repeat block register |

Opcode

| LSW: 1111 1111 1111 0000 |

Description

Save the RB register on the stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

Flags

This instruction does not affect any flags floating-point Unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction for the first iteration, and zero cycles thereafter.

Example

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```plaintext
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
_interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Save RB register only if a RPTB block is used in the
ISR
...
RPTB #BlockEnd, AL ; Execute the block AL+1 times
...
BlockEnd ; End of block to be repeated
...
POP RB ; Restore RB register
...
IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```plaintext
; Repeat Block within a Low-Priority Interrupt (Interruptible)
_interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Always save RB register
...
CLRC INTM ; Enable interrupts only after saving RB
...
... ; ISR may or may not include a RPTB block
...
SETC INTM ; Disable interrupts before restoring RB
...
POP RB ; Always restore RB register
...
IRET ; RA = RAS, RAS = 0
```

See also

- POP RB
- RPTB label, #RC
- RPTB label, loc16
**RESTORE — Restore the Floating-Point Registers**

### Operands

| none      | This instruction does not have any operands |

### Opcode

| LSW: 1110 0101 0110 0010 |

### Description

Restore the floating-point register set (R0H - R7H and STF) from their shadow registers. The SAVE and RESTORE instructions should be used in high-priority interrupts. That is interrupts that cannot themselves be interrupted. In low-priority interrupt routines the floating-point registers should be pushed onto the stack.

### Restrictions

The RESTORE instruction cannot be used in any delay slots for pipelined operations. Doing so will yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the RESTORE operation.

```
; The following is INVALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
RESTORE ; INVALID, do not use RESTORE in a delay slot
```

```
; The following is VALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
NOP ; 1 delay cycle, R2H updated after this instruction
RESTORE ; VALID
```

### Flags

Restoring the status register will overwrite all flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Pipeline

This is a single-cycle instruction.
Example

The following example shows a complete context save and restore for a high-priority interrupt. Note that the CPU automatically stores the following registers: ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1 and PC. If an interrupt is low priority (that is it can be interrupted), then push the floating point registers onto the stack instead of using the SAVE and RESTORE operations.

; Interrupt Save
_HighestPriorityISR: ; Uninterruptable
ASP ; Align stack
PUSH RB ; Save RB register if used in the ISR
PUSH AR1H:AR0H ; Save other registers if used
PUSH XAR2
PUSH XAR3
PUSH XAR4
PUSH XAR5
PUSH XAR6
PUSH XAR7
PUSH XT
SPM 0 ; Set default C28 modes
CLRC AMODE
CLRC PAGE0,OVM
SAVE RNDF32=1 ; Save all FPU registers
... ; set default FPU modes
...
; Interrupt Restore
...
RESTORE ; Restore all FPU registers
POP XT ; restore other registers
POP XAR7
POP XAR6
POP XAR5
POP XAR4
POP XAR3
POP XAR2
POP AR1H:AR0H
POP RB ; restore RB register
NASP ; un-align stack
IRET ; return from interrupt

See also
SAVE FLAG, VALUE
**RPTB label, loc16** — Repeat A Block of Code

**Operands**

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block.</td>
</tr>
<tr>
<td>loc16</td>
<td>16-bit location for the repeat count value.</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011 0101 0bbb bbbb</td>
<td>0000 0000 loc16</td>
</tr>
</tbody>
</table>

**Description**

Initialize repeat block loop, repeat count from [loc16]

**Restrictions**

- The maximum block size is $\leq 127$ 16-bit words.
- An even aligned block must be $\geq 9$ 16-bit words.
- An odd aligned block must be $\geq 8$ 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch, or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

**Flags**

This instruction does not affect any flags in the floating-point unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This instruction takes four cycles on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

**Example**

The minimum size for the repeat block is 9 words if the block is even-aligned and 8 words if the block is odd-aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even-aligned. Since a NOP is a 16-bit instruction the RPTB will be odd-aligned. For blocks of 9 or more words, this is not required.

```
; Repeat Block of 8 Words (Interruptible)

; find the largest element and put its address in XAR6
.align 2
NOP
RPTB VECTOR_MAX_END, AR7 ; Execute the block AR7+1 times
MOVL ACC, XAR0
MOV32 R1H,*XAR0++ ; min size = 8, 9 words
MAXF32 R0H,R1H ; max size = 127 words
MOVST0 NF,ZF
MOV32 XAR6,ACC,LT
VECTOR_MAX_END: ; label indicates the end
    ; RA is cleared
```

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.
A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```assembly
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
; Interrupt: ; RAS = RA, RA = 0
... PUSH RB ; Save RB register only if a RPTB block is used in the ISR
... ...
RPTB #BlockEnd, AL ; Execute the block AL+1 times
... ...
... BlockEnd ; End of block to be repeated ...
... POP RB ; Restore RB register ...
IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```assembly
; Repeat Block within a Low-Priority Interrupt (Interruptible)
; Interrupt: ; RAS = RA, RA = 0
... PUSH RB ; Always save RB register ...
... CLRC INTM ; Enable interrupts only after saving RB ...
... ...
... ; ISR may or may not include a RPTB block ...
... ...
SETC INTM ; Disable interrupts before restoring RB ...
... POP RB ; Always restore RB register ...
... IRET ; RA = RAS, RAS = 0
```

See also

- `POP RB`
- `PUSH RB`
- `RPTB label, #RC`
RPTB label, #RC — Repeat a Block of Code

Operands

| label | This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block. |
| #RC  | 16-bit location |

Opcode

| LSW: 1011 0101 1bbb bbbb |
| MSW: cccc cccc cccc cccc |

Description

Repeat a block of code. The repeat count is specified as a immediate value.

Restrictions

- The maximum block size is \( \leq 127 \) 16-bit words.
- An even aligned block must be \( \geq 9 \) 16-bit words.
- An odd aligned block must be \( \geq 8 \) 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

This instruction does not affect any flags int the floating-point unit:

Pipeline

This instruction takes one cycle on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

Example

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

```assembly
; Repeat Block (Interruptible)
;
; find the largest element and put its address in XAR6
.align 2

NOP
RPTB VECTOR_MAX_END, #(4-1) ; Execute the block 4 times
MOV ACC,XAR0
MOV32 R1H,*XAR0++ ; 8 or 9 words block size 127 words
MAXF32 R0H,R1H
MOVS0 NF,ZF
MOVL XAR6,ACC,LT
VECTOR_MAX_END: ; RE indicates the end address
               ; RA is cleared
```

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.
A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt:       ; RAS = RA, RA = 0
...
PUSH RB           ; Save RB register only if a RPTB block is used in the ISR
...
RPTB #BlockEnd, #5 ; Execute the block 5+1 times
...
...
BlockEnd         ; End of block to be repeated
...
POP RB           ; Restore RB register
...
IRET             ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```
; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt:       ; RAS = RA, RA = 0
...
PUSH RB           ; Always save RB register
...
CLRC INTM        ; Enable interrupts only after saving RB
...
...
...               ; ISR may or may not include a RPTB block
...
...
SETC INTM        ; Disable interrupts before restoring RB
...
POP RB           ; Always restore RB register
...
IRET             ; RA = RAS, RAS = 0
```

See also

- POP RB
- PUSH RB
- RPTB #RSIZE, loc16
SAVE FLAG, VALUE — Save Register Set to Shadow Registers and Execute SETFLG

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG</td>
<td>11 bit mask indicating which floating-point status flags to change.</td>
</tr>
<tr>
<td>VALUE</td>
<td>11 bit mask indicating the flag value; 0 or 1.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 01FF FFFF
MSW: FFFF FVVV VVVV VVVV

Description

This operation copies the current working floating-point register set (R0H to R7H and STF) to the shadow register set and combines the SETFLG FLAG, VALUE operation in a single cycle. The status register is copied to the shadow register before the flag values are changed. The STF[SHDWM] flag is set to 1 when the SAVE command has been executed. The SAVE and RESTORE instructions should be used in high-priority interrupts. That is interrupts that cannot themselves be interrupted. In low-priority interrupt routines the floating-point registers should be pushed onto the stack.

Restrictions

Do not use the SAVE instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the SAVE operation.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Modified</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Any flag can be modified by this instruction.

Pipeline

This is a single-cycle instruction.

Example

To make it easier and more legible, the assembler will accept a FLAG=VALUE syntax for the SETFLG operation as shown below:

SAVE RNDF32=0, TF=1, ZF=0 ; FLAG = 01001000100, VALUE = X0X0XXX1XX
MOVST0 TF, ZF, LUF ; Copy the indicated flags to ST0
; Note: X means this flag will not be modified.
; The assembler will set these X values to 0.

The following example shows a complete context save and restore for a high priority interrupt. Note that the CPU automatically stores the following registers: ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1 and PC.
SAVE FLAG, VALUE — Save Register Set to Shadow Registers and Execute SETFLG

```
_HighestPriorityISR:
ASP    ;Align stack
PUSH RB       ; Save RB register if used in the ISR
PUSH AR1H:AR0H ; Save other registers if used
PUSH XAR2
PUSH XAR3
PUSH XAR4
PUSH XAR5
PUSH XAR6
PUSH XAR7
PUSH XT
SPM 0         ; Set default C28 modes
CLRC AMODE
CLRC PAGE0,OVM
SAVE RNDF32=0  ; Save all FPU registers
...          ; set default FPU modes
...          ...
...          ...
RESTORE       ; Restore all FPU registers
POP XT        ; restore other registers
POP XAR7
POP XAR6
POP XAR5
POP XAR4
POP XAR3
POP XAR2
POP AR1H:AR0H
POP RB        ; restore RB register
NASP          ; un-align stack IRET
              ; return from interrupt
```

See also

RESTORE
SETFLG FLAG, VALUE
SETFLG FLAG, VALUE — Set or clear selected floating-point status flags

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG</td>
<td>11 bit mask indicating which floating-point status flags to change.</td>
</tr>
<tr>
<td>VALUE</td>
<td>11 bit mask indicating the flag value; 0 or 1.</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 11 10 01 10 00 FF FFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: FFFF FVVV VVVV VVVV</td>
</tr>
</tbody>
</table>

Description

The SETFLG instruction is used to set or clear selected floating-point status flags in the STF register. The FLAG field is an 11-bit value that indicates which flags will be changed. That is, if a FLAG bit is set to 1 it indicates that flag will be changed; all other flags will not be modified. The bit mapping of the FLAG field is shown below:

```
  10  9  8  7  6  5  4  3  2  1  0
reserved RNDF32 reserved reserved TF ZI NI ZF NF LUF LVF
```

The VALUE field indicates the value the flag should be set to; 0 or 1.

Restrictions

Do not use the SETFLG instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the SETFLG operation.

```
; The following is INVALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
SETFLG RNDF32=1 ; INVALID, do not use SETFLG in a delay slot

; The following is VALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
NOP ; 1 delay cycle, R2H updated after this instruction
SETFLG RNDF32=1 ; VALID
```

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Any flag can be modified by this instruction.

Pipeline

This is a single-cycle instruction.

Example

To make it easier and legible, the assembler will accept a FLAG=VALUE syntax for the STFLG operation as shown below:

```
SETFLG RNDF32=0, TF=1, ZF=0 ; FLAG = 01001001000, VALUE = X0XX1XX0XXX
MOVST0 TF, ZF, LUF ; Copy the indicated flags to ST0
; X means this flag is not modified.
; The assembler will set X values to 0
```

See also

SAVE FLAG, VALUE
SUBF32 RaH, RbH, RcH  32-bit Floating-Point Subtraction

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R1)</td>
</tr>
<tr>
<td>RcH</td>
<td>Floating-point source register (R0H to R1)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0010 0000
MSW: 0000 000c ccbb baaa

Description

Subtract the contents of two floating-point registers

RaH = RbH - RcH

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if SUBF32 generates an underflow condition.
- LVF = 1 if SUBF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

SUBF32 RaH, RbH, RcH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <--- SUBF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

Calculate Y - A + B - C:

MOVL XAR4, #A
MOVL XAR4, #B
ADD   R0H,R1H,R0H ; Add A + B and in parallel
|| MOVL XAR4, #C
|| MOVF R2H,*XAR4 ; Load R2H with C
; <--- ADDF32 complete
SUBF32 R0H,R0H,R2H ; Subtract C from (A + B)
NOP ; <--- SUBF32 completes
MOV32 *XAR4,R0H ; Store the result

See also

SUBF32 RaH, #16FHi, RbH
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
**SUBF32 RaH, #16FHi, RbH  32-bit Floating Point Subtraction**

**Operands**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R1)</td>
</tr>
<tr>
<td>#16FHi</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R1)</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 1000</td>
<td>IIII IIII</td>
</tr>
<tr>
<td>11II IIII</td>
<td>IIbb baaa</td>
</tr>
</tbody>
</table>

**Description**

Subtract RbH from the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as # -1.5 or # 0xBFC0.

RaH = #16FHi:0 - RbH

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- **LUF** = 1 if MPYF32 generates an underflow condition.
- **LVF** = 1 if MPYF32 generates an overflow condition.

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

```
SUBF32 RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- SUBF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

**Example**

Calculate \( Y = 2.0 - (A + B) \):

```
MOVL XAR4, #A
MOV32 R0H, *XAR4 ; Load R0H with A
MOVL XAR4, #B
MOV32 R1H, *XAR4 ; Load R1H with B
ADDF32 R0H, R1H, R0H ; Add A + B and in parallel
NOP ; <-- ADDF32 complete
SUBF32 R0H, #2.0, R2H ; Subtract (A + B) from 2.0
NOP ; <-- SUBF32 completes
MOV32 *XAR4, R0H ; Store the result
```

**See also**

- SUBF32 RaH, RbH, RcH
- SUBF32 RdH, ReH, RiH || MOV32 RaH, mem32
- SUBF32 RdH, ReH, RiH || MOV32 mem32, RaH
- MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RiH
SUBF32 RdH, ReH, RfH ||MOV32 RaH, mem32 | 32-bit Floating-Point Subtraction with Parallel Move

Operands

<table>
<thead>
<tr>
<th>RdH</th>
<th>Floating-point destination register (R0H to R7H) for the SUBF32 operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReH</td>
<td>Floating-point source register (R0H to R7H) for the SUBF32 operation</td>
</tr>
<tr>
<td>RfH</td>
<td>Floating-point source register (R0H to R7H) for the SUBF32 operation</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H) for the MOV32 operation</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to 32-bit source memory location for the MOV32 operation</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 0010 fffe
MSW: eedd daaa mem32

Description

Subtract the contents of two floating-point registers and move from memory to a floating-point register.

RdH = ReH - RfH, RaH = [mem32]

Restrictions

The destination register for the SUBF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

- LUF = 1 if SUBF32 generates an underflow condition.
- LVF = 1 if SUBF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;

Pipeline

SUBF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

SUBF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
|| MOV32 RaH, mem32 ; 1 cycle
; <-- MOV32 completes, RaH updated
NOP ; 1 cycle delay or non-conflicting instruction
; <-- SUBF32 completes, RdH updated
NOP

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.
Example

MOV L XAR1, #0xC000 ; XAR1 = 0xC000
SUBF32 R0H, R1H, R2H ; (A) R0H = R1H - R2H
|| MOV32 R3H, *XAR1 ;
; <= R3H valid
; <= (A) completes, R0H valid, R4H valid
ADD F32 R5H, R4H, R3H ; (B) R5H = R4H + R3H
|| MOV32 *+XAR1[4], R0H ;
; <= R0H stored
MOV L XAR2, #0xE000 ;
; <= (B) completes, R5H valid
MOV32 *XAR2, R5H ;
; <= R5H stored

See also

SUBF32 RaH, RbH, RcH
SUBF32 RaH, #16FHi, RbH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
SUBF32 RdH, ReH, RfH \(\parallel\) MOV32 mem32, RaH — 32-bit Floating-Point Subtraction with Parallel Move

### Operands

- **RdH**: Floating-point destination register (R0H to R7H) for the SUBF32 operation
- **ReH**: Floating-point source register (R0H to R7H) for the SUBF32 operation
- **RfH**: Floating-point source register (R0H to R7H) for the SUBF32 operation
- **mem32**: Pointer to 32-bit destination memory location for the MOV32 operation
- **RaH**: Floating-point source register (R0H to R7H) for the MOV32 operation

### Opcode

- **LSW**: 1110 0000 0010 fffe
- **MSW**: eedd daaa mem32

### Description

Subtract the contents of two floating-point registers and move from a floating-point register to memory.

\[ \text{RdH} = \text{ReH} - \text{RfH}, \]

\[ [\text{mem32}] = \text{RaH} \]

### Flags

This instruction modifies the following flags in the STF register:

- **TF**: No
- **ZI**: No
- **NI**: No
- **ZF**: No
- **NF**: No
- **LUF**: Yes
- **LVF**: Yes

The STF register flags are modified as follows:

- **LUF** = 1 if SUBF32 generates an underflow condition.
- **LVF** = 1 if SUBF32 generates an overflow condition.

### Pipeline

SUBF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

```
SUBF32 RdH, ReH, RfH ; 2 pipeline cycles (2p)
\| MOV32 mem32, RaH ; 1 cycle
\| NOP ; <-- MOV32 completes, mem32 updated
\| NOP ; <-- ADDF32 completes, RdH updated
```

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

### Example

```
ADDF32 R3H, R6H, R4H ; (A) R3H = R6H + R4H and R7H = I3
\| MOV32 R7H, *-SP[2] ;
\| --- R7H valid
SUBF32 R6H, R6H, R4H ; (B) R6H = R6H - R4H
\| --- ADDF32 (A) completes, R3H valid
SUBF32 R3H, R1H, R7H ; (C) R3H = R1H - R7H and store R3H (A)
\| MOV32 ++XAR5[2], R3H ;
\| --- SUBF32 (B) completes, R6H valid
\| --- MOV32 completes, (A) stored
ADDF32 R4H, R7H, R1H ; R4H = D = R7H + R1H and store R6H (B)
\| MOV32 ++XAR5[6], R6H ;
\| --- SUBF32 (C) completes, R3H valid
\| --- MOV32 completes, (B) stored
MOV32 ++XAR5[0], R3H ; store R3H (C)
\| --- MOV32 completes, (C) stored
\| --- ADDF32 (D) completes, R4H valid
MOV32 ++XAR5[4], R4H ; store R4H (D)
\| --- MOV32 completes, (D) stored
```
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH — 32-bit Floating-Point Subtraction with Parallel Move

See also

- SUBF32 RaH, RbH, RcH
- SUBF32 RaH, #16FHi, RbH
- SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
- MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
**SWAPF RaH, RbH{, CNDF} — Conditional Swap**

### Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>floating-point register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>floating-point register (R0H to R7H)</td>
</tr>
<tr>
<td>CNDF</td>
<td>condition tested</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1110 CNDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

### Description

Conditional swap of RaH and RbH.

\[
\text{if } (\text{CNDF} == \text{true}) \text{ swap } \text{RaH} \text{ and } \text{RbH}
\]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode (1)</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.

(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected

### Pipeline

This is a single-cycle instruction.

### Example

; find the largest element and put it in R1H

```assembly
MOV L XAR1, #0xB000 ;
MOV32 R1H, *XAR1 ; Initialize R1H
.align 2

NOP
RPTB LOOP_END, #(10-1); Execute the block 10 times
MOV32 R2H, *XAR1++ ; Update R2H with next element
CMPPF32 R2H, R1H ; Compare R2H with R1H
SWAPF R1H, R2H, GT ; Swap R1H and R2H if R2 > R1
NOP ; For minimum repeat block size
NOP ; For minimum repeat block size
LOOP_END:
```
### Test STF Register Flag Condition

**Operands**

<table>
<thead>
<tr>
<th>CNDF</th>
<th>condition to test</th>
</tr>
</thead>
</table>

**Opcode**

```
LSW: 1110 0101 1000 CNDF
```

**Description**

Test the floating-point condition and if true, set the TF flag. If the condition is false, clear the TF flag. This is useful for temporarily storing a condition for later use.

```c
if (CNDF == true) TF = 1; else TF = 0;
```

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode (1)</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.

(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>Ni</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

```
TF = 0; if (CNDF == true) TF = 1;
```

Note: If (CNDF == UNC or UNCF), the TF flag will be set to 1.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
CMF32 R0H, #0.0 ; Compare R0H against 0
TESTTF LT ; Set TF if R0H less than 0 (NF == 0)
ABS R0H, R0H ; Get the absolute value of R0H

; Perform calculations based on ABS R0H
MOVST0 TF ; Copy TF to TC in ST0
SBF End, NTC ; Branch to end if TF was not set
NEG32 R0H, R0H
End
```

**See also**
UI16TOF32 RaH, mem16  

**Convert unsigned 16-bit integer to 32-bit floating-point value**

### Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>pointer to 16-bit source memory location</td>
</tr>
</tbody>
</table>

### Opcode

LSW: 1110 0010 1100 0100  
MSW: 0000 0aaa mem16

### Description

RaH = UI16ToF32[mem16]

### Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
UI16TOF32 RaH, mem16 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
      <-- UI16TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

### Example

```
; float32 y, m, b;
; AdcRegs.RESULT0 is an unsigned int
; Calculate: y = (float)AdcRegs.RESULT0 * m + b;
; MOVW DP @0x01C4
UI16TOF32 R0H, @8 ; R0H = (float)AdcRegs.RESULT0
MOV32 R1H, *[SP]{6} ; R1H = M
      <-- Conversion complete, R0H valid
MPYF32 R0H, R1H, R0H ; R0H = (float)X * M
MOV32 R1H, *[SP]{8} ; R1H = B
      <-- MPYF32 complete, R0H valid
ADDF32 R0H, R0H, R1H ; R0H = Y = (float)X * M + B
NOP ; <-- ADDF32 complete, R0H valid
MOV32 *-[SP], R0H ; Store Y
```

### See also

- F32TOI16 RaH, RbH
- F32TOI16R RaH, RbH
- F32TOU16 RaH, RbH
- F32TOU16R RaH, RbH
- I16TOF32 RaH, RbH
- I16TOF32 RaH, mem16
- UI16TOF32 RaH, RbH
UI16TOF32 RaH, RbH — Convert unsigned 16-bit integer to 32-bit floating-point value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 1111
MSW: 0000 0000 00bb baaa

Description

RaH = UI16ToF32[RbH]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

UI16TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
     ; <-- UI16TOF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

MOVX R5H, #0x800F ; R5H[15:0] = 32783 (0x800F)
UI16TOF32 R6H, R5H ; R6H = UI16TOF32 (R5H[15:0])
NOP ; 1 cycle delay for UI16TOF32 to complete
     ; R6H = 32783.0 (0x47000F00)

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI32TOF32 RaH, mem32 — Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value

UI32TOF32 RaH, mem32  Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>pointer to 32-bit source memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 0100
MSW: 0000 0aaa mem32

Description

RaH = UI32ToF32[mem32]

Flags

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

UI32TOF32 RaH, mem32 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay non-conflicting instruction
    ; <-- UI32TOF32 completes, RaH updated
NOP

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

; unsigned long X
; float Y, M, B
; ...
; Calculate Y = (float)X * M + B
;
    UI32TOF32 R0H, *-SP[2] ; R0H = (float)X
    MOV32 R1H, *-SP[6] ; R1H = M
    ; <-- Conversion complete, R0H valid
    MPYF32 R0H, R1H, R0H ; R0H = (float)X * M
    MOV32 R1H, *-SP[8] ; R1H = B
    ; <-- MPYF32 complete, R0H valid
    ADDF32 R0H, R0H, R1H ; R0H = Y = (float)X * M + B
    NOP
    ; <-- ADDF32 complete, R0H valid
    MOV32 *[-SP], R0H ; Store Y

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, mem32
I32TOF32 RaH, RbH
UI32TOF32 RaH, RbH
UI32TOF32 RaH, RbH  

**Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value**

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

- **LSW:** 1110 0110 1000 1011
- **MSW:** 0000 0000 00bb baaa

**Description**

RaH = UI32ToF32[RbH]

**Flags**

This instruction does not affect any flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a 2 pipeline cycle (2p) instruction. That is:

```plaintext
UI32TOF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
    ; <-- UI32TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

**Example**

```plaintext
MOVIZ R3H, #0x8000 ; R3H[31:16] = 0x8000
MOVX R3H, #0x1111 ; R3H[15:0] = 0x1111
    ; R3H = 2147488017
UI32TOF32 R4H, R3H ; R4H = UI32TOF32 (R3H)
NOP ; 1 cycle delay for UI32TOF32 to complete
    ; R4H = 2147488017.0 (0x4F000011)
```

**See also**

- F32TOI32 RaH, RbH
- F32TOUI32 RaH, RbH
- I32TOF32 RaH, mem32
- I32TOF32 RaH, RbH
- UI32TOF32 RaH, mem32
**ZERO RaH — Zero the Floating-Point Register RaH**

**Operands**

| RaH | floating-point register (R0H to R7H) |

**Opcode**

LSW: 1110 0101 1001 0aaa

**Description**

Zero the indicated floating-point register:

RaH = 0

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
;for(i = 0; i < n; i++)
 {
 ; real += (x[2*i] * y[2*i]) - (x[2*i+1] * y[2*i+1]);
 ; imag += (x[2*i] * y[2*i+1]) + (x[2*i+1] * y[2*i]);
 }
 ;Assume AR7 = n-1
  ZERO R4H ; R4H = real = 0
  ZERO R5H ; R5H = imag = 0
 LOOP
  MOV AL, AR7
  MOV ACC, AL << 2
  MOV AR0, ACC
  MOV32 R0H, +XAR4[AR0] ; R0H = x[2*i]
  MOV32 R1H, +XAR5[AR0] ; R1H = y[2*i]
  ADD AR0, #2
  MPYF32 R6H, R0H, R1H; ; R6H = x[2*i] * y[2*i]
  || MOV32 R2H, +XAR4[AR0] ; R2H = x[2*i+1]
  || MPYF32 R1H, R1H, R2H ; R1H = y[2*i+1] * x[2*i+2]
  || MOV32 R3H, +XAR5[AR0] ; R3H = y[2*i+1]
  || MPYF32 R2H, R2H, R3H ; R2H = x[2*i+1] * y[2*i+1]
  || ADDF32 R4H, R4H, R6H ; R4H += x[2*i] * y[2*i+1]
  || MPYF32 R0H, R0H, R3H ; R0H = x[2*i] * y[2*i+1]
  || ADDF32 R5H, R5H, R1H ; R5H += y[2*i] * x[2*i+2]
  || SUBF32 R4H, R4H, R2H ; R4H -= x[2*i+1] * y[2*i+1]
  || ADDF32 R5H, R5H, R0H ; R5H += x[2*i] * y[2*i+1]
  BANZ LOOP , AR7--
```

**See also**

ZEROA
OPERANDS

none

OPCODE

LSW: 1110 0101 0110 0011

DESCRIPTION

Zero all floating-point registers:

ROH = 0
RIH = 0
R2H = 0
R3H = 0
R4H = 0
R5H = 0
R6H = 0
R7H = 0

FLAGS

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

PIPELINE

This is a single-cycle instruction.

EXAMPLE

;for(i = 0; i < n; i++)
;{
;  real += (x[2*i] * y[2*i]) - (x[2*i+1] * y[2*i+1]);
;  imag += (x[2*i] * y[2*i+1]) + (x[2*i+1] * y[2*i]);
;}
;Assume AR7 = n-1
ZEROA ; Clear all RaH registers

LOOP
  MOV AL, AR7
  MOV ACC, AL << 2
  MOV AR0, ACC
  MOV32 R0H, +XAR4[AR0] ; R0H = x[2*i]
  MOV32 R1H, +XAR5[AR0] ; R1H = y[2*i]
  ADD AR0,#2
  MPYF32 R6H, R0H, R1H; ; R6H = x[2*i] * y[2*i]
  MPYF32 R2H, R0H, R1H; ; R2H = x[2*i+1] * y[2*i+1]
  MPYF32 R1H, R1H, R2H ; R1H = y[2*i] * x[2*i+2]
  MPYF32 R3H, R2H, R3H ; R3H = y[2*i+1]
  MPYF32 R0H, R0H, R3H ; R0H = x[2*i] * y[2*i+1]
  ADDF32 R4H, R4H, R6H ; R4H += x[2*i] * y[2*i]
  ADDF32 R5H, R5H, R5H ; R5H += y[2*i] * x[2*i+2]
  SUBF32 R4H, R4H, R2H ; R4H -= x[2*i+1] * y[2*i+1]
  ADDF32 R5H, R5H, R0H ; R5H += x[2*i] * y[2*i+1]
  BANZ LOOP , AR7--

SEE ALSO

ZEROA
MOV32 RaL, mem32{, CNDF} — Conditional 32-bit Move

**Operands**

<table>
<thead>
<tr>
<th>RaL</th>
<th>Floating-point destination register (R0L to R7L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
<tr>
<td>CNDF</td>
<td>optional condition.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 1001 CNDF
MSW: 0000 0aaa mem32

**Description**

If the condition is true, then move the contents of memory referenced by mem32 to floating-point register indicated by RaL.

if(CNDF == true) RaH = unchanged, RaL = [mem32]

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode (1)</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.

(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

if(CNDF == UNCF)
{
    NF = RxH(31);
    ZF = 0;
    if(RaH(30:20) == 0)
    {
        ZF = 1; NF = 0;
    }
    else
    {
        ZI = 0;
    }
} else
No flags modified;

**Pipeline**

This is a single-cycle instruction.
**MOVDD32 RaL,mem32**  
*Move From Register To Memory 32-bit Move*

**Operands**

<table>
<thead>
<tr>
<th>RaL</th>
<th>Floating-point destination register (R0L to R7L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 0100 0010  
MSW: 0000 0aaa mem32

**Description**

RaH = [mem32], RaL = unchanged, [mem32+4] = [mem32].

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

NF = RxH(31);  
ZF = 0;  
if(RaH{30:20} == 0)  
{ ZF = 1; NF = 0; }  
if(RaL{31:0} != 0)  
ZI = 0;

**Pipeline**

This is a single-cycle instruction.
**MOVDD32 RaH,mem32** — Move From Register To Memory 32-bit Move

---

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 0100 0011  
MSW: 0000 0aaa mem32

**Description**

RaH = [mem32], RaL = unchanged, [mem32+4] = [mem32].

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0)
  { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0)
  ZI = 1;

**Pipeline**

This is a single-cycle instruction.
MOV32 mem32,RaN — Move From Memory to Register 32-bit Move

Operands

<table>
<thead>
<tr>
<th>RaL</th>
<th>Floating-point source register (R0L to R7L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 0010
MSW: 0000 0aaa mem32

Description

[mem32] = RaL.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

Pipeline

This is a single-cycle instruction.
MOVIX RaL,#16I — Load the Upper 16-bits of a 32-bit Floating-Point Register

Operands

<table>
<thead>
<tr>
<th>RaL</th>
<th>Floating-point destination register (R0L to R7L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16I</td>
<td>A 16-bit immediate value.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1001 0000 0III
MSW: IIII IIII IIII Iaaa

Description

RaL(15:0) = unchanged RaL(31:16) = #16I.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

Pipeline

This is a single-cycle instruction.
**MOVXI RaL, #16I**  
*Load the Lower 16-bits of a 32-bit Floating-Point Register*

**Operands**

<table>
<thead>
<tr>
<th>RaL</th>
<th>Floating-point destination register (R0L to R7L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16I</td>
<td>A 16-bit immediate value.</td>
</tr>
</tbody>
</table>

**Opode**

LSW: 1110 1001 0000 IIII  
MSW: IIII IIII IIII Iaaa

**Description**

RaL(15:0) = #16I  
RaL(31:16) = unchanged.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

No flags affected.

**Pipeline**

This is a single-cycle instruction.
**Operands**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>Floating-point destination register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaL</td>
<td>Floating-point destination register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0011 1000 fffe  
MSW: eedd daaa mem32

**Description**

Multiply the contents of two floating-point registers and load another  
Rd = Re * Rf, RaL = [mem32].  
The destination register for the MOV32 cannot be the same as the destination registers for the MPYF64.

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MPY operation generated an underflow condition.  
The LVF flag is set to 1 if the MPY operation generated an overflow condition.  
The ZI, ZF, NF flags are modified only by the respective MOV32 operations. Refer to earlier descriptions of MOV32 operations for flag setting details.

**Pipeline**

MPYF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
MPYF64 Rd,Re,Rf || MOV32 mem32,RaL — 64-bit Floating-Point Multiply with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Rd</th>
<th>Floating-point destination register for the MPYF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaL</td>
<td>Floating-point source register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0000 1000 fffe
MSW: eedd daaa mem32

Description
Multiply the contents of two floating-point registers and write from Register to memory.

\[ Rd = Re \times Rf, \ [\text{mem32}] = RaL \]

Flags
This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MPY operation generated an underflow condition.
The LVF flag is set to 1 if the MPY operation generated an overflow condition.

Pipeline

MPYF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
ADD64 Rd,Re,Rf || MOV32 RaL, mem32  
64-bit Floating-Point Addition with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Rd</th>
<th>Floating-point destination register for the ADDF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaL</td>
<td>Floating-point destination register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1001 fffe  
MSW: eedd daaa mem32

Description

Perform an ADDF64 and a MOV32 in parallel.  
Rd = Re + Rf, RaL = [mem32]  
The destination register for the MOV32 cannot be the same as the destination registers for the ADDF64.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the ADD operation generated an underflow condition.  
The LVF flag is set to 1 if the ADD operation generated an overflow condition.  
The ZI, ZF, NF flags are modified only by the respective MOV32 operations. Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

ADD64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
ADD6F4 Rd,Re,Rf || MOV32 mem32, RaL  64-bit Floating-Point Addition with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>Floating-point destination register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaL</td>
<td>Floating-point source register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0000 1001 fffe  
MSW: eedd daaa mem32

Description

Perform an ADDF64 and a MOV32 in parallel.  
Rd = Re + Rf, [mem32] = RaL

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the ADD operation generated an underflow condition.  
The LVF flag is set to 1 if the ADD operation generated an overflow condition.

Pipeline

ADD6F4 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
SUBF64 Rd,Re,Rf ||MOV32 RaL,mem32  — 64-bit Floating-Point Subtraction with Parallel Move

SUBF64 Rd,Re,Rf ||MOV32 RaL,mem32  64-bit Floating-Point Subtraction with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Rd</th>
<th>Floating-point destination register for the SUBF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1010 fffe  
MSW: eedd daaa mem32

Description

Perform an SUBF64 and a MOV32 in parallel.

Rd = Re - Rf, RaL = [mem32]

The destination register for the MOV32 cannot be the same as the destination registers for the SUBF64.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the SUB operation generated an underflow condition.  
The LVF flag is set to 1 if the SUB operation generated an overflow condition.  
The ZI, ZF, NF flags are modified only by the respective MOV32 operations. Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

SUBF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
SUBF64 Rd,Re,Rf ||MOV32 mem32, RaL — 64-bit Floating-Point Subtraction with Parallel Move

### Operands

<table>
<thead>
<tr>
<th>Rd</th>
<th>Floating-point destination register for the SUBF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaL</td>
<td>Floating-point source register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0000 1010 fffe</td>
<td>eedd daaa mem32</td>
</tr>
</tbody>
</table>

### Description

Perform an SUBF64 and a MOV32 in parallel.

\[ Rd = Re - Rf, \ [\text{mem32}] = \text{RaL} \]

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the SUB operation generated an underflow condition. The LVF flag is set to 1 if the SUB operation generated an overflow condition.

### Pipeline

SUBF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
MACF64 R3,R2,Rd,Re,Rf MOV32 RaL, mem32 64-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>Floating-point destination/source register R3 for the add operation</td>
</tr>
<tr>
<td>R2</td>
<td>Floating-point source register R2 for the add operation</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point destination register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>RaL</td>
<td>Floating-point destination register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>LSW: 1110 0011 1011 fffe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSW: eedd daaa mem32</td>
</tr>
</tbody>
</table>

Description

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF64.

\[ R3 = R3 + R2, \text{Rd} = \text{Re} \times \text{Rf}, \text{RaL} = \lbrack\text{mem32}\rbrack \]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MAC operation generated an underflow condition.
The LVF flag is set to 1 if the MAC operation generated an overflow condition.
The ZI, ZF, NF flags are modified only by the respective MOV32 operations. Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

MACF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
**MACF64 R7,R6,Rd,Re,Rf || MOV32 RaL, mem32 — 64-bit Floating-Point Multiply and Accumulate with Parallel Move**

**Operands**

<table>
<thead>
<tr>
<th>R7</th>
<th>Floating-point destination/source register R7 for the add operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R6</td>
<td>Floating-point source register R6 for the add operation</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point destination register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>RaL</td>
<td>Floating-point destination register (R0L to R7L)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW: 1110 0011 1110 fffe</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: eedd daaa mem32</td>
</tr>
</tbody>
</table>

**Description**

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF64.

\[
R7 = R7 + R6, \quad Rd = Re \times Rf, \quad RaL = \text{[mem32]} \]

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MAC operation generated an underflow condition. The LVF flag is set to 1 if the MAC operation generated an overflow condition. The ZI, ZF, NF flags are modified only by the respective MOV32 operations. Refer to earlier descriptions of MOV32 operations for flag setting details.

**Pipeline**

MACF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
MPYF64 Rd,Re,Rf | MOV32 RaH,mem32 — 64-bit Floating-Point Multiply with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Rd</th>
<th>Floating-point destination register for the MPYF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 0100 fffe
MSW: eedd daaa mem32

Description

Perform a MPYF64 and a MOV32 in parallel.

\[ \text{Rd} = \text{Re} \times \text{Rf}, \text{RaH} = [\text{mem32}] \]

The destination register for the MOV32 cannot be the same as the destination registers for the MPYF64.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MPY operation generated an underflow condition.
The LVF flag is set to 1 if the MPY operation generated an overflow condition.
The ZI, NI, ZF, NF flags are modified only by the respective MOV32 operations.
Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

MPYF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
MPYF64 Rd,Re,Rf || MOV32 mem32, RaH — 64-bit Floating-Point Multiply with Parallel Move

**Operands**

| Rd   | Floating-point destination register for the MPYF64 (R0 to R7) |
| Re   | Floating-point source register for the MPYF64 (R0 to R7) |
| Rf   | Floating-point source register for the MPYF64 (R0 to R7) |
| RaH  | Floating-point source register (R0H to R7H) |
| mem32| Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location |

**Opcode**

LSW: 1110 0000 0100 fffe
MSW: eedd daaa mem32

**Description**

Perform a MPYF64 and a MOV32 in parallel.

Rd = Re * Rf, [mem32] = RaH

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MPY operation generated an underflow condition. The LVF flag is set to 1 if the MPY operation generated an overflow condition.

**Pipeline**

MPYF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
ADDF64 Rd,Re,Rf || MOV32 RaH,mem32 — 64-bit Floating-Point Addition with Parallel Move

ADDF64 Rd,Re,Rf || MOV32 RaH,mem32  64-bit Floating-Point Addition with Parallel Move

Operands

| Rd       | Floating-point destination register for the ADDF64 (R0 to R7) |
| Re       | Floating-point source register for the ADDF64 (R0 to R7)     |
| Rf       | Floating-point source register for the ADDF64 (R0 to R7)     |
| RaH      | Floating-point destination register (R0H to R7H)             |
| mem32    | Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location |

Opcode

LSW: 1110 0011 0101 fffe
MSW: eedd daaa mem32

Description

Perform a ADDF64 and a MOV32 in parallel.

Rd = Re + Rf, RaH = [mem32]

The destination register for the MOV32 cannot be the same as the destination registers for the ADDF64.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the ADD operation generated an underflow condition. The LF flag is set to 1 if the ADD operation generated an overflow condition. The ZI, NI, ZF, NF flags are modified only by the respective MOV32 operations. Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

ADDF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
ADDF64 Rd,Re,Rf ||MOV32 mem32, RaH  

64-bit Floating-Point Addition with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Rd</th>
<th>Floating-point destination register for the ADDF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0000 0101 fffe
MSW: eedd daaa mem32

Description

Perform a ADDF64 and a MOV32 in parallel.

\[ Rd = Re + Rf, [mem32] = RaH \]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the ADD operation generated an underflow condition.
The LVF flag is set to 1 if the ADD operation generated an overflow condition.

Pipeline

ADDF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
SUBF64 Rd,Re,Rf | MOV32 RaH,mem32  —  64-bit Floating-Point Subtraction with Parallel Move

SUBF64 Rd,Re,Rf | MOV32 RaH,mem32  64-bit Floating-Point Subtraction with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>Floating-point destination register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 0110 fffe
MSW: eedd daaa mem32

Description

Perform a SUBF64 and a MOV32 in parallel.

\[ Rd = Re - Rf, RaH = [\text{mem32}] \]

The destination register for the MOV32 cannot be the same as the destination registers for the SUBF64

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the SUB operation generated an underflow condition.
The LVF flag is set to 1 if the SUB operation generated an overflow condition.
The ZI, NI, ZF, NF flags are modified only by the respective MOV32 operations.
Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

SUBF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
**SUBF64 Rd,Re,Rf LINUX mem32, RaH — 64-bit Floating-Point Subtraction with Parallel Move**

Subtract Rd = Re - Rf, and store the result in a memory location.

**Operands**

<table>
<thead>
<tr>
<th>Rd</th>
<th>Floating-point destination register for the SUBF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0000 0110 fffe  
MSW: eedd daaa mem32

**Description**

Perform a SUBF64 and a MOV32 in parallel.

```
Rd = Re - Rf, [mem32] = RaH
```

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the SUB operation generated an underflow condition. The LVF flag is set to 1 if the SUB operation generated an overflow condition.

**Pipeline**

SUBF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
MACF64 R3,R2,Rd,Re,Rf MOV32 RaH, mem32 — 64-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>Floating-point destination/source register R3 for the add operation</td>
</tr>
<tr>
<td>R2</td>
<td>Floating-point source register R2 for the add operation</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point destination register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 0111 fffe
MSW: eedd daaa mem32

Description

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF64.

\[ R3 = R3 + R2, \quad Rd = Re \times Rf, \quad RaH = [mem32] \]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MAC operation generated an underflow condition.
The LVF flag is set to 1 if the MAC operation generated an overflow condition.
The ZI, NI, ZF, NF flags are modified only by the respective MOV32 operations.
Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

MACF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
MACF64 R7,R6,Rd,Re,Rf || MOV32 RaH, mem32  
64-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7</td>
<td>Floating-point destination/source register R7 for the add operation</td>
</tr>
<tr>
<td>R6</td>
<td>Floating-point source register R6 for the add operation</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point destination register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register (R0 to R7) for the multiply operation</td>
</tr>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1101 fffe  
MSW: eedd daaa mem32

Description

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF64.

\[ R7 = R7 + R6, \quad Rd = Re \times Rf, \quad RaH = [mem32] \]

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if the MAC operation generated an underflow condition.
The LVF flag is set to 1 if the MAC operation generated an overflow condition.
The ZI, NI, ZF, NF flags are modified only by the respective MOV32 operations.
Refer to earlier descriptions of MOV32 operations for flag setting details.

Pipeline

MACF64 takes 3 pipeline-cycles (3p) and MOV32 takes a single cycle.
MPYF64 Ra,Rb,Rc || ADDF64 Rd,Re,Rf — 64-bit Floating-Point Multiply with Parallel Addition

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Floating-point destination register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rc</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point destination register for ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register for ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for ADDF64 (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 1100 00ff  
MSW: feee dddc cbb baaa

Description

Perform a MPYF64 and a ADDF64 in parallel.

Ra = Rb * Rc, Rd = Re + Rf

The destination register for the ADDF64 cannot be the same as the destination registers for the MPYF64

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if either the MPY operation or ADD operation generated an underflow condition.

The LVF flag is set to 1 if either the MPY operation or ADD operation generated an overflow condition.

Pipeline

MPYF64 takes 3 pipeline-cycles (3p) and ADDF64 takes 3 pipeline-cycles (3p)
MPYF64 Ra,Rb,Rc |SUBF64 Rd,Re,Rf  64-bit Floating-Point Multiply with Parallel Subtraction

Operands

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Floating-point destination register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rc</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point destination register for SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Re</td>
<td>Floating-point source register for SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rf</td>
<td>Floating-point source register for SUBF64 (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 0111 1101 00ff
- MSW: feee dddc ccbb baaa

Description

Perform a MPYF64 and a SUBF64 in parallel.

- Ra = Rb * Rc, Rd = Re - Rf
- The destination register for the SUBF64 cannot be the same as the destination registers for the MPYF64

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if either the MPY operation or SUB operation generated an underflow condition.
The LVF flag is set to 1 if either the MPY operation or SUB operation generated an overflow condition.

Pipeline

- MPYF64 takes 3 pipeline-cycles (3p) and SUBF64 takes 3 pipeline-cycles (3p)
MPYF64 Ra,Rb,Rc — 64-bit Floating-Point Multiply

Operands

- **Ra**: Floating-point destination register for the MPYF64 (R0 to R7)
- **Rb**: Floating-point source register for the MPYF64 (R0 to R7)
- **Rc**: Floating-point source register for the MPYF64 (R0 to R7)

Opcode

LSW: 1110 0111 1000 0000
MSW: 0000 000c ccbb baaa

Description

Ra = Rb * Rc

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if MPY operation generated an underflow condition.
The LVF flag is set to 1 if MPY operation generated an overflow condition.

Pipeline

MPYF64 takes 3 pipeline-cycles (3p)
ADDF64 Ra,Rb,Rc  

**64-bit Floating-Point Addition**

### Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Floating-point destination register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rc</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
</tbody>
</table>

### Opcode

LSW: 1110 0111 1001 0000  
MSW: 0000 000c ccbb baaa

### Description

Ra = Rb + Rc

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- The LUF flag is set to 1 if ADD operation generated an underflow condition.
- The LVF flag is set to 1 if ADD operation generated an overflow condition.

### Pipeline

ADDF64 takes 3 pipeline-cycles (3p)
SUBF64 Ra,Rb,Rc — 64-bit Floating-Point Subtraction

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register for the SUBF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rc</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 1010 0000
MSW: 0000 000c ccbb baaa

Description

Ra = Rb - Rc

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if SUB operation generated an underflow condition.
The LVF flag is set to 1 if SUB operation generated an overflow condition.

Pipeline

SUBF64 takes 3 pipeline-cycles (3p)
MPYF64 Ra,Rb,#16F OR MPYF64 Ra,#16F, Rb  64-bit Floating-Point Multiply

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Floating-point destination register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register for the MPYF64 (R0 to R7)</td>
</tr>
<tr>
<td>#16F</td>
<td>16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1001 01II IIII
MSW: IIII IIII IIbb baaa

Description

Ra = Rb * #16F:0

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if MPY operation generated an underflow condition. The LVF flag is set to 1 if MPY operation generated an overflow condition.

Pipeline

MPYF64 takes 3 pipeline-cycles (3p)
ADDF64  Ra,Rb,#16F  OR  ADDF64  Ra,#16F, Rb  

--- 64-bit Floating-Point Addition ---

**Operands**

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register for the ADDF64 (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register for the ADDF64 (R0 to R7)</td>
</tr>
<tr>
<td>#16F</td>
<td>16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 1001 10II IIII  
MSW: IIII IIII IIbb baaa

**Description**

Ra = Rb + #16F:0

**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if ADD operation generated an underflow condition. The LVF flag is set to 1 if ADD operation generated an overflow condition.

**Pipeline**

ADDF64 takes 3 pipeline-cycles (3p)
SUBF64 Ra,#16F,Rb  64-bit Floating-Point Subtraction

Operands

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Floating-point destination register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register for the SUBF64 (R0 to R7)</td>
</tr>
<tr>
<td>#16F</td>
<td>16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1001 11II IIII
MSW: IIII IIII IIbb baaa

Description

Ra = #16F:0 - Rb

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The LUF flag is set to 1 if SUB operation generated an underflow condition.
The LVF flag is set to 1 if SUB operation generated an overflow condition.

Pipeline

SUBF64 takes 3 pipeline-cycles (3p)
**CMPF64 Ra, Rb — 64-bit Floating-Point Compare for Equal, Less Than or Greater Than**

### Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point source register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

### Opcode

- **LSW:** 1110 0110 1001 1000
- **MSW:** 0000 0000 00bb baaa

### Description

Set ZF and NF flags on the result of Ra - Rb. The CMPF64 instruction is performed as a logical compare operation. This is possible because of the IEEE format offsetting the exponent. Basically the bigger the binary number, the bigger the floating-point value.

### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- If(Ra == Rb/#16F/#0.0) ZF=1, NF=0
- If(Ra > Rb/#16F/#0.0) ZF=0, NF=0
- If(Ra < Rb/#16F/#0.0) ZF=0, NF=1

### Pipeline

This is a single cycle instruction.
CMPF64 Ra,#16F  

64-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point source register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16F</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 1001 0001 0III
MSW: IIII IIII IIII Iaaa

Description

Set ZF and NF flags on the result of (Ra - #16F:0). The CMPF64 instruction is performed as a logical compare operation. This is possible because of the IEEE format offsetting the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

If(Ra == #16F) ZF=1, NF=0
If(Ra > #16F) ZF=0, NF=0
If(Ra < #16F) ZF=0, NF=1

Pipeline

This is a single cycle instruction.
CMPF64 Ra,#0.0 — 64-bit Floating-Point Compare for Equal, Less Than or Greater Than

CMPF64 Ra,#0.0  64-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point source register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0.0</td>
<td>zero</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1011 0aaa

Description

Set ZF and NF flags on the result of (Ra - #0.0). The CMPF64 instruction is performed as a logical compare operation. This is possible because of the IEEE format offsetting the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

If (Ra == #0.0) ZF=1, NF=0
If (Ra > #0.0) ZF=0, NF=0
If (Ra < #0.0) ZF=0, NF=1

Pipeline

This is a single cycle instruction.
## MAXF64 Ra, Rb

### 64-bit Floating-Point Maximum

#### Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point source register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

#### Opcode

- **LSW**: 1110 0110 1001 1010
- **MSW**: 0000 0000 00bb baaa

#### Description

$$\text{if}(Ra < Rb) \text{ Ra} = \text{Rb}$$

#### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- **If (Ra == Rb)**: ZF=1, NF=0
- **If (Ra > Rb)**: ZF=0, NF=0
- **If (Ra < Rb)**: ZF=0, NF=1

#### Pipeline

MAXF64 takes 2 pipeline-cycles (2p).
MAXF64 Ra, Rb ||MOV64 Rc,Rd — 64-bit Floating-Point Maximum with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>floating-point source/destination register for the MAXF64 operation (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register for the MAXF64 operation (R0 to R7)</td>
</tr>
<tr>
<td>Rc</td>
<td>Floating-point destination register for the MOV64 operation (R0 to R7)</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point source register for the MOV64 operation (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 1110
MSW: 0000 dddc cccb baaa

Description

if(Ra < Rb) { Ra = Rb; Rc = Rd; }

The destination register for the MOV64 cannot be the same as the destination registers for the MAXF64

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

If(Ra == Rb) ZF=1, NF=0
If(Ra > Rb) ZF=0, NF=0
If(Ra < Rb) ZF=0, NF=1

Pipeline

MAXF64 in parallel with MOV64 takes 2 pipeline-cycles (2p).
### MAXF64 Ra, #16F  
64-bit Floating-Point Maximum

#### Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>floating-point source/destination register for the MAXF64 operation (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16F</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

#### Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 1001 0010 0III</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: IIII IIII IIII Iaaa</td>
</tr>
</tbody>
</table>

#### Description

if(Ra < #16F:0) Ra = #16F:0

#### Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

If(Ra == #16F) ZF=1, NF=0
If(Ra > #16F) ZF=0, NF=0
If(Ra < #16F) ZF=0, NF=1

#### Pipeline

This instruction takes 2 pipeline-cycles (2p).
MINF64 Ra, Rb — 64-bit Floating-Point Minimum

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>floating-point source/destination register for the MINF64 operation (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 1011
MSW: 0000 0000 00bb baaa

Description

if(Ra > Rb) Ra = Rb

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

If(Ra == Rb) ZF=1, NF=0
If(Ra > Rb) ZF=0, NF=0
If(Ra < Rb) ZF=0, NF=1

Pipeline

This instruction takes 2 pipeline-cycles (2p).
MINF64 Ra, Rb ∥ MOV64 Rc, Rd  

64-bit Floating-Point Minimum with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>floating-point source/destination register for the MINF64 operation (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register for the MINF64 operation (R0 to R7)</td>
</tr>
<tr>
<td>Rc</td>
<td>Floating-point destination register for the MOV64 operation (R0 to R7)</td>
</tr>
<tr>
<td>Rd</td>
<td>Floating-point source register for the MOV64 operation (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 1111
MSW: 0000 dddc cbbb baaa

Description

if(Ra > Rb) { Ra = Rb; Rc = Rd; }

The destination register for the MOV64 cannot be the same as the destination registers for the MINF64

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

If(Ra == Rb) ZF=1, NF=0
If(Ra > Rb) ZF=0, NF=0
If(Ra < Rb) ZF=0, NF=1

Pipeline

MINF64 in parallel with MOV64 takes 2 pipeline-cycles (2p).
MINF64 Ra, #16F — 64-bit Floating-Point Minimum

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>floating-point source/destination register for the MINF64 operation (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16F</td>
<td>A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.</td>
</tr>
</tbody>
</table>

Opcode

| LSW: 1110 1001 0011 0III |
| MSW: IIII IIII IIII Iaaa |

Description

if(Ra > #16F:0) Ra = #16F:0

Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

If(Ra == #16F) ZF=1, NF=0
If(Ra > #16F) ZF=0, NF=0
If(Ra < #16F) ZF=0, NF=1

Pipeline

This instruction takes 2 pipeline-cycles (2p).
F64TOI32 RaH,Rb — Convert 64-bit Floating-Point Value to 32-bit Integer

**Operands**

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1000 0100  
MSW: 0000 0000 00bb baaa

**Description**

RaH = F64ToI32(Rb)

**Flags**

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This instruction takes 2 pipeline-cycles (2p).
F64TOUI32 RaH,Rb — Convert 64-bit Floating-Point Value to 32-bit Unsigned Integer

Operands

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 0110  
MSW: 0000 0000 00bb baaa

Description

RaH = F64ToUI32(Rb)

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
### I32TOF64 Ra,mem32  
**Convert 32-bit Integer to 64-bit Floating-Point Value**

#### Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

#### Opcode

<table>
<thead>
<tr>
<th>LSW:</th>
<th>1110 0010 1000 1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW:</td>
<td>0000 0aaa mem32</td>
</tr>
</tbody>
</table>

#### Description

Ra = I32ToF64[mem32]

#### Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Pipeline

This instruction takes 2 pipeline-cycles (2p).
I32TOF64 Ra,RbH — Convert 32-bit Integer to 64-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 0101
MSW: 0000 0000 00bb baaa

Description

Ra = I32ToF64(RbH)

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
UI32TOF64 Ra,mem32  —  Convert unsigned 32-bit Integer to 64-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 0101
MSW: 0000 0aaa mem32

Description

Ra = UI32ToF64[mem32]

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
F64TOI64 Ra,Rb — Convert 64-bit Floating-Point Value to 64-bit Integer

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 0100
MSW: 0000 0000 00bb baaa

Description

Ra = F64ToI64(Rb)

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>N1</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
F64TOUI64 Ra,Rb  —  Convert 64-bit Floating-Point Value to 64-bit unsigned Integer

**Operands**

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1000 0110  
MSW: 1000 0000 00bb baaa  

**Description**

Ra = F64ToUI64(Rb)

**Flags**

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This instruction takes 2 pipeline-cycles (2p).
**I64TOF64 Ra,Rb** — Convert 64-bit Integer to 64-bit Floating-Point Value

**Operands**

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: `1110 0110 1000 0101`  
MSW: `1000 0000 00bb baaa`

**Description**

Ra = I64ToF64(Rb)

**Flags**

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This instruction takes 2 pipeline-cycles (2p).
UI64TOF64 Ra,Rb  —  Convert 64-bit unsigned Integer to 64-bit Floating-Point Value

**Operands**

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1000 0111
MSW: 1000 0000 00bb baaa

**Description**

Ra = UI64ToF64(Rb)

**Flags**

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This instruction takes 2 pipeline-cycles (2p).
I64TOF64 Ra,Rb — Convert 64-bit Integer to 64-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Floating-point destination register (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>LSW: 1110 0110 1000 0101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSW: 1000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

Description

Ra = I64ToF64(Rb)

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
UI64TOF64 Ra,Rb — Convert 64-bit unsigned Integer to 64-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 0111
MSW: 1000 0000 00bb baaa

Description

Ra = UI64ToF64(Rb)

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
FRACF64 Ra,Rb — Fractional Portion of a 64-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1111 0001
MSW: 1000 0000 00bb baaa

Description

Returns in Ra the fractional portion of F64 value in Rb.

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
F64TOF32 RaH,Rb  —  Convert 64-bit Floating-Point Value to 32-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 0000
MSW: 0000 0000 00bb baaa

Description

RaH = F64ToF32(Rb)

(if RNDF32 == 1, round to nearest)

Flags

This instruction does not affect any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipeline-cycles (2p).
F32TOF64 Ra,RbH — Convert 32-bit Floating-Point Value to 64-bit Floating-Point Value

**Operands**

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1001 0001
MSW: 0000 0000 00bb baaa

**Description**

Ra = F32ToF64(RbH)

**Flags**

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

NF = RaH(31);
ZF = 0;
if(RaH(30:20) == 0)
  { ZF = 1; NF = 0; }

**Pipeline**

This instruction takes 1 cycle.
F32TOF64 Ra, mem32  

Convert 32-bit Floating-Point Value to 64-bit Floating-Point Value

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 1100
MSW: 0000 0aaa mem32

Description

Ra = F32ToF64[mem32]

Flags

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

NF = RaH(31);
ZF = 0;
if(RaH(30:20) == 0)
{ ZF = 1; NF = 0; }

Pipeline

This instruction takes 1 cycle.
### F32DTOF64 Ra, mem32

**Convert 32-bit Floating-Point Value to 64-bit Floating-Point Value**

**Operands**

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0010 0010 0001</td>
<td>0000 0aaa mem32</td>
</tr>
</tbody>
</table>

**Description**

Ra = F32ToF64[mem32],
[mem32+2] = [mem32]

**Flags**

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

NF = RaH(31);
ZF = 0;
if (RaH(30:20) == 0)
  { ZF = 1; NF = 0; }

**Pipeline**

This instruction takes 1 cycle.
ABSF64 Ra, Rb  

64-bit Floating-Point Absolute Value

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 1001
MSW: 0000 0000 00bb baaa

Description

if( Rb < 0 ) { Ra = -Rb }
else { Ra = Rb }

Flags

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

NF = 0; ZF = 0;
if(RaH(30:20) == 0)
  ZF = 1;

Pipeline

This instruction takes 1 cycle.
NEGF64 Ra, Rb{, CNDF} — Conditional Negation

NEGF64 Ra, Rb{, CNDF}  Conditional Negation

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
<tr>
<td>CNDF</td>
<td>condition tested</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1011 CNDF
MSW: 0000 0000 00bb baaa

Description

if(CNDF == true) { Ra = -Rb }
else { Ra = Rb }

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

if(CNDF == UNCF)
{
  NF = RaH(31); ZF = 0;
  if(RaH(30:20) == 0)
  { ZF = 1; NF = 0; }
}
else
  No flags modified;

Pipeline

This instruction takes 1 cycle.
MOV64 Ra, Rb{, CNDF}  Conditional 64-bit Move

Operands

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Floating-point destination register (R0 to R7)</td>
</tr>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
<tr>
<td>CNDF</td>
<td>condition tested</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1101 CNDF
MSW: 0000 0000 00bb baaa

Description

if(CNDF == true) Ra = Rb

CNDF is one of the following conditions:

<table>
<thead>
<tr>
<th>Encode (1)</th>
<th>CNDF</th>
<th>Description</th>
<th>STF Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NEQ</td>
<td>Not equal to zero</td>
<td>ZF == 0</td>
</tr>
<tr>
<td>0001</td>
<td>EQ</td>
<td>Equal to zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than zero</td>
<td>ZF == 0 AND NF == 0</td>
</tr>
<tr>
<td>0011</td>
<td>GEQ</td>
<td>Greater than or equal to zero</td>
<td>NF == 0</td>
</tr>
<tr>
<td>0100</td>
<td>LT</td>
<td>Less than zero</td>
<td>NF == 1</td>
</tr>
<tr>
<td>0101</td>
<td>LEQ</td>
<td>Less than or equal to zero</td>
<td>ZF == 1 AND NF == 1</td>
</tr>
<tr>
<td>1010</td>
<td>TF</td>
<td>Test flag set</td>
<td>TF == 1</td>
</tr>
<tr>
<td>1011</td>
<td>NTF</td>
<td>Test flag not set</td>
<td>TF == 0</td>
</tr>
<tr>
<td>1100</td>
<td>LU</td>
<td>Latched underflow</td>
<td>LUF == 1</td>
</tr>
<tr>
<td>1101</td>
<td>LV</td>
<td>Latched overflow</td>
<td>LVF == 1</td>
</tr>
<tr>
<td>1110</td>
<td>UNC</td>
<td>Unconditional</td>
<td>None</td>
</tr>
<tr>
<td>1111</td>
<td>UNCF (2)</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

if(CNDF == UNCF)
{
    NF = RaH(31);  ZF = 0;
    if(RaH(30:20) == 0)
        { ZF = 1;  NF = 0; }

    NI = RaH(31);
    ZI = 0;
    if(Ra(63:0) == 0)
        ZI = 1;
}
else
    No flags modified.

Pipeline

This instruction takes 1 cycle.
EISQRTF64 Ra, Rb  —  64-bit Floating-Point Square-Root Reciprocal Approximation

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 0110 1001 0010
- MSW: 1000 0000 00bb baaa

Description

This operation generates an estimate of "1/sqrt(X)" in F64 format and then this value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

\[
Ye = \text{Estimate}(1/\sqrt{Xi});
Ye = Ye \times (1.5 - Ye \times Ye \times Xi / 2.0)
Ye = Ye \times (1.5 - Ye \times Ye \times Xi / 2.0)
\]

After about ~4 iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to F64 format. On iteration the mantissa bit accuracy approximately doubles. The EISQRTF64 operation will not generate a -ve, De-Norm or NaN value.

Ra = Estimate Of 1/sqrt(Rb)

Flags

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipe-line cycles (2p).
EINVF64 Ra, Rb  

64-bit Floating-Point Reciprocal Approximation

Operands

<table>
<thead>
<tr>
<th>Ra</th>
<th>Floating-point destination register (R0 to R7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>Floating-point source register (R0 to R7)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1001 0011
MSW: 1000 0000 00bb baaa

Description

This operation generates an estimate of "1/X" in F64 format and then this value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

\[
Ye = \text{Estimate}(1/X);
Ye = Ye \times (2.0 - Ye \times X)
Ye = Ye \times (2.0 - Ye \times X)
\]

After about ~4 iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to F64 format. On iteration the mantissa bit accuracy approximately doubles. The EINVF64 operation will not generate a -ve zero, De-Norm or NaN value.

Ra = Estimate Of 1/Rb

Flags

This instruction affects the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This instruction takes 2 pipe-line cycles (2p).
The C28x Viterbi, Complex Math and CRC Unit (VCU) is a fully programmable block which accelerates the performance of communications-based algorithms by up to a factor of 8X over C28x CPU alone. In addition to eliminating the need for a second processor to manage the communications link, the performance gains of the VCU provides headroom for future system growth and higher bit rates or, conversely, enables devices to operate at a lower MHz to reduce system cost and power consumption. This document provides an overview of the architectural structure and instruction set of the C28x VCU.

The VCU module described in this chapter is a Type 0/1 VCU. See the TMS320x28xx, 28xxx DSP Peripheral Reference Guide (SPRU566) for a list of all devices with a VCU module of the same type, to determine the differences between the types, and for a list of device-specific differences within a type. This document describes the architecture, pipeline, instruction set, and interrupts of the C28x+VCU.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Overview</td>
<td>339</td>
</tr>
<tr>
<td>3.2 Components of the C28x plus VCU</td>
<td>340</td>
</tr>
<tr>
<td>3.3 Emulation Logic</td>
<td>341</td>
</tr>
<tr>
<td>3.4 Register Set</td>
<td>344</td>
</tr>
<tr>
<td>3.5 Pipeline</td>
<td>351</td>
</tr>
<tr>
<td>3.6 Instruction Set</td>
<td>356</td>
</tr>
<tr>
<td>3.7 Rounding Mode</td>
<td>461</td>
</tr>
</tbody>
</table>
3.1 Overview

The C28x with VCU (C28x+VCU) processor extends the capabilities of the C28x fixed-point or floating-point CPU by adding registers and instructions to support the following algorithm types:

- **Viterbi decoding**
  
  Viterbi decoding is commonly used in baseband communications applications. The viterbi decode algorithm consists of three main parts: branch metric calculations, compare-select (viterbi butterfly) and a traceback operation. Table 3-1 shows a summary of the VCU performance for each of these operations.

<table>
<thead>
<tr>
<th>Viterbi Operation</th>
<th>VCU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Metric Calculation (code rate = 1/2)</td>
<td>1</td>
</tr>
<tr>
<td>Branch Metric Calculation (code rate = 1/3)</td>
<td>2p</td>
</tr>
<tr>
<td>Viterbi Butterfly (add-compare-select)</td>
<td>2</td>
</tr>
<tr>
<td>Traceback per Stage</td>
<td>3</td>
</tr>
</tbody>
</table>

(1) C28x CPU takes 15 cycles per butterfly.
(2) C28x CPU takes 22 cycles per stage.

- **Cyclic redundancy check (CRC)**

CRC algorithms provide a straightforward method for verifying data integrity over large data blocks, communication packets, or code sections. The C28x+VCU can perform 8-, 16-, and 32-bit CRCs. For example, the VCU can compute the CRC for a block length of 10 bytes in 10 cycles. A CRC result register contains the current CRC which is updated whenever a CRC instruction is executed.

- **Complex math**

  Complex math is used in many applications. The VCU A few of which are:
  
  - Fast fourier transform (FFT)
    
    The complex FFT is used in spread spectrum communications, as well in many signal processing algorithms.
  
  - Complex filters
    
    Complex filters improve data reliability, transmission distance, and power efficiency. The C28x+VCU can perform a complex I and Q multiple with coefficients (four multiplies) in a single cycle. In addition, the C28x+VCU can read/write the real and imaginary parts of 16-bit complex data to memory in a single cycle.

Table 3-2 shows a summary of the VCU operations enabled by the VCU:

<table>
<thead>
<tr>
<th>Complex Math Operation</th>
<th>VCU Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Or Subtract</td>
<td>1</td>
<td>32 +/- 32 = 32-bit (Useful for filters)</td>
</tr>
<tr>
<td>Add or Subtract</td>
<td>1</td>
<td>16 +/- 32 = 15-bit (Useful for FFT)</td>
</tr>
<tr>
<td>Multiply</td>
<td>2p</td>
<td>16 x 16 = 32-bit</td>
</tr>
<tr>
<td>Multiply &amp; Accumulate (MAC)</td>
<td>2p</td>
<td>32 + 32 = 32-bit, 16 x 16 = 32-bit</td>
</tr>
<tr>
<td>RPT MAC</td>
<td>2p+N</td>
<td>Repeat MAC. Single cycle after the first operation.</td>
</tr>
</tbody>
</table>

This C28x+VCU draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The C2000 features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and modified Harvard architecture (usable in Von Neumann mode). The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation. The modified Harvard architecture of the CPU enables instruction and data fetches to be performed in parallel. The CPU can read instructions and data while it writes data simultaneously to maintain the single-cycle instruction operation across the pipeline. The CPU does this over six separate address/data buses.

Throughout this document the following notations are used:
• C28x refers to the C28x fixed-point CPU.
• C28x plus Floating-Point and C28x+FPU both refer to the C28x CPU with enhancements to support IEEE single-precision floating-point operations.
• C28x plus VCU and C28x+VCU both refer to the C28x CPU with enhancements to support viterbi decode, complex math and CRC.
• Some devices have both the FPU and the VCU. These are referred to as C28x+FPU+VCU.

3.2 Components of the C28x plus VCU

The VCU extends the capabilities of the C28x CPU and C28x+FPU processors by adding additional instructions. No changes have been made to existing instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x are completely compatible with the C28x+VCU. All of the features of the C28x documented in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430) apply to the C28x+VCU. All features documented in the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRU02) apply to the C28x+FPU+VCU. Figure 3-1 shows the block diagram of the VCU.

The C28x+VCU contains the same features as the C28x fixed-point CPU:
• A central processing unit for generating data and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory.
• Emulation logic for monitoring and controlling various parts and functions of the device and for testing device operation. This logic is identical to that on the C28x fixed-point CPU.
• Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts. This logic is identical to the C28x fixed-point CPU.
• Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.

• Address register arithmetic unit (ARAU). The ARAU generates data memory addresses and increments or decrements pointers in parallel with ALU operations.

• Fixed-Point instructions are pipeline protected. This pipeline for fixed-point instructions is identical to that on the C28x fixed-point CPU. The CPU implements an 8-phase pipeline that prevents a write to and a read from the same location from occurring out of order.

• Barrel shifter. This shifter performs all left and right shifts of fixed-point data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.

• Fixed-Point Multiplier. The multiplier performs 32-bit × 32-bit 2s-complement multiplication with a 64-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.

The VCU adds the following features:

• Instructions to support Cyclic Redundancy Check (CRC) or a polynomial code checksum:
  – CRC8
  – CRC16
  – CRC32

• Clocked at the same rate as the main CPU (SYSCLKOUT).

• Instructions to support a software implementation of a Viterbi Decoder
  – Branch metrics calculations
  – Add-Compare Select or Viterbi Butterfly
  – Traceback

• Complex Math Arithmetic Unit
  – Add or Subtract
  – Multiply
  – Multiply and Accumulate (MAC)
  – Repeat MAC (RPT || MAC)

• Independent register space. These registers function as source and destination registers for VCU instructions.

• Some VCU instructions require pipeline alignment. This alignment is done through software to allow the user to improve performance by taking advantage of required delay slots. See Section 3.5 for more information.

Devices with the floating-point unit also include:

• Floating point unit (FPU). The 32-bit FPU performs IEEE single-precision floating-point operations.

• Dedicated floating-point registers.

### 3.3 Emulation Logic

The emulation logic is identical to that on the C28x fixed-point CPU. This logic includes the following features. For more details about these features, refer to the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430):

• Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline

• A counter for performance benchmarking.

• Multiple debug events. Any of the following debug events can cause a break in program execution:
  – A breakpoint initiated by the ESTOP0 or ESTOP1 instruction.
  – An access to a specified program-space or data-space location. When a debug event causes the C28x to enter the debug-halt state, the event is called a break event.

• Real-time mode of operation.
3.3.1 Memory Map

Like the C28x, the C28x+VCU uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x+VCU designs are uniformly mapped to both program and data space. For specific details about each of the map segments, see the data manual for a particular device device.

3.3.2 CPU Interrupt Vectors

The C28x+VCU interrupt vectors are identical to those on the C28x CPU. Sixty-four addresses in program space are set aside for a table of 32 CPU interrupt vectors. For more information about the CPU vectors, see TMS320C28x CPU and Instruction Set Reference Guide (literature number SPRU430). Typically the CPU interrupt vectors are only used during the boot up of the device by the boot ROM. Once an application has taken control it should initialize and enable the peripheral interrupt expansion block (PIE).

3.3.3 Memory Interface

The C28x+VCU memory interface is identical to that on the C28x. The C28x+VCU memory map is accessible outside the CPU by the memory interface, which connects the CPU logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed. The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the CPU supports special byte-access instructions that can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.

3.3.4 Address and Data Buses

Like the C28x, the memory interface has three address buses:

- **PAB**: Program address bus: The 22-bit PAB carries addresses for reads and writes from program space.
- **DRAB**: Data-read address bus: The 32-bit DRAB carries addresses for reads from data space.
- **DWAB**: Data-write address bus: The 32-bit DWAB carries addresses for writes to data space.

The memory interface also has three data buses:

- **PRDB**: Program-read data bus: The 32-bit PRDB carries instructions during reads from program space.
- **DRDB**: Data-read data bus: The 32-bit DRDB carries data during reads from data space.
- **DWDB**: Data-/Program-write data bus: The 32-bit DWDB carries data during writes to data space or program space.

A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time. This behavior is identical to the C28x CPU.

3.3.5 Alignment of 32-Bit Accesses to Even Addresses

The C28x+VPU expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU will begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the CPU.
You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space.
### 3.4 Register Set

Devices with the C28x+VCU include the standard C28x register set plus an additional set of VCU specific registers. The additional VCU registers are the following:

- Result registers: VR0, VR1... VR8
- Traceback registers: VT0, VT1
- Configuration and status register: VSTATUS
- CRC result register: VCRC
- Repeat block register: RB

Figure 3-2 shows the register sets for the 28x CPU, the FPU and the VCU. The following section discusses the VCU register set in detail.

#### Figure 3-2. C28x + FPU + VCU Registers

<table>
<thead>
<tr>
<th>Standard C28x Register Set</th>
<th>Additional 32-bit FPU Registers</th>
<th>Standard VCU Register Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC (32-bit)</td>
<td>R0H (32-bit)</td>
<td>VR0</td>
</tr>
<tr>
<td>P (32-bit)</td>
<td>R1H (32-bit)</td>
<td>VR1</td>
</tr>
<tr>
<td>XT (32-bit)</td>
<td>R2H (32-bit)</td>
<td>VR2</td>
</tr>
<tr>
<td>XAR0 (32-bit)</td>
<td>R3H (32-bit)</td>
<td>VR3</td>
</tr>
<tr>
<td>XAR1 (32-bit)</td>
<td>R4H (32-bit)</td>
<td>VR4</td>
</tr>
<tr>
<td>XAR2 (32-bit)</td>
<td>R5H (32-bit)</td>
<td>VR5</td>
</tr>
<tr>
<td>XAR3 (32-bit)</td>
<td>R6H (32-bit)</td>
<td>VR6</td>
</tr>
<tr>
<td>XAR4 (32-bit)</td>
<td>R7H (32-bit)</td>
<td>VR7</td>
</tr>
<tr>
<td>XAR5 (32-bit)</td>
<td>FPU Status Register (STF)</td>
<td>VR8</td>
</tr>
<tr>
<td>XAR6 (32-bit)</td>
<td>Repeat Block Register (RB)</td>
<td>VT0</td>
</tr>
<tr>
<td>XAR7 (32-bit)</td>
<td></td>
<td>VT1</td>
</tr>
<tr>
<td>PC (22-bit)</td>
<td></td>
<td>VSTATUS</td>
</tr>
<tr>
<td>RPC (22-bit)</td>
<td></td>
<td>VCRC</td>
</tr>
<tr>
<td>DP (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST0 (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST1 (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IER (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFR (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGIER (16-bit)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 3.4.1 VCU Register Set

The table below describes the VCU module register set. The last three columns indicate whether the particular module within the VCU can make use of the register.

FPU registers R0H - R7H and STF are shadowed for fast context save and restore.
### Table 3-3. VCU Register Set

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Size</th>
<th>Description</th>
<th>Viterbi</th>
<th>Complex Math</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>32-bits</td>
<td>General purpose register 0</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR1</td>
<td>32-bits</td>
<td>General purpose register 1</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bits</td>
<td>General purpose register 2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bits</td>
<td>General purpose register 3</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bits</td>
<td>General purpose register 4</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR5</td>
<td>32-bits</td>
<td>General purpose register 5</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR6</td>
<td>32-bits</td>
<td>General purpose register 6</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR7</td>
<td>32-bits</td>
<td>General purpose register 7</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR8</td>
<td>32-bits</td>
<td>General purpose register 8</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VT0</td>
<td>32-bits</td>
<td>32-bit transition bit register 0</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VT1</td>
<td>32-bits</td>
<td>32-bit transition bit register 1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VSTATUS</td>
<td>32-bits</td>
<td>VCU status and configuration register (1)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VCRC</td>
<td>32-bits</td>
<td>Cyclic redundancy check (CRC) result register</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(1) Debugger writes are not allowed to the VSTATUS register.

Table 3-4 lists the CPU registers available on devices with the C28x, the C28x+FPU, the C28x+VCU and the C28x+FPU+VCU.
### Table 3-4. 28x CPU Register Summary

<table>
<thead>
<tr>
<th>Register</th>
<th>C28x CPU</th>
<th>C28x+FPU</th>
<th>C28x+VCU</th>
<th>C28x+FPU+VCU</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Fixed-point accumulator</td>
</tr>
<tr>
<td>AH</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High half of ACC</td>
</tr>
<tr>
<td>AL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of ACC</td>
</tr>
<tr>
<td>XAR0 - XAR7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Auxiliary register 0 - 7</td>
</tr>
<tr>
<td>AR0 - AR7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of XAR0 - XAR7</td>
</tr>
<tr>
<td>DP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Data-page pointer</td>
</tr>
<tr>
<td>IFR</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Interrupt flag register</td>
</tr>
<tr>
<td>IER</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Interrupt enable register</td>
</tr>
<tr>
<td>DBGIER</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Debug interrupt enable register</td>
</tr>
<tr>
<td>P</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Fixed-point product register</td>
</tr>
<tr>
<td>PH</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High half of P</td>
</tr>
<tr>
<td>PL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of P</td>
</tr>
<tr>
<td>PC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Program counter</td>
</tr>
<tr>
<td>RPC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Return program counter</td>
</tr>
<tr>
<td>SP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>ST0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Status register 0</td>
</tr>
<tr>
<td>ST1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Status register 1</td>
</tr>
<tr>
<td>XT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Fixed-point multiplicand register</td>
</tr>
<tr>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High half of XT</td>
</tr>
<tr>
<td>TL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of XT</td>
</tr>
<tr>
<td>ROH - R7H</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Floating-point Unit result registers</td>
</tr>
<tr>
<td>STF</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Floating-point Unit status register</td>
</tr>
<tr>
<td>RB</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Repeat block register</td>
</tr>
<tr>
<td>VR0 - VR8</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>VCU general purpose registers</td>
</tr>
<tr>
<td>VT0, VT1</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>VCU transition bit register 0 and 1</td>
</tr>
<tr>
<td>VSTATUS</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>VCU status and configuration</td>
</tr>
<tr>
<td>VCRC</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>CRC result register</td>
</tr>
</tbody>
</table>

### 3.4.2 VCU Status Register (VSTATUS)

The VCU status register (VSTATUS) register is described in Figure 3-3. There is no single instruction to directly transfer the VSTATUS register to a C28x register. To transfer the contents:

1. Store VSTATUS into memory using `VMOV32 mem32, VSTATUS` instruction
2. Load the value from memory into a main C28x CPU register.

Configuration bits within the VSTATUS registers are set or cleared using VCU instructions.

#### Figure 3-3. VCU Status Register (VSTATUS)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>24</th>
<th>23</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
</tbody>
</table>

**Legend:** R/W = Read/Write; R = Read only; _n_ = value after reset
### Table 3-5. VCU Status (VSTATUS) Register Field Descriptions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 - 14</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>
| 13    | OVFI  | 0 | Overflow or Underflow Flag: Imaginary Part  
|       |       | 1 | Indicates an overflow or underflow has occurred during the computation of the imaginary part of operations shown in Table 3-6. This bit will be set regardless of the value of the VSTATUS[SAT] bit. OVFI bit will remain set until it is cleared by executing the VCLRQVFI instruction. |
| 12    | OVFR  | 0 | Overflow or Underflow Flag: Real Part  
|       |       | 1 | Indicates overflow or underflow has occurred during a real number calculation for operations shown in Table 3-6. This bit will be set regardless of the value of the VSTATUS[SAT] bit. This bit will remain set until it is cleared by executing the VCLRQVFR instruction. |
| 11    | RND   | 0 | Rounding  
|       |       | 1 | When a right-shift operation is performed the lower bits of the value will be lost. The RND bit determines if the shifted value is rounded or if the shifted-out bits are simply truncated. This is described in operations which use right-shift and rounding are shown in Table 3-6. The RND bit is set by the VRNDON instruction and cleared by the VRNDOFF instruction.  
|       |       | 1 | Rounding is performed. Refer to the instruction descriptions for information on how the operation is affected by the RND bit. |
| 10    | SAT   | 0 | Saturation  
|       |       | 1 | This bit determines whether saturation will be performed for operations shown in Table 3-6. The SAT bit is set by the VSATON instruction and is cleared by the VSATOFF instruction.  
|       |       | 1 | Saturation is performed. |
| 9-5   | SHIFTL| 0 | Left Shift  
|       |       | 0x01 - 0x1F | Operations which use left-shift are shown in Table 3-6. The shift SHIFTL field can be set or cleared by the VSETSHL instruction.  
|       |       | 1 | No left shift.  
|       |       | 0x01 - 0x1F | Refer to the instruction description for information on how the operation is affected by the shift value. During the left-shift, the lower bits are filled with 0's. |
| 4-0   | SHIFTR| 0 | Right Shift  
|       |       | 0x01 - 0x1F | Operations which use right-shift and rounding are shown in Table 3-6. The shift SHIFTR field can be set or cleared by the VSETSHR instruction.  
|       |       | 1 | No right shift.  
|       |       | 0x01 - 0x1F | Refer to the instruction descriptions for information on how the operation is affected by the shift value. During the right-shift, the lower bits are lost, and the shifted value is sign extended. If rounding is enabled (VSTATUS[RND] == 1), then the value will be rounded instead of truncated. |

Table 3-6 shows a summary of the operations that are affected by or modify bits in the VSTATUS register.

### Table 3-6. Operation Interaction with VSTATUS Bits

<table>
<thead>
<tr>
<th>Operation (1)</th>
<th>Description</th>
<th>OVFI</th>
<th>OVFR</th>
<th>RND</th>
<th>SAT</th>
<th>SHIFTL</th>
<th>SHIFTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITDLADDSUB</td>
<td>Viterbi Add and Subtract Low</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VITDHDADDSUB</td>
<td>Viterbi Add and Subtract High</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VITDLSUBADD</td>
<td>Viterbi Subtract and Add Low</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VITDHSUBADD</td>
<td>Viterbi Subtract and Add High</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VITBM2</td>
<td>Viterbi Branch Metric CR 1/2</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VITBM3</td>
<td>Viterbi Branch Metric CR 1/3</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCADD</td>
<td>Complex 32 + 32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>VCDADD16</td>
<td>Complex 16 + 32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

(1) Some parallel instructions also include these operations. In this case, the operation will also modify, or be affected by, VSTATUS bits as when used as part of a parallel instruction.
Table 3-6. Operation Interaction with VSTATUS Bits (continued)

<table>
<thead>
<tr>
<th>Operation (1)</th>
<th>Description</th>
<th>OVFI</th>
<th>OVFR</th>
<th>RND</th>
<th>SAT</th>
<th>SHIFT L</th>
<th>SHIFT R</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCDSUB16</td>
<td>Complex 16 - 32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>VCMAC</td>
<td>Complex 32 + 32 = 32, 16 x 16 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>VCMPY</td>
<td>Complex 16 x 16 = 32</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCSUB</td>
<td>Complex 32 -32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>
3.4.3 Repeat Block Register (RB)

The repeat block instruction (RPTB) applies to devices with the C28x+FPU and the C28x+VCU. This instruction allows you to repeat a block of code as shown in Example 3-1.

Example 3-1. The Repeat Block (RPTB) Instruction uses the RB Register

```
; find the largest element and put its address in XAR6
; This example makes use of floating-point (C28x + FPU) instructions
;
MOV32 R0H, *XAR0++ ; Aligns the next instruction to an even address
NOP ; Makes RPTB odd aligned - required for a block size of 8
RPTB VECTOR_MAX_END, AR7 ; RA is set to 1
MOVL ACC,XAR0
MOV32 R1H,*XAR0++ ; RSIZE reflects the size of the RPTB block
MAXF32 R0H,R1H ; in this case the block size is 8
MOVST0 NF,ZF
MOVL XAR6,ACC,LT
VECTOR_MAX_END: ; RE indicates the end address. RA is cleared
```

The C28x FPU or VCU automatically populates the RB register based on the execution of a RPTB instruction. This register is not normally read by the application and does not accept debugger writes.

Figure 3-4. Repeat Block Register (RB)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31    | RAS   | 0     | Repeat Block Active Shadow Bit
|       |       | 1     | A repeat block was active when the interrupt was taken. |
| 30    | RA    | 0     | Repeat Block Active Bit
|       |       | 1     | A repeat block was active when the interrupt was taken. |
| 29-23 | RSIZE | 0-7   | Repeat Block Size
|       |       | 8/9-0x7F | A RPTB block that starts at an even address must include at least 9 16-bit words and a block that starts at an odd address must include at least 8 16-bit words. The maximum block size is 127 16-bit words. The codegen assembler will check for proper block size and alignment.
|       |       | 8/9-0x7F | This 7-bit value specifies the number of 16-bit words within the repeat block. This field is initialized when the RPTB instruction is executed. The value is calculated by the assembler and inserted into the RPTB instruction's RSIZE opcode field. |

Table 3-7. Repeat Block (RB) Register Field Descriptions
Table 3-7. Repeat Block (RB) Register Field Descriptions (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 22-16 | RE    |       | Repeat Block End Address  
|       |       |       | This 7-bit value specifies the end address location of the repeat block. The RE value is calculated by hardware based on the RSIZE field and the PC value when the RPTB instruction is executed.  
|       |       |       | RE = lower 7 bits of (PC + 1 + RSIZE) |
| 15-0  | RC    | 0     | Repeat Count  
|       |       | 1-0xFFFF | The block will not be repeated; it will be executed only once. In this case the repeat active, RA, bit will not be set.  
|       |       |       | This 16-bit value determines how many times the block will repeat. The counter is initialized when the RPTB instruction is executed and is decremented when the PC reaches the end of the block. When the counter reaches zero, the repeat active bit is cleared and the block will be executed one more time. Therefore the total number of times the block is executed is RC+1. |
3.5 Pipeline

This section describes the VCU pipeline stages and presents cases where pipeline alignment must be considered.

3.5.1 Pipeline Overview

The C28x VCU pipeline is identical to the C28x pipeline for all standard C28x instructions. In the decode2 stage (D2), it is determined if an instruction is a C28x instruction, a FPU instruction, or a VCU instruction. The pipeline flow is shown in Figure 3-5.

Notice that stalls due to normal C28x pipeline stalls (D2) and memory waitstates (R2 and W) will also stall any C28x VCU instruction. Most C28x VCU instructions are single cycle and will complete in the VCU E1 or W stage which aligns to the C28x pipeline. Some instructions will take an additional execute cycle (E2). For these instructions you must wait a cycle for the result from the instruction to be available. The rest of this section will describe when delay cycles are required. Keep in mind that the assembly tools for the C28x+VCU will issue an error if a delay slot has not been handled correctly.

![Figure 3-5. C28x + FCU + VCU Pipeline](image)

3.5.2 General Guidelines for Floating-Point Pipeline Alignment

The majority of the VCU instructions do not require any special pipeline considerations. This section lists the few operations that do require special consideration.

While the C28x+VCU assembler will issue errors for pipeline conflicts, you may still find it useful to understand when software delays are required. This section describes three guidelines you can follow when writing C28x+VCU assembly code.

VCU instructions that require delay slots have a ‘p’ after their cycle count. For example ‘2p’ stands for 2 pipelined cycles. This means that an instruction can be started every cycle, but the result of the instruction will only be valid one instruction later.

There are three general guidelines to determine if an instruction needs a delay slot:

1. Branch metric calculation for a code rate of 1/3 takes 2p cycles.
2. Complex multiply and MAC take 2p cycles.
3. Everything else does not require a delay slot.
An example of the complex multiply instruction is shown in Example 3-2. VCMPY is a 2p instruction and therefore requires one delay slot. The destination registers for the operation, VR2 and VR3, will be updated one cycle after the instruction for a total of 2 cycles. Therefore, a NOP or instruction that does not use VR2 or VR3 must follow this instruction.

Any memory stall or pipeline stall will also stall the VCU. This keeps the VCU aligned with the C28x pipeline and there is no need to change the code based on the waitstates of a memory block.

**Example 3-2. 2p Instruction Pipeline Alignment**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPY VR3, VR2, VR1, VR0</td>
<td>2 pipeline cycles (2p)</td>
</tr>
<tr>
<td>NOP</td>
<td>1 cycle delay or non-conflicting instruction</td>
</tr>
<tr>
<td>NOP</td>
<td>Any instruction</td>
</tr>
</tbody>
</table>

3.5.3 **Parallel Instructions**

Parallel instructions are single opcodes that perform two operations in parallel. The guidelines provided in Section 3.5.2 apply to parallel instructions as well. In this case the cycle count will be given for both operations. For example, a branch metric calculation for code rate of 1/3 with a parallel load takes 2p/1 cycles. This means the branch metric portion of the operation takes 2 pipelined cycles while the move portion of the operation is single cycle. NOPs or other non conflicting instructions must be inserted to align the branch metric calculation portion of the operation as shown in Example 3-4.

**Example 3-3. Branch Metric CR 1/2 Calculation with Parallel Load**

```plaintext
; VITBM2 || VMOV32 instruction: branch metrics calculation with parallel load
; VBITM2 is a 1 cycle operation  (code rate = 1/2)
; VMOV32 is a 1 cycle operation
; VITBM2 VR0 ; Load VR0 with the 2 branch metrics
|| VMOV32 VR2, @Val ; VR2 gets the contents of Val
; <-- VMOV32 completes here (VR2 is valid)
; <-- VITBM2 completes here (VR0 is valid)
<instruction 2> ; Any instruction, can use VR2 and/or VR0
```

**Example 3-4. Branch Metric CR 1/3 Calculation with Parallel Load**

```plaintext
; VITBM3 || VMOV32 instruction: branch metrics calculation with parallel load
; VBITM3 is a 2p cycle operation  (code rate = 1/3)
; VMOV32 is a 1 cycle operation
; VITBM3 VR0, VR1, VR2 ; Load VR0 and VR1 with the 4 branch metrics
|| VMOV32 VR2, @Val ; VR2 gets the contents of Val
; <-- VMOV32 completes here (VR2 is valid)
<instruction 2> ; Must not use VR0 or VR1. Can use VR2.
; <-- VITBM3 completes here (VR0, VR1 are valid)
<instruction 3> ; Any instruction, can use VR2 and/or VR0
```

3.5.4 **Invalid Delay Instructions**

All VCU, FPU and fixed-point instructions can be used in VCU instruction delay slots as long as source and destination register conflicts are avoided. The C28x+VCU assembler will issue an error anytime you use an conflicting instruction within a delay slot. The following guidelines can be used to avoid these conflicts.
NOTE: Destination register conflicts in delay slots:

Any operation used for pipeline alignment delay must not use the same destination register as the instruction requiring the delay. See Example 3-5.

In Example 3-5 the VCMPY instruction uses VR2 and VR3 as its destination registers. The next instruction should not use VR2 or VR3 as a destination. Since the VMOV32 instruction uses the VR3 register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than VR2 for the VMOV32 instruction as shown in Example 3-6.
Example 3-5. Destination Register Conflict

```
; Invalid delay instruction.
; Both instructions use the same destination register (VR3)
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VMOV32 VR3, mem32 ; Invalid delay instruction
; <-- VCMPY completes, VR3, VR2 are valid
```

Example 3-6. Destination Register Conflict Resolved

```
; Valid delay instruction
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VMOV32 VR7, mem32 ; Valid delay instruction
```

NOTE: Instructions in delay slots cannot use the instruction's destination register as a source register.

Any operation used for pipeline alignment delay must not use the destination register of the instruction requiring the delay as a source register as shown in Example 3-7. For parallel instructions, the current value of a register can be used in the parallel operation before it is overwritten as shown in Example 3-9.

In Example 3-7 the VCMPY instruction again uses VR3 and VR2 as its destination registers. The next instruction should not use VR3 or VR2 as its source since the VCMPY will take an additional cycle to complete. Since the VCADD instruction uses the VR2 as a source register a pipeline conflict will be issued by the assembler. The use of VR3 will also cause a pipeline conflict. This conflict can be resolved by using a register other than VR2 or VR3 or by inserting a non-conflicting instruction between the VCMPY and VCADD instructions. Since the VNEG does not use VR2 or VR3 this instruction can be moved before the VCADD as shown in Example 3-8.

Example 3-7. Destination/Source Register Conflict

```
; Invalid delay instruction.
; VCADD should not use VR2 or VR3 as a source operand
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VCADD VR5, VR4, VR3, VR2 ; Invalid delay instruction
VNEG VR0 ; <- VCMPY completes, VR3, VR2 valid
```

Example 3-8. Destination/Source Register Conflict Resolved

```
; Valid delay instruction.
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VNEG VR0 ; Non conflicting instruction or NOP
VCADD VR5, VR4, VR3, VR2 ; <- VCMPY completes, VR3, VR2 valid
```

It should be noted that a source register for the 2nd operation within a parallel instruction can be the same as the destination register of the first operation. This is because the two operations are started at the same time. The 2nd operation is not in the delay slot of the first operation. Consider Example 3-9 where the VCMPY uses VR3 and VR2 as its destination registers. The VMOV32 is the 2nd operation in the instruction and can freely use VR3 or VR2 as a source register. In the example, the contents of VR3 before the multiply will be used by MOV32.
Example 3-9. Parallel Instruction Destination/Source Exception

; Valid parallel operation.
; VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
|| VMOV32 mem32, VR3 ; <-- Uses VR3 before the VCMPY update
|| NOP ; <-- Delay for VCMPY
; <-- VR2, VR3 updated

Likewise, the source register for the 2nd operation within a parallel instruction can be the same as one of the source registers of the first operation. The VCMPY operation in Example 3-10 uses the VR0 register as one of its sources. This register is also updated by the VMOV32 instruction. The multiplication operation will use the value in VR0 before the VMOV32 updates it.

Example 3-10. Parallel Instruction Destination/Source Exception

; Valid parallel operation.
VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
|| VMOV32 VR0, mem32 ; <-- Uses VR3 before the VCMPY update
|| NOP ; <-- Delay for VCMPY
; <-- VR2, VR3 updated

NOTE: Operations within parallel instructions cannot use the same destination register.
When two parallel operations have the same destination register, the result is invalid.
For example, see Example 3-11.

If both operations within a parallel instruction try to update the same destination register as shown in Example 3-11 the assembler will issue an error.

Example 3-11. Invalid Destination Within a Parallel Instruction

; Invalid parallel instruction. Both operations use VR3 as a destination register
; VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
|| VMOV32 VR3, mem32 ; <-- Invalid
3.6 Instruction Set

This section describes the assembly language instructions of the VCU. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are independent from C28x and C28x+FPU instruction sets.

3.6.1 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Operands
- Opcode
- Description
- Exceptions
- Pipeline
- Examples
- See also

The example INSTRUCTION is shown to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information. VCU instructions follow the same format as the C28x; the source operand(s) are always on the right and the destination operand(s) are on the left.

The explanations for the syntax of the operands used in the instruction descriptions for the C28x VCU are given in Table 3-8.

Table 3-8. Operand Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>16-bit immediate (hex or float) value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FHiHex</td>
<td>16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32Fhex</td>
<td>32-bit immediate value that represents an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32F</td>
<td>Immediate float value represented in floating-point representation</td>
</tr>
<tr>
<td>#0.0</td>
<td>Immediate zero</td>
</tr>
<tr>
<td>#5-bit</td>
<td>5-bit immediate unsigned value</td>
</tr>
<tr>
<td>addr</td>
<td>Opcode field indicating the addressing mode</td>
</tr>
<tr>
<td>Im(X), Im(Y)</td>
<td>Imaginary part of the input X or input Y</td>
</tr>
<tr>
<td>Im(Z)</td>
<td>Imaginary part of the output Z</td>
</tr>
<tr>
<td>Re(X), Re(Y)</td>
<td>Real part of the input X or input Y</td>
</tr>
<tr>
<td>Re(Z)</td>
<td>Real part of the output Z</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
<tr>
<td>VRa</td>
<td>VR0 - VR8 registers. Some instructions exclude VR8. Refer to the instruction description for details.</td>
</tr>
<tr>
<td>VR0H, VR1H...VR7H</td>
<td>VR0 - VR7 registers, high half.</td>
</tr>
<tr>
<td>VR0L, VR1L...VR7L</td>
<td>VR0 - VR7 registers, low half.</td>
</tr>
<tr>
<td>VT0, VT1</td>
<td>Transition bit register VT0 or VT1.</td>
</tr>
</tbody>
</table>

Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).
### Table 3-9. INSTRUCTION dest, source1, source2 Short Description

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dest1</td>
<td>Description for the 1st operand for the instruction</td>
</tr>
<tr>
<td>source1</td>
<td>Description for the 2nd operand for the instruction</td>
</tr>
<tr>
<td>source2</td>
<td>Description for the 3rd operand for the instruction</td>
</tr>
<tr>
<td>Opcode</td>
<td>This section shows the opcode for the instruction</td>
</tr>
<tr>
<td>Description</td>
<td>Detailed description of the instruction execution is described. Any constraints on</td>
</tr>
<tr>
<td></td>
<td>the operands imposed by the processor or the assembler are discussed.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Any constraints on the operands or use of the instruction imposed by the processor</td>
</tr>
<tr>
<td></td>
<td>are discussed.</td>
</tr>
<tr>
<td>Pipeline</td>
<td>This section describes the instruction in terms of pipeline cycles as described in</td>
</tr>
<tr>
<td></td>
<td>Section 3.5</td>
</tr>
<tr>
<td>Example</td>
<td>Examples of instruction execution. If applicable, register and memory values are</td>
</tr>
<tr>
<td></td>
<td>given before and after instruction execution. Some examples are code fragments while</td>
</tr>
<tr>
<td></td>
<td>other examples are full tasks that assume the VCU is correctly configured and the</td>
</tr>
<tr>
<td></td>
<td>main CPU has passed it data.</td>
</tr>
<tr>
<td>Operands</td>
<td>Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).</td>
</tr>
</tbody>
</table>
### 3.6.2 General Instructions

The instructions are listed alphabetically, preceded by a summary.

#### Table 3-10. General Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP RB — Pop the RB Register from the Stack</td>
<td>359</td>
</tr>
<tr>
<td>PUSH RB — Push the RB Register onto the Stack</td>
<td>361</td>
</tr>
<tr>
<td>RPTB label, loc16 — Repeat A Block of Code</td>
<td>363</td>
</tr>
<tr>
<td>RPTB label, #RC — Repeat a Block of Code</td>
<td>365</td>
</tr>
<tr>
<td>VCLEAR VRa — Clear General Purpose Register</td>
<td>367</td>
</tr>
<tr>
<td>VCLEARALL — Clear All General Purpose and Transition Bit Registers</td>
<td>368</td>
</tr>
<tr>
<td>VCLROVFI — Clear Imaginary Overflow Flag</td>
<td>369</td>
</tr>
<tr>
<td>VCLROVFRI — Clear Real Overflow Flag</td>
<td>370</td>
</tr>
<tr>
<td>VMOV16 mem16, VRaL — Store General Purpose Register, Low Half</td>
<td>371</td>
</tr>
<tr>
<td>VMOV16 VRaL, mem16 — Load General Purpose Register, Low Half</td>
<td>372</td>
</tr>
<tr>
<td>VMOV32 mem32, VRa — Store General Purpose Register</td>
<td>373</td>
</tr>
<tr>
<td>VMOV32 mem32, VSTATUS — Store VCU Status Register</td>
<td>374</td>
</tr>
<tr>
<td>VMOV32 VTa, mem32 — Load 32-bit Transition Bit Register</td>
<td>375</td>
</tr>
<tr>
<td>VMOV32 VRa, mem32 — Load 32-bit General Purpose Register</td>
<td>376</td>
</tr>
<tr>
<td>VMOV32 VSTATUS, mem32 — Load VCU Status Register</td>
<td>377</td>
</tr>
<tr>
<td>VMOV32 VTa, mem32 — Load 32-bit Transition Bit Register</td>
<td>378</td>
</tr>
<tr>
<td>VMOVD32 VRa, mem32 — Load Register with Data Move</td>
<td>379</td>
</tr>
<tr>
<td>VMOVIX VRa, #16I — Load Upper Half of a General Purpose Register with 16-bit Immediate</td>
<td>380</td>
</tr>
<tr>
<td>VMOVXI VRa, #16I — Load General Purpose Register with Immediate</td>
<td>381</td>
</tr>
<tr>
<td>VRNDOFF — Disable Rounding</td>
<td>383</td>
</tr>
<tr>
<td>VRNDON — Enable Rounding</td>
<td>384</td>
</tr>
<tr>
<td>VSATOFF — Disable Saturation</td>
<td>385</td>
</tr>
<tr>
<td>VSATON — Enable Saturation</td>
<td>386</td>
</tr>
<tr>
<td>VSETSHL #5-bit — Initialize the Left Shift Value</td>
<td>387</td>
</tr>
<tr>
<td>VSETSHR #5-bit — Initialize the Left Shift Value</td>
<td>388</td>
</tr>
</tbody>
</table>
**POP RB**  
*Pop the RB Register from the Stack*

**Operands**

| RB | repeat block register |

**Opcode**

`LSW: 1111 1111 1111 0001`

**Description**

Restore the RB register from stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```plaintext
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt:       ; RAS = RA, RA = 0
...               ...
  PUSH RB         ; Save RB register only if a RPTB block is used in the ISR
  ...            ...
  RPTB _BlockEnd, AL ; Execute the block AL+1 times
  ...            ...
  ...            ...
  _BlockEnd       ; End of block to be repeated
  ...            ...
  POP RB          ; Restore RB register ...
  IRET           ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must store before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```plaintext
; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt:     ; RAS = RA, RA = 0
...              ...
  PUSH RB        ; Always save RB register
  ...          ...
  CLRC INTM      ; Enable interrupts only after saving RB
  ...          ...
  ...          ...
  ; ISR may or may not include a RPTB block
  ...          ...
  SETC INTM      ; Disable interrupts before restoring RB
  ...          ...
  POP RB         ; Always restore RB register
  ...          ...
  IRET          ; RA = RAS, RAS = 0
```

**See also**

PUSH RB
RPTB label, loc16
RPTB label, #RC
PUSH RB

Push the RB Register onto the Stack

Operands

RB

repeat block register

Opcode

LSW: 1111 1111 1111 0000

Description

Save the RB register on the stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```assembly
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
; Interrupt:   ; RAS = RA, RA = 0
;            ...
;            PUSH RB ; Save RB register only if a RPTB block is used in the ISR
;            ...
;            ...
;            RPTB _BlockEnd, AL ; Execute the block AL+1 times
;            ...
;            ...
;            _BlockEnd ; End of block to be repeated
;            ...
;            ...
;            POP RB ; Restore RB register ...
;            IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```assembly
; Repeat Block within a Low-Priority Interrupt (Interruptible)
; Interrupt:                         ; RAS = RA, RA = 0
;                                     ...
;                                     PUSH RB ; Always save RB register
;                                     ...
;                                     CLRC INTM ; Enable interrupts only after saving RB
;                                     ...
;                                     ...
;                                     ...
; ISR may or may not include a RPTB block
;                                     ...
;                                     ...
;                                     SETC INTM ; Disable interrupts before restoring RB
;                                     ...
;                                     POP RB ; Always restore RB register
;                                     ...
;                                     IRET ; RA = RAS, RAS = 0
```

See also

POP RB
PUSH RB — Push the RB Register onto the Stack

RPTB label, loc16
RPTB label, #RC
RPTB label, loc16

Repeat A Block of Code

Operands

| label | This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block. |
| loc16 | 16-bit location for the repeat count value. |

Opcode

- LSW: 1011 0101 0bbb bbbb
- MSW: 0000 0000 loc16

Description

Initialize repeat block loop, repeat count from [loc16]

Restrictions

- The maximum block size is ≤127 16-bit words.
- An even aligned block must be ≥ 9 16-bit words.
- An odd aligned block must be ≥ 8 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This instruction takes four cycles on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

Example

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

; Repeat Block of 8 Words (Interruptible)
;
; Note: This example makes use of floating-point (C28x+FPU) instructions
;
; find the largest element and put its address in XAR6
.align 2
NOP
RPTB _VECTOR_MAX_END, AR7
; Execute the block AR7+1 times
MOVL ACC,XAR0 MOV32 R1H,*XAR0++ ; min size = 8, 9 words
MAXF32 R0H,R1H ; max size = 127 words
MOVST0 NF,ZF
MOVL XAR6,ACC,LT
VECTOR_MAX_END: ; label indicates the end
; RA is cleared

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the
interrupt. If the interrupt service routine does not include a RPTB block, then you do not
have to save the RB register.

; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Save RB register only if a RPTB block is used in the ISR
...
...
RPTB _BlockEnd, AL ; Execute the block AL+1 times
...
...
...
_BLOCKEND ; End of block to be repeated
...
...
POP RB ; Restore RB register ...
IRET ; RA = RAS, RAS = 0

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The
RB register must always be saved and restored in a low-priority interrupt. The RB
register must stored before interrupts are enabled. Likewise before restoring the RB
register interrupts must first be disabled.

; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Always save RB register
...
CLRC INTM ; Enable interrupts only after saving RB
...
...
...
; ISR may or may not include a RPTB block
...
...
SETC INTM ; Disable interrupts before restoring RB
...
POP RB ; Always restore RB register
...
IRET ; RA = RAS, RAS = 0

See also
POP RB
PUSH RB
RPTB label, #RC
RPTB label, #RC  

Repeat a Block of Code

Operands

| label | This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block. |
| #RC | 16-bit immediate value for the repeat count. |

Opcode

LSW: 1011 0101 1bbb bbbb
MSW: cccc cccc cccc cccc

Description

Repeat a block of code. The repeat count is specified as a immediate value.

Restrictions

- The maximum block size is $\leq 127$ 16-bit words.
- An even aligned block must be $\geq 9$ 16-bit words.
- An odd aligned block must be $\geq 8$ 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This instruction takes one cycle on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

Example

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

```
; Repeat Block of 8 Words (Interruptible)
;
; Note: This example makes use of floating-point (C28x+FPU) instructions
;
; find the largest element and put its address in XAR6
;
; .align 2
; NOP
; RPTB _VECTOR_MAX_END, AR7
; Execute the block AR7+1 times
    MOVL ACC,XAR0 MOV32 R1H,*XAR0++ ; min size = 8, 9 words
    MAXF32 R0H,R1H
    MOVST0 NF,ZF
    MOVL XAR6,ACC,LT
_Vector_MAX_END: ; label indicates the end
    ; RA is cleared
```

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the
interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Save RB register only if a RPTB block is used in the ISR...
...
RPTB #_BlockEnd, #5 ; Execute the block AL+1 times...
...
...

_BlockEnd ; End of block to be repeated...
...

POP RB ; Restore RB register...
IRET ; RA = RAS, RAS = 0

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Always save RB register...
...
CLRC INTM ; Enable interrupts only after saving RB...
...
...
}

; ISR may or may not include a RPTB block...
...

SETC INTM ; Disable interrupts before restoring RB...

POP RB ; Always restore RB register...
IRET ; RA = RAS, RAS = 0

See also
POP RB
PUSH RB
RPTB label, loc16
VCLEAR VRa — Clear General Purpose Register

**Operands**

| VRa | General purpose register: VR0, VR1... VR8 |

**Opcode**

| LSW: 1110 0110 1111 1000 |
| MSW: 0000 0000 0000 aaaa |

**Description**

Clear the specified general purpose register.

VRa = 0x00000000;

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
; Code fragment from a viterbi traceback
; For the first iteration the previous state metric must be
; initialized to zero (VR0).

; VCLEAR VR0 ; Clear the VR0 register
; MOV XAR5,*+XAR4[0] ; Point XAR5 to an array

; For first stage

; VMOV32 VT0, *--XAR3
; VMOV32 VT1, *--XAR3
; VTRACE *XAR5++,VR0,VT0,VT1 ; Uses VR0 (which is zero)

; etc...
```

**See also**

VCLEARALL
VTCLEAR
VCLEARALL — Clear All General Purpose and Transition Bit Registers

Operands

none

Opcode

LSW: 1110 0110 1111 1001
MSW: 0000 0000 0000 0000

Description

Clear all of the general purpose registers (VR0, VR1... VR8) and the transition bit registers (VT0 and VT1).

VR0 = 0x00000000;
VR1 = 0x00000000;
VR2 = 0x00000000;
VR3 = 0x00000000;
VR4 = 0x00000000;
VR5 = 0x00000000;
VR6 = 0x00000000;
VR7 = 0x00000000;
VR8 = 0x00000000;
VT0 = 0x00000000;
VT1 = 0x00000000;

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

; Context save all VCU VRa and VTa registers
;     VMOV32  *SP++, VR0
     VMOV32  *SP++, VR1
     VMOV32  *SP++, VR2
     VMOV32  *SP++, VR3
     VMOV32  *SP++, VR4
     VMOV32  *SP++, VR5
     VMOV32  *SP++, VR6
     VMOV32  *SP++, VR7
     VMOV32  *SP++, VR8
     VMOV32  *SP++, VT0
     VMOV32  *SP++, VT1
;
; Clear VR0 - VR8, VT0 and VT1
;     VCLEARALL
;     ; etc...

See also

VCLEAR VRa
VTCLEAR
<table>
<thead>
<tr>
<th>VCLROVFI</th>
<th><strong>Clear Imaginary Overflow Flag</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operands</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0000 1011</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Clear the imaginary overflow flag in the VSTATUS register. To clear the real flag, use the VCLROVFR instruction. The imaginary flag bit can be set by instructions shown in Table 3-6. Refer to individual instruction descriptions for details.</td>
</tr>
<tr>
<td><strong>VSTATUS[OVFI] = 0;</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>This instruction clears the OVFI flag.</td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>This is a single-cycle instruction.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
</tbody>
</table>
| **See also** | VCLROVFR  
VRNDON  
VSATOFF  
VSATON |
**VCLROVFR — Clear Real Overflow Flag**

**Operands**

none

**Opcode**

LSW: 1110 0101 0000 1010

**Description**

Clear the real overflow flag in the VSTATUS register. To clear the imaginary flag, use the VCLROVFI instruction. The imaginary flag bit can be set by instructions shown in Table 3-6. Refer to individual instruction descriptions for details.

VSTATUS[OVFR] = 0;

**Flags**

This instruction clears the OVFR flag.

**Pipeline**

This is a single-cycle instruction.

**See also**

VCLROVFI
VRNDON
VSATFOFF
VSATON
VMOV16 mem16, VRaL  

**Store General Purpose Register, Low Half**

<table>
<thead>
<tr>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
</tr>
<tr>
<td>VRaL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW: 1110 0010 0001 1000</td>
</tr>
<tr>
<td>MSW: 0000 aaaa mem16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store the low 16-bits of the specified general purpose register into the 16-bit memory location.</td>
</tr>
<tr>
<td>([\text{mem16}] = \text{VRa}[15:0];)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>This instruction does not modify any flags in the VSTATUS register.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a single-cycle instruction.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV16 VRaL, mem16</td>
</tr>
</tbody>
</table>
VMOV16 VRaL, mem16  —  Load General Purpose Register, Low Half

Operands

<table>
<thead>
<tr>
<th>VRaL</th>
<th>Low word of a general purpose register: VR0L, VR1L,...VR8L</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>Pointer to a 16-bit memory location. This will be the source for the VMOV16.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1100 1001
MSW: 0000 aaaa mem16

Description

Load the lower 16 bits of the specified general purpose register with the contents of memory pointed to by mem16.

VRa[15:0] = [mem16];

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

; Loop will run 106 times for 212 inputs to decoder
; Code fragment from viterbi decoder
; _LOOP:
;
; Calculate the branch metrics for code rate = 1/3
; Load VR0L, VR1L and VR2L with inputs
to the decoder from the array pointed to by XAR5
;
; VMOV16 VR0L, *XAR5++
VMOV16 VR1L, *XAR5++
VMOV16 VR2L, *XAR5++
;
; VR0L = BM0
; VR0H = BM1
; VR1L = BM2
; VR1H = BM3
; VR2L = pt_old[0]
; VR2H = pt_old[1]
;
VITBM3 VR0, VR1, VR2
VMOV32 VR2, *XAR1++
; etc...

See also

VMOV16 mem16, VRaL
VMOV32 mem32, VRa  

**Store General Purpose Register**

**Operands**

| mem32 | Pointer to a 32-bit memory location. This will be the destination of the VMOV32. |
| VRa   | General purpose register VR0, VR1... VR8 |

**Opcode**

<table>
<thead>
<tr>
<th>LSW: 1110 0010 0000 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 aaaa mem32</td>
</tr>
</tbody>
</table>

**Description**

Store the 32-bit contents of the specified general purpose register into the memory location pointed to by mem32.

\[\text{mem32} = \text{VRa};\]

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**See also**

- VMOV32 mem32, VSTATUS
- VMOV32 mem32, VTa
- VMOV32 VRa, mem32
- VMOV32 VTa, mem32
### VMOV32 mem32, VSTATUS — Store VCU Status Register

The **VMOV32 mem32, VSTATUS** instruction is used to store the VCU status register into a memory location specified by `mem32`.

#### Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the destination of the VMOV32.</td>
</tr>
<tr>
<td>VSTATUS</td>
<td>VCU status register.</td>
</tr>
</tbody>
</table>

#### Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0010 0000 1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 0000 mem32</td>
</tr>
</tbody>
</table>

#### Description

Store the VSTATUS register into the memory location pointed to by `mem32`:

\[
\text{[mem32]} = \text{VSTATUS} ;
\]

#### Flags

This instruction does not modify any flags in the VSTATUS register.

#### Pipeline

This is a single-cycle instruction.

#### Example

Use the following instructions to implement the VMOV32 mem32, VSTATUS instruction:

- `VMOV32 mem32, VRa`
- `VMOV32 mem32, VTa`
- `VMOV32 VRa, mem32`
- `VMOV32 VSTATUS, mem32`
- `VMOV32 VTa, mem32`
VMOV32 mem32, VTa  

**Store Transition Bit Register**

**Operands**

| mem32 | pointer to a 32-bit memory location. This will be the destination of the VMOV32. |
| VTa | Transition bits register VT0 or VT1 |

**Opcode**

LSW: 1110 0010 0000 0101  
MSW: 0000 00tt mem32

**Description**

Store the 32-bits of the specified transition bits register into the memory location pointed to by mem32.  

\[ \text{[mem32]} = \text{VTa}; \]

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

See also

VMOV32 mem32, VRa  
VMOV32 mem32, VSTATUS  
VMOV32 VRa, mem32  
VMOV32 VSTATUS, mem32  
VMOV32 VTa, mem32
**VMOV32 VRa, mem32 — Load 32-bit General Purpose Register**

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRa</td>
<td>General purpose register VR0, VR1,...,VR8</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW: 1110 0011 1111 0000</td>
<td>Load the specified general purpose register with the 32-bit value in memory pointed to by mem32. VRa = [mem32];</td>
</tr>
<tr>
<td>MSW: 0000 aaaa mem32</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This instruction does not modify any flags in the VSTATUS register.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a single-cycle instruction.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>See also</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV32 mem32, VRa</td>
<td></td>
</tr>
<tr>
<td>VMOV32 mem32, VSTATUS</td>
<td></td>
</tr>
<tr>
<td>VMOV32 mem32, VTa</td>
<td></td>
</tr>
<tr>
<td>VMOV32 VSTATUS, mem32</td>
<td></td>
</tr>
<tr>
<td>VMOV32 VTa, mem32</td>
<td></td>
</tr>
</tbody>
</table>
VMOV32 VSTATUS, mem32  Load VCU Status Register

**Operands**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VSTATUS</td>
<td>VCU status register</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 1011 0000  
MSW: 0000 0000 mem32

**Description**

Load the VSTATUS register with the 32-bit value in memory pointed to by mem32.

VSTATUS = [mem32];

**Flags**

This instruction modifies all bits within the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

VMOV32 mem32, VSTATUS  
VMOV32 mem32, VTa  
VMOV32 VRa, mem32  
VMOV32 VTa, mem32
VMOV32 VTa, mem32 — Load 32-bit Transition Bit Register

VMOV32 VTa, mem32  Load 32-bit Transition Bit Register

Operands

<table>
<thead>
<tr>
<th>VTa</th>
<th>Transition bit register: VT0, VT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 0001

MSW: 0000 00tt mem32

Description

Load the specified transition bit register with the 32-bit value in memory pointed to by mem32.

VTa = [mem32];

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VRa, mem32
VMOV32 VSTATUS, mem32
VMOVD32 VRa, mem32  
**Load Register with Data Move**

### Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register, VR0, VR1,..., VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 0010 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 aaaa mem32</td>
</tr>
</tbody>
</table>

### Description

Load the specified general purpose register with the 32-bit value in memory pointed to by mem32. In addition, copy the next 32-bit value in memory to the location pointed to by mem32.

VRa = [mem32];
[mem32 + 2] = [mem32];

### Flags

This instruction does not modify any flags in the VSTATUS register.

### Pipeline

This is a single-cycle instruction.

### See also
VMOVIX VRa, #16I — Load Upper Half of a General Purpose Register with 16-bit Immediate

Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register, VR0, VR1... VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16I</td>
<td>16-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 1110 IIII
MSW: IIII IIII IIII aaaa

Description

Load the upper 16-bits of the specified general purpose register with an immediate value. Leave the lower 16-bits of the register unchanged.

VRa[15:0] = unchanged;
VRa[31:16] = #16I;

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VMOVZI VRa, #16I
VMOVXI VRa, #16I
VMOVZI VRa, #16I — Load General Purpose Register with Immediate

Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register, VR0, VR1...VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16I</td>
<td>16-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 1111 IIII
MSW: IIII IIII IIII aaaa

Description

Load the lower 16-bits of the specified general purpose register with an immediate value. Clear the upper 16-bits of the register.

VRa[15:0] = #16I;
VRa[31:16] = 0x0000;

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VMOVIX VRa, #16I
VMOVXI VRa, #16I
VMOVXI VRa, #16I — Load Low Half of a General Purpose Register with Immediate

**Operands**

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register, VR0 - VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16I</td>
<td>16-bit immediate value</td>
</tr>
</tbody>
</table>

**Opcode**

- LSW: 1110 0111 0111 IIII
- MSW: IIII IIII IIII aaaa

**Description**

Load the lower 16-bits of the specified general purpose register with an immediate value. Leave the upper 16 bits unchanged.

VRa[15:0] = #16I;
VRa[31:16] = unchanged;

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

**See also**

- VMOVIX VRa, #16I
- VMOVZI VRa, #16I
VRNDOFF  Disable Rounding

Operands  none

Opcode  LSW: 1110 0101 0000 1001

Description  This instruction disables the rounding mode by clearing the RND bit in the VSTATUS register. When rounding is disabled, the result of the shift right operation for addition and subtraction operations will be truncated instead of rounded. The operations affected by rounding are shown in Table 3-6. Refer to the individual instruction descriptions for information on how rounding effects the operation. To enable rounding use the VRNDON instruction.

For more information on rounding, refer to .

VSTATUS[RND] = 0;

Flags  This instruction clears the RND bit in the VSTATUS register. It does not change any flags.

Pipeline  This is a single-cycle instruction.

Example

See also  VCLROVFI
VCLROVFR
VRNDON
VSATFOFF
VSATON
**VRNDON — Enable Rounding**

**Operands**

none

**Opcode**

LSW: 1110 0101 0000 1000

**Description**

This instruction enables the rounding mode by setting the RND bit in the VSTATUS register. When rounding is enabled, the result of the shift right operation for addition and subtraction operations will be rounded instead of being truncated. The operations affected by rounding are shown in Table 3-6. Refer to the individual instruction descriptions for information on how rounding effects the operation. To disable rounding use the VRNDOFF instruction.

For more information on rounding, refer to .

VSTATUS[RND] = 1;

**Flags**

This instruction sets the RND bit in the VSTATUS register. It does not change any flags.

**Pipeline**

This is a single-cycle instruction.

**Example**

See also

VCLROVFI
VCLROVFR
VRNDOFF
VSATFOFF
VSATON
**VSATOFF — Disable Saturation**

**Operands**
none

**Opcode**
LSW: 1110 0101 0000 0111

**Description**
This instruction disables the saturation mode by clearing the SAT bit in the VSTATUS register. When saturation is disabled, results of addition and subtraction are allowed to overflow or underflow. When saturation is enabled, results will instead be set to a maximum or minimum value instead of being allowed to overflow or underflow. To enable saturation use the **VSATON** instruction.

VSTATUS[SAT] = 0

**Flags**
This instruction clears the SAT bit in the VSTATUS register. It does not change any flags.

**Pipeline**
This is a single-cycle instruction.

**Example**

**See also**
- VCLROVFI
- VCLROVFR
- VRNDOFF
- VRNDON
- VSATON
VSATON — Enable Saturation

VSATON

Operands
none

Opcode
LSW: 1110 0101 0000 0110

Description
This instruction enables the saturation mode by setting the SAT bit in the VSTATUS register. When saturation is enabled, results of addition and subtraction are not allowed to overflow or underflow. Results will, instead, be set to a maximum or minimum value. To disable saturation use the VSATOFF instruction.

VSTATUS[SAT] = 1

Flags
This instruction sets the SAT bit in the VSTATUS register. It does not change any flags.

Pipeline
This is a single-cycle instruction.

See also
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATOFF
VSETSHL #5-bit — Initialize the Left Shift Value

**Operands**

| #5-bit | 5-bit, unsigned, immediate value |

**Opcode**

LSW: 1110 0101 110s ssss

**Description**

Load VSTATUS[SHIFTL] with an unsigned, 5-bit, immediate value. The left shift value specifies the number of bits an operand is shifted by. A value of zero indicates no shift will be performed. The left shift is used by the and VCDSUB16 and VCDADD16 operations. Refer to the description of these instructions for more information. To load the right shift value use the VSETSHR #5-bit instruction.

VSTATUS[VSHIFTL] = #5-bit

**Flags**

This instruction changes the VSHIFTL value in the VSTATUS register. It does not change any flags.

**Pipeline**

This is a single-cycle instruction.

**Example**

See also

VSETSHR #5-bit
## VSETSHR #5-bit — Initialize the Left Shift Value

### Operands

| #5-bit | 5-bit, unsigned, immediate value |

### Opcode

`LSW: 1110 0101 010s ssss`

### Description

Load `VSTATUS[SHIFTR]` with an unsigned, 5-bit, immediate value. The right shift value specifies the number of bits an operand is shifted by. A value of zero indicates no shift will be performed. The right shift is used by the VCADD, VCSUB, VCDADD16 and VCDSUB16 operations. It is also used by the addition portion of the VCMAC. Refer to the description of these instructions for more information.

`VSTATUS[VSHIFTR] = #5-bit`

### Flags

This instruction changes the `VSHIFTR` value in the `VSTATUS` register. It does not change any flags.

### Pipeline

This is a single-cycle instruction.

### Example

See also

- [VSETSHL #5-bit](#)
### 3.6.3 Complex Math Instructions

The instructions are listed alphabetically, preceded by a summary.

**Table 3-11. Complex Math Instructions**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCADD VR5, VR4, VR3, VR2 — Complex 32 + 32 = 32 Addition</td>
<td>390</td>
</tr>
<tr>
<td>VCADD VR5, VR4, VR3, VR2</td>
<td></td>
</tr>
<tr>
<td>VCADD VR7, VR6, VR5, VR4 — Complex 32 + 32 = 32 Addition</td>
<td>394</td>
</tr>
<tr>
<td>VCDADD16 VR5, VR4, VR3, VR2 — Complex 16 + 32 = 16 Addition</td>
<td>396</td>
</tr>
<tr>
<td>VCDADD16 VR5, VR4, VR3, VR2</td>
<td></td>
</tr>
<tr>
<td>VCDSUB16 VR6, VR4, VR3, VR2 — Complex 16-32 = 16 Subtract</td>
<td>402</td>
</tr>
<tr>
<td>VCDSUB16 VR6, VR4, VR3, VR2</td>
<td></td>
</tr>
<tr>
<td>VCMAC VR5, VR4, VR3, VR2, VR1, VR0 — Complex Multiply and Accumulate</td>
<td>408</td>
</tr>
<tr>
<td>VCMAC VR5, VR4, VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ — Complex Multiply and Accumulate</td>
<td>412</td>
</tr>
<tr>
<td>VCMPY VR3, VR2, VR1, VR0 — Complex Multiply</td>
<td>416</td>
</tr>
<tr>
<td>VCMPY VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCMPY VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VNEG VRa — Two's Complement Negate</td>
<td>422</td>
</tr>
<tr>
<td>VCSUB VR5, VR4, VR3, VR2 — Complex 32 - 32 = 32 Subtraction</td>
<td>423</td>
</tr>
<tr>
<td>VCSUB VR5, VR4, VR3, VR2</td>
<td></td>
</tr>
</tbody>
</table>
VCADD VR5, VR4, VR3, VR2  Complex 32 + 32 = 32 Addition

Operands

Before the operation, the inputs should be loaded into registers as shown below. Each operand for this instruction includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) + (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) + (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0000 0010

Description

Complex 32 + 32 = 32-bit addition operation.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFTR] bits before the addition. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
//
// X: VR5 = Re(X) VR4 = Im(X)
// Y: VR3 = Re(Y) VR2 = Im(Y)
//
// Calculate Z = X + Y
//
// if (RND == 1)
// {
//     VR5 = VR5 + round(VR3 >> SHIFTR); // Re(Z)
//     VR4 = VR4 + round(VR2 >> SHIFTR); // Im(Z)
// }
// else
// {
//     VR5 = VR5 + (VR3 >> SHIFTR); // Re(Z)
//     VR4 = VR4 + (VR2 >> SHIFTR); // Im(Z)
// }
// if (SAT == 1)
// {
//     sat32(VR5);
//     sat32(VR4);
// }
```

Flags

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR5 computation (real part) overflows or underflows.
- OVFI is set if the VR4 computation (imaginary part) overflows or underflows.

Pipeline

This is a single-cycle instruction.
VCADD VR5, VR4, VR3, VR2 — Complex 32 + 32 = 32 Addition

Example

See also

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCLROVF
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR #5-bit
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex 32+32 = 32 Add with Parallel Load

Operands
Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) + (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) + (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VRa</td>
<td>contents of the memory pointed to by [mem32]. VRa can not be VR5, VR4 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 0011 1111 1000
- MSW: 0000 aaaa mem32

Description

Complex 32 + 32 = 32-bit addition operation with parallel register load.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFTR] bits before the addition. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

In parallel with the addition, VRa is loaded with the contents of memory pointed to by mem32.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
//
// VR5 = Re(X) VR4 = Im(X)
// VR3 = Re(Y) VR2 = Im(Y)
//
// Z = X + Y
//
if (RND == 1)
{
    VR5 = VR5 + round(VR3 >> SHIFTR); // Re(Z)
    VR4 = VR4 + round(VR2 >> SHIFTR); // Im(Z)
}
else
{
    VR5 = VR5 + (VR3 >> SHIFTR); // Re(Z)
    VR4 = VR4 + (VR2 >> SHIFTR); // Im(Z)
}
if (SAT == 1)
{
    sat32(VR5);
    sat32(VR4);
}
VRa = [mem32];
```
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex 32+32 = 32 Add with Parallel Load

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR5 computation (real part) overflows.
- OVFI is set if the VR4 computation (imaginary part) overflows.

Pipeline
Both operations complete in a single cycle (1/1 cycles).

Example

See also
VCADD VR7, VR6, VR5, VR4
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR #5-bit
VCADD VR7, VR6, VR5, VR4  Complex 32 + 32 = 32- Addition

Operands

Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR6</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR7 and VR6 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) + (Re(Y) &gt;&gt; SHIFTR)</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) + (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR6</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) + (Re(Y) &gt;&gt; SHIFTR)</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) + (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0010 1010

Description

Complex 32 + 32 = 32-bit addition operation.

The second input operand (stored in VR5 and VR4) is shifted right by VSTATUS[SHIFTR] bits before the addition. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```c
// RND    is VSTATUS[RND]
// SAT    is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
//
// VR5 = Re(X)    VR4 = Im(X)
// VR3 = Re(Y)    VR2 = Im(Y)
//
// Z = X + Y
//
// if (RND == 1)
// {
//    VR7 = VR7 + round(VR5 >> SHIFTR); // Re(Z)
//    VR6 = VR6 + round(VR4 >> SHIFTR); // Im(Z)
// }
// else
// {
//    VR7 = VR5 + (VR5 >> SHIFTR);     // Re(Z)
//    VR6 = VR4 + (VR4 >> SHIFTR);     // Im(Z)
// }
// if (SAT == 1)
// {
//    sat32(VR7);
//    sat32(VR6);
// }
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR7 computation (real part) overflows.
- OVFI is set if the VR6 computation (imaginary part) overflows.

Pipeline

This is a single-cycle instruction.
See also

- VCADD VR5, VR4, VR3, VR2
- VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
- VCLROVFI
- VCLROVFR
- VRNDOFF
- VRNDON
- VSATON
- VSATOFF
- VSETSHR #5-bit
VCDADD16 VR5, VR4, VR3, VR2  Complex 16 + 32 = 16 Addition

Operands
Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the second input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the second input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5H</td>
<td>16-bit integer representing the real part of the result: Re(Z) = (Re(X) &lt;&lt; SHIFTL) + (Re(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VR5L</td>
<td>16-bit integer representing the imaginary part of the result: Im(Z) = (Im(X) &lt;&lt; SHIFTL) + (Im(Y) ) &gt;&gt; SHIFTR</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0000 0100

Description
Complex 16 + 32 = 16-bit operation. This operation is useful for algorithms similar to a complex FFT. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the addition is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

```
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit
//
// Calculate Z = X + Y
//
// temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
// temp2 = sign_extend(VR4L); // 32-bit extended Im(X)
// temp1 = (temp1 << SHIFTL) + VR3; // Re(Z) intermediate
// temp2 = (temp2 << SHIFTL) + VR2; // Im(Z) intermediate

if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
    temp2 = round(temp2 >> SHIFTR);
}
else
{
    temp1 = truncate(temp1 >> SHIFTR);
    temp2 = truncate(temp2 >> SHIFTR);
}
```
if (SAT == 1)
{
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
}
else
{
    VR5H = temp1[15:0];
    VR5L = temp2[15:0];
}

Flags
This instruction modifies the following bits in the VSTATUS register:
• OVFR is set if the real-part computation (VR5H) overflows or underflows.
• OVFI is set if the imaginary-part computation (VR5L) overflows or underflows.

Pipeline
This is a single-cycle instruction.

Example

; Example: Z = X + Y
; X = 4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 + 12j (32-bit real + 32-bit imaginary)
;
; Real:
; temp1 = 0x00000004 + 0x0000000D = 0x00000011
; VR5H = temp1[15:0] = 0x0011 = 17
; Imaginary:
; temp2 = 0x00000003 + 0x0000000C = 0x0000000F
; VR5L = temp2[15:0] = 0x000F = 15
;
; VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR #0 ; VSTATUS[SHIFTR] = 0
VSETSHL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13
VMOVXI VR2, #12 ; VR2 = Im(Y) = 12
VMOVXI VR4, #3
VMOVIX VR4, #4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x0011000F = 17 + 15j

The next example illustrates the operation with a right shift value defined.

; Example: Z = X + Y with Right Shift
;
; X = 4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 + 12j (32-bit real + 32-bit imaginary)
;
; Real:
; temp1 = (0x00000004 + 0x0000000D ) >> 1
; temp1 = (0x00000011) >> 1 = 0x0000008.8
; VR5H = temp1[15:0] = 0x0008 = 8
; Imaginary:
; temp2 = (0x00000003 + 0x0000000C ) >> 1
; temp2 = (0x0000000F) >> 1 = 0x0000007.8
; VR5L = temp2[15:0] = 0x0007 = 7
;
; VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13
VMOVXI VR2, #12 ; VR2 = Im(Y) = 12
VCDADD16 VR5, VR4, VR3, VR2 — Complex 16 + 32 = 16 Addition

VMOVXI VR4, #3
VMOVIX VR4, #4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x00080007 = 8 + 7j

The next example illustrates the operation with a right shift value defined as well as rounding.

;
; Example: Z = X + Y with Right Shift and Rounding
;
; X = 4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 + 12j (32-bit real + 32-bit imaginary)
;
; Real:
;   temp1 = round((0x00000004 + 0x0000000D ) >> 1)
;   temp1 = round(0x00000011 >> 1)
;   temp1 = round(0x00000008.8) = 0x00000009
;   VR5H = temp1[15:0] = 0x0011 = 8
; Imaginary:
;   temp2 = round(0x00000003 + 0x0000000C ) >> 1)
;   temp2 = round(0x0000000F >> 1)
;   temp2 = round(0x0000007.8) = 0x00000008
;   VR5L = temp2[15:0] = 0x0008 = 8
;
; VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETHS #1 ; VSTATUS[SHIFTR] = 1
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13
VMOVXI VR2, #12 ; VR2 = Im(Y) = 12
VMOVXI VR4, #3
VMOVIX VR4, #4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x00090008 = 9 + 8j

The next example illustrates the operation with both a right and left shift value defined along with rounding.

;
; Example: Z = X + Y with Right Shift, Left Shift and Rounding
;
; X = -4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 - 9j (32-bit real + 32-bit imaginary)
;
; Real:
;   temp1 = 0xFFFFFFFC << 2 + 0x0000000D
;   temp1 = 0xFFFFFFF0 + 0x0000000D = 0xFFFFFFFD
;   temp1 = 0xFFFFFFFD >> 1 = 0xFFFFFFFE.8
;   temp1 = round(0xFFFFFFFFE.8) = 0xFFFFFFFF
;   VR5H = temp1[15:0] 0xFFFF = -1;
; Imaginary:
;   temp2 = 0x00000003 << 2 + 0xFFFFFFF7
;   temp2 = 0x0000000C + 0xFFFFFFF7 = 0x00000003
;   temp2 = 0x00000003 >> 1 = 0x00000001.8
;   temp1 = round(0x00000001.8 = 0x00000002
;   VR5L = temp2[15:0] 0x0002 = 2
;
; VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHL #2 ; VSTATUS[SHIFTL] = 2
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, #-9 ; VR2 = Im(Y) = -9
VMOVIX VR2, #0xFFFF ; sign extend VR2 = 0xFFFFFFF7
VMOVXI VR4, #3
VMOVIX VR4, #-4 ; VR4 = X = 0xFFF00003 = -4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0xFFFF0002 = -1 + 2j
See also

VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETHL #5-bit
VSETHR #5-bit
VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex Double Add with Parallel Load

Operands
Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| VR5H            | 16-bit integer representing the real part of the result: 
                      Re(Z) = (Re(X) << SHIFTL) + (Re(Y)) >> SHIFTR |
| VR5L            | 16-bit integer representing the imaginary part of the result: 
                      Im(Z) = (Im(X) << SHIFTL) + (Im(Y)) >> SHIFTR |
| VRa             | Contents of the memory pointed to by [mem32]. VRa can not be VR5 or VR8. |

Opcode

LSW: 1110 0011 1111 1010
MSW: 0000 aaaa mem32

Description
Complex 16 + 32 = 16-bit operation with parallel register load. This operation is useful for algorithms similar to a complex FFT.

The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the addition is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]

// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

temp1 = sign_extend(VR4H);  // 32-bit extended Re(X)
temp2 = sign_extend(VR4L);  // 32-bit extended Im(X)
temp1 = (temp1 << SHIFTL) + VR3;  // Re(Z) intermediate
temp2 = (temp2 << SHIFTL) + VR2;  // Im(Z) intermediate

if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
    temp2 = round(temp2 >> SHIFTR);
}
else
{
    temp1 = truncate(temp1 >> SHIFTR);
}
```
temp2 = truncate(temp2 >> SHIFTR);
}
if (SAT == 1)
{
  VRSH = sat16(temp1);
  VRSL = sat16(temp2);
}
else
{
  VRSH = temp1[15:0];
  VRSL = temp2[15:0];
}
VRa = [mem32];

Flags
This instruction modifies the following bits in the VSTATUS register:
• OVFR is set if the real-part (VR5H) computation overflows or underflows.
• OVFI is set if the imaginary-part (VR5L) computation overflows or underflows.

Pipeline
Both operations complete in a single cycle.

Example
For more information regarding the addition operation, please refer to the examples for the VCDADD16 VR5, VR4, VR3, VR2 instruction.

; ;Example: Right Shift, Left Shift and Rounding
; ; X = -4 + 3j (16-bit real + 16-bit imaginary)
; ; Y = 13 - 9j (32-bit real + 32-bit imaginary)
;
; ; Real:
; ; temp1 = 0xFFFFFFFC << 2 + 0x0000000D
; ; temp1 = 0xFFFFFF0 + 0x0000000D = 0xFFFFFFFF
; ; temp1 = 0xFFFFFFFD >> 1 = 0xFFFFFFFE.8
; ; temp1 = round(0xFFFFFFFFE.8) = 0xFFFFFFFF
; ; VR5H = temp1[15:0] 0xFFFF = -1;
; ; Imaginary:
; ; temp2 = 0x00000003 << 2 + 0xFFFFFFF7
; ; temp2 = 0x0000000C + 0xFFFFFFF7 = 0x00000003
; ; temp2 = 0x00000001.8
; ; temp1 = round(0x000000001.8) = 0x00000002
; ; VR5L = temp2[15:0] 0x0002 = 2

VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #2 ; VSTATUS[SHIFTL] = 2
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, #9 ; VR2 = Im(Y) = -9
VMOVIX VR2, #0xFFFF ; sign extend VR2 = 0xFFFFFFFF
VMOVIX VR4, #3
VMOVIX VR4, #4 ; VR4 = X = 0xFFFC0003 = -4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0xFFFF0002 = -1 + 2j
|| VCMOV32 VR2, *XAR7 ; VR2 = value pointed to by XAR7

See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHL #5-bit
VSETSHR #5-bit
VCDSUB16 VR6, VR4, VR3, VR2 — Complex 16-32 = 16 Subtract

Operands
Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR6 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR6H</td>
<td>16-bit integer representing the real part of the result: Re(Z) = (Re(X) &lt;&lt; SHIFTL) - (Re(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VR6L</td>
<td>16-bit integer representing the imaginary part of the result: Im(Z) = (Im(X) &lt;&lt; SHIFTL) - (Im(Y)) &gt;&gt; SHIFTR</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0000 0101

Description

Complex 16 - 32 = 16-bit operation. This operation is useful for algorithms similar to a complex FFT.

The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[SHIFTL] bits. The result of the subtraction is left shifted by VSTATUS[SHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

// 32-bit extended Re(X)
temp1 = sign_extend(VR4H);
temp2 = sign_extend(VR4L);

// 32-bit intermediate
temp1 = (temp1 << SHIFTL) - VR3; // Re(Z) intermediate
temp2 = (temp2 << SHIFTL) - VR2; // Im(Z) intermediate

if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
    temp2 = round(temp2 >> SHIFTR);
}
else
{
    temp1 = truncate(temp1 >> SHIFTR);
    temp2 = truncate(temp2 >> SHIFTR);
}

if (SAT == 1)
{
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
}
```
VCDSUB16 VR6, VR4, VR3, VR2 — Complex 16-32 = 16 Subtract

```
VR5L = sat16(temp2);
}
else
{
  VR5H = temp1[15:0];
  VR5L = temp2[15:0];
}
```

### Flags
This instruction modifies the following bits in the VSTATUS register:

- **OVFR** is set if the real-part (VR6H) computation overflows or underflows.
- **OVFI** is set if the imaginary-part (VR6L) computation overflows or underflows.

### Pipeline
This is a single-cycle instruction.

### Example
```
; Example: Z = X - Y
; X = 4 + 6j (16-bit real + 16-bit imaginary)
; Y = 13 + 22j (32-bit real + 32-bit imaginary)
; Z = (4 - 13) + (6 - 22)j = -9 - 16j

VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETS #0 ; VSTATUS[SHIFT] = 0
VSETS #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 = 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVIX VR4, #4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0xFFF7FFF0 = -9 + -16j
```

### Example
```
; Example: Z = X - Y with Right Shift
; Y = 4 + 6j (16-bit real + 16-bit imaginary)
; X = 13 + 22j (32-bit real + 32-bit imaginary)

; Real:
; temp1 = (0x00000004 - 0x0000000D) >> 1
; temp1 = (0xFFFFFFFF7) >> 1
; temp1 = 0xFFFFFFFFB
; VR5H = temp1[15:0] = 0xFFFB = -5

; Imaginary:
; temp2 = (0x00000006 - 0x00000016) >> 1
; temp2 = (0xFFFFFFFF0) >> 1
; temp2 = 0xFFFFFFFF8
; VR5L = temp2[15:0] = 0xFFF8 = -8

VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETS #1 ; VSTATUS[SHIFT] = 1
VSETS #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 = 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVIX VR4, #4
; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0xFFF7FFF8 = -5 + -8j
```

### Example
```
The next example illustrates the operation with a right shift value defined.

; Example: Z = X - Y with Right Shift
; Y = 4 + 6j (16-bit real + 16-bit imaginary)
; X = 13 + 22j (32-bit real + 32-bit imaginary)

; Real:
; temp1 = (0x00000004 - 0x0000000D) >> 1
; temp1 = (0xFFFFFFFF7) >> 1
; temp1 = 0xFFFFFFFFB
; VR5H = temp1[15:0] = 0xFFFB = -5

; Imaginary:
; temp2 = (0x00000006 - 0x00000016) >> 1
; temp2 = (0xFFFFFFFF0) >> 1
; temp2 = 0xFFFFFFFF8
; VR5L = temp2[15:0] = 0xFFF8 = -8

VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETS #1 ; VSTATUS[SHIFT] = 1
VSETS #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 = 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVIX VR4, #4
; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0xFFF7FFF8 = -5 + -8j
```

### Example
```
The next example illustrates rounding with a right shift value defined.

; Example: Z = X - Y with Right Shift
; Y = 4 + 6j (16-bit real + 16-bit imaginary)
; X = 13 + 22j (32-bit real + 32-bit imaginary)

; Real:
; temp1 = (0x00000004 - 0x0000000D) >> 1
; temp1 = (0xFFFFFFFF7) >> 1
; temp1 = 0xFFFFFFFFB
; VR5H = temp1[15:0] = 0xFFFB = -5

; Imaginary:
; temp2 = (0x00000006 - 0x00000016) >> 1
; temp2 = (0xFFFFFFFF0) >> 1
; temp2 = 0xFFFFFFFF8
; VR5L = temp2[15:0] = 0xFFF8 = -8

VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETS #1 ; VSTATUS[SHIFT] = 1
VSETS #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 = 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVIX VR4, #4
; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0xFFF7FFF8 = -5 + -8j
```
VCDSUB16 VR6, VR4, VR3, VR2 — Complex 16-32 = 16 Subtract

; Example: Z = X-Y with Rounding and Right Shift
;
; X = 4 + 6j  (16-bit real + 16-bit imaginary)
; Y = -13 + 22j  (32-bit real + 32-bit imaginary)
;
; Real:
;  temp1 = round((0x00000004 - 0xFFFFFFF3) >> 1)
;  temp1 = round((0x00000010 - 0xFFFFFFF3) >> 1)
;  temp1 = round((0x00000001D >> 1)
;  temp1 = round((0x0000000E.8) = 0x0000000F
;  VR5H = temp1[15:0] = 0x000F = 15
;
; Imaginary:
;  temp2 = round((0x00000006 << 2 - 0x00000016) >> 1)
;  temp2 = round((0x00000012 - 0x00000016) >> 1)
;  temp2 = round((0x000000022 >> 1)
;  temp2 = round((0x00000001.0) = 0x00000001
;  VR5L = temp2[15:0] = 0x0001 = 1
;

VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #-13 ; VR3 = Re(Y)
VMOVIX VR3, #0xFFFF ; sign extend VR3 = -13 = 0xFFFFFFF3
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVIX VR4, #4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0x0009F001 = 9 + -8j

The next example illustrates rounding with both a left and a right shift value defined.

; Example: Z = X-Y with Rounding and both Left and Right Shift
;
; X = 4 + 6j  (16-bit real + 16-bit imaginary)
; Y = -13 + 22j  (32-bit real + 32-bit imaginary)
;
; Real:
;  temp1 = round((0x00000004 << 2 - 0xFFFFFFF3) >> 1)
;  temp1 = round((0x00000010 - 0xFFFFFFF3) >> 1)
;  temp1 = round((0x00000001D >> 1)
;  temp1 = round((0x0000000E.8) = 0x0000000F
;  VR5H = temp1[15:0] = 0x000F = 15
;
; Imaginary:
;  temp2 = round((0x00000006 << 2 - 0x00000016) >> 1)
;  temp2 = round((0x00000012 - 0x00000016) >> 1)
;  temp2 = round((0x000000022 >> 1)
;  temp2 = round((0x00000001.0) = 0x00000001
;  VR5L = temp2[15:0] = 0x0001 = 1
;

VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #2 ; VSTATUS[SHIFTL] = 2
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #-13 ; VR3 = Re(Y)
VMOVIX VR3, #0xFFFF ; sign extend VR3 = -13 = 0xFFFFFFF3
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVIX VR4, #4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0x0009F001 = 9 + -8j

See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VCDSUB16 VR6, VR4, VR3, VR2 — Complex 16-32 = 16 Subtract

VRNDON
VSATON
VSATOFF
VSETHL #5-bit
VSETHR #5-bit
### Operands

Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR6 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR6H</td>
<td>16-bit integer representing the real part of the result: Re(Z) = (Re(X) &lt;&lt; SHIFTL) + (Re(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VR6L</td>
<td>16-bit integer representing the imaginary part of the result: Im(Z) = (Im(X) &lt;&lt; SHIFTL) + (Im(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VRa</td>
<td>Contents of the memory pointed to by [mem32]. VRa can not be VR6 or VR8.</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW:</th>
<th>1110 0010 1100 1010</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW:</td>
<td>0000 0000 mem16</td>
</tr>
</tbody>
</table>

### Description

Complex 16 - 32 = 16-bit operation with parallel load. This operation is useful for algorithms similar to a complex FFT.

The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the subtraction is left shifted by VSTATUS[SHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
temp2 = sign_extend(VR4L); // 32-bit extended Im(X)

if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
    temp2 = round(temp2 >> SHIFTR);
}
else
{
    temp1 = truncate(temp1 >> SHIFTR);
    temp2 = truncate(temp2 >> SHIFTR);
}
if (SAT == 1)
```
VCDSUB16 VR6, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex 16+32 = 16 Add with Parallel Load

```c
{
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
}
else
{
    VR5H = temp1[15:0];
    VR5L = temp2[15:0];
}
VRa = [mem32];
```

**Flags**

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the real-part (VR6H) computation overflows or underflows.
- OVFI is set if the imaginary-part (VR6I) computation overflows or underflows.

**Pipeline**

Both operations complete in a single cycle.

**Example**

For more information regarding the subtraction operation, please refer to VCDSUB16 VR6, VR4, VR3, VR2.

```c
; Example: Z = X - Y with Rounding and both Left and Right Shift
; X = 4 + 6j (16-bit real + 16-bit imaginary)
; Y = -13 + 22j (32-bit real + 32-bit imaginary)
; Real:
; temp1 = round((0x00000004 << 2 - 0xFFFFFFF3) >> 1)
; temp1 = round((0x00000010 - 0xFFFFFFF3) >> 1)
; temp1 = round(0x00000001D >> 1)
; temp1 = round(0x0000000E.8) = 0x0000000F
; VR5H = temp1[15:0] = 0x000F = 15
; Imaginary:
; temp2 = round((0x00000006 << 2 - 0x00000016) >> 1)
; temp2 = round((0x00000018 - 0x00000016) >> 1)
; temp2 = round(0x00000002 >> 1)
; temp1 = round(0x00000001.0) = 0x00000001
; VR5L = temp2[15:0] = 0x0001 = 1
;
VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 1
VSETSHL #1 ; VSTATUS[SHIFTR] = 1
VCLRALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #-13 ; VR3 = Re(Y)
VMOVX VR3, #0xFFFF ; sign extend VR3 = -13 = 0xFFFFFFF3
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVX VR4, #4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0x000F0001 = 15 + 1j
|| VCMOV32 VR2, *XAR7 ; VR2 = contents pointed to by XAR7
See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VRNDOFF
VSETSHL #5-bit
VSETSHL #5-bit
```
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 — Complex Multiply and Accumulate

### Operands

Before the operation, the inputs should be loaded into registers as shown below.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer, previous real-part accumulation</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer, previous imaginary-part accumulation</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer, real result from the previous multiply</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer, imaginary result from the previous multiply</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR0L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit integer representing the real part of the second input: Re(Y)</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit integer representing the imaginary part of the second input: Im(Y)</td>
</tr>
</tbody>
</table>

**Note:** The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.

The result is stored as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit real part of the total accumulation Re(sum) = Re(sum) + Re(mpy)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit imaginary part of the total accumulation Im(sum) = Im(sum) + Im(mpy)</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
</tr>
</tbody>
</table>

### Description

// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
// VR0 = X + jX: VR0[31:16] = X, VR0[15:0] = jX

// Perform add
if (RND == 1)
{
    VR5 = VR5 + round(VR3 >> SHIFTR);
    VR4 = VR4 + round(VR2 >> SHIFTR);
}
else
{
    VR5 = VR5 + (VR3 >> SHIFTR);
    VR4 = VR4 + (VR2 >> SHIFTR);
}

// Perform multiply (X + jX) * (Y * jY)

VR3 = VR0H * VR1H - VR0L * VR1L; Real result
VR2 = VR0H * VR1L + VR0L * VR1H; Imaginary result
if(SAT == 1)
{
    sat32(VR3);
    sat32(VR2);
}
VRa = [mem32];

### Flags

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

### Pipeline

This is a 2p-cycle instruction.
Example

See also  
VCLROVFI  
VCLROVFR  
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32  
VSATON  
VSATOFF
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32  — Complex Multiply and Accumulate with Parallel Load

**Operands**

Before the operation, the inputs should be loaded into registers as shown below.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>Previous real-part accumulation</td>
</tr>
<tr>
<td>VR4</td>
<td>Previous imaginary-part accumulation</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit real result from the previous multiply</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit imaginary result from the previous multiply</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR0L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit integer representing the real part of the second input: Re(Y)</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit integer representing the imaginary part of the second input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location.</td>
</tr>
</tbody>
</table>

**Note:** The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.

The result is stored as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit real part of the total accumulation Re(sum) = Re(sum) + Re(mpy)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit imaginary part of the total accumulation Im(sum) = Im(sum) + Im(mpy)</td>
</tr>
<tr>
<td>VRa</td>
<td>Contents of the memory pointed to by [mem32]. VRa cannot be VR5, VR4 or VR8</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 1100 1010  
MSW: 0000 0000 mem32  

**Description**

Complex multiply operation.

```c
// VR5 = Accumulation of the real part  
// VR4 = Accumulation of the imaginary part  
//  
// VR0 = X + Xj: VR0[31:16] = Re(X), VR0[15:0] = Im(X)  
// VR1 = Y + Yj: VR1[31:16] = Re(Y), VR1[15:0] = Im(Y)  
//  
// Perform add  
//  
// if (RND == 1)  
//  
//      VR5 = VR5 + round(VR3 >> SHIFTR);  
//      VR4 = VR4 + round(VR2 >> SHIFTR);  
//  
// else  
//  
//      VR5 = VR5 + (VR3 >> SHIFTR);  
//      VR4 = VR4 + (VR2 >> SHIFTR);  
//  
// Perform multiply Z = (X + Xj) * (Y * Yj)  
//  
// VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z)  
// VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z)  
// if(SAT == 1)  
//  
//      sat32(VR3);  
//      sat32(VR2);  
//  
// VRa = [mem32];
```
Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline

This is a 2p/1-cycle instruction. The multiply and accumulate is a 2p-cycle operation and the VMOV32 is a single-cycle operation.

Example

See also

VCLROVFI
VCLROVFR
VCMAC VR5, VR4, VR3, VR2, VR1, VR0
VSATON
VSATOFF
VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++  — Complex Multiply and Accumulate

Operands
The VMAC alternates which registers are used between each cycle. For odd cycles (1, 3, 5, etc) the following registers are used:

<table>
<thead>
<tr>
<th>Odd Cycle Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>Previous real-part total accumulation: Re(odd_sum)</td>
</tr>
<tr>
<td>VR4</td>
<td>Previous imaginary-part total accumulation: Im(odd_sum)</td>
</tr>
<tr>
<td>VR1</td>
<td>Previous real result from the multiply: Re(odd_mpy)</td>
</tr>
<tr>
<td>VR0</td>
<td>Previous imaginary result from the multiply: Im(odd_mpy)</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Pointer to a 32-bit memory location representing the first input to the multiply</td>
</tr>
<tr>
<td></td>
<td>[mem32][31:16] = Re(X)</td>
</tr>
<tr>
<td></td>
<td>[mem32][15:0] = Im(X)</td>
</tr>
<tr>
<td>XAR7</td>
<td>Pointer to a 32-bit memory location representing the second input to the multiply</td>
</tr>
<tr>
<td></td>
<td>*XAR7[31:16] = Re(Y)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[15:0] = Im(Y)</td>
</tr>
</tbody>
</table>

The result from odd cycle is stored as shown below:

<table>
<thead>
<tr>
<th>Odd Cycle Output</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit real part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Re(odd_sum) = Re(odd_sum) + Re(odd_mpy)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit imaginary part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Im(sum) = Im(odd_sum) + Im(odd_mpy)</td>
</tr>
<tr>
<td>VR1</td>
<td>32-bit real result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)</td>
</tr>
<tr>
<td>VR0</td>
<td>32-bit imaginary result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = Re(X)*Im(Y) + Re(Y)*Im(X)</td>
</tr>
</tbody>
</table>

For even cycles (2, 4, 6, etc) the following registers are used:

<table>
<thead>
<tr>
<th>Even Cycle Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>Previous real-part total accumulation: Re(even_sum)</td>
</tr>
<tr>
<td>VR6</td>
<td>Previous imaginary-part total accumulation: Im(even_sum)</td>
</tr>
<tr>
<td>VR3</td>
<td>Previous real result from the multiply: Re(even_mpy)</td>
</tr>
<tr>
<td>VR2</td>
<td>Previous imaginary result from the multiply: Im(even_mpy)</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Pointer to a 32-bit memory location representing the first input to the multiply</td>
</tr>
<tr>
<td></td>
<td>[mem32][31:16] = Re(X); (a)</td>
</tr>
<tr>
<td></td>
<td>[mem32][15:0] = Im(X); (b)</td>
</tr>
<tr>
<td>XAR7</td>
<td>Pointer to a 32-bit memory location representing the second input to the multiply:</td>
</tr>
<tr>
<td></td>
<td>*XAR7[31:16] = Re(Y); (c)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[15:0] = Im(Y); (d)</td>
</tr>
</tbody>
</table>

The result from even cycles is stored as shown below:

<table>
<thead>
<tr>
<th>Even Cycle Output</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>32-bit real part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Re(even_sum) = Re(even_sum) + Re(even_mpy)</td>
</tr>
<tr>
<td>VR6</td>
<td>32-bit imaginary part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Im(even_sum) = Im(even_sum) + Im(even_mpy)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit real result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit imaginary result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = Re(X)*Im(Y) + Re(Y)*Im(X)</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0010 0101 0000
MSW: 00bb baaa mem32

Description
Perform a repeated multiply and accumulate operation. This instruction is the only VCU instruction that can be repeated using the single repeat instruction (RPT ||). When repeated, the destination of the accumulate will alternate between VR7/VR6 and VR5/VR4 on each cycle.
// Cycle 1:
// Perform accumulate
//
if(RND == 1)
{
    VR5 = VR5 + round(VR1 >> SHIFTR)
    VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
    VR5 = VR5 + (VR1 >> SHIFTR)
    VR4 = VR4 + (VR0 >> SHIFTR)
}
// X and Y array element 0
//
VR1 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) + Re(Y)*Im(X)

// Cycle 2:
// Perform accumulate
//
if(RND == 1)
{
    VR7 = VR7 + round(VR3 >> SHIFTR)
    VR6 = VR6 + round(VR2 >> SHIFTR)
}
else
{
    VR7 = VR7 + (VR3 >> SHIFTR)
    VR6 = VR6 + (VR2 >> SHIFTR)
}
// X and Y array element 1
//
VR3 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = Re(X)*Im(Y) + Re(Y)*Im(X)

// Cycle 3:
// Perform accumulate
//
if(RND == 1)
{
    VR5 = VR5 + round(VR1 >> SHIFTR)
    VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
    VR5 = VR5 + (VR1 >> SHIFTR)
    VR4 = VR4 + (VR0 >> SHIFTR)
}
// X and Y array element 2
//
VR1 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) + Re(Y)*Im(X)

etc...

Restrictions
VR0, VR1, VR2, and VR3 will be used as temporary storage by this instruction.

Flags
The VSTATUS register flags are modified as follows:

- OVFR is set in the case of an overflow or underflow of the addition or subtraction
operations.

- OVFI is set in the case an overflow or underflow of the imaginary part of the addition
  or subtraction operations.

Pipeline

When repeated the VMAC takes 2p + N cycles where N is the number of times the
instruction is repeated. When repeated, this instruction has the following pipeline
restrictions:

```
<instruction1> ; No restriction
<instruction2> ; Cannot be a 2p instruction that writes
                   to VR0, VR1...VR7 registers
RPT #(N-1) ; Execute N times, where N is even
| VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
<instruction3> ; No restrictions.
                   ; Can read VR0, VR1... VR8
```
MACF32 can also be used standalone. In this case, the instruction takes 2 cycles and the following pipeline restrictions apply:

\[
\begin{align*}
\text{<instruction1>} & ; \text{No restriction} \\
\text{<instruction2>} & ; \text{Cannot be a 2p instruction that writes to R2H, R3H, R6H or R7H} \\
& \text{MACF32 R7H, R3H, *XAR6, *XAR7} ; \text{R3H} = \text{R3H} + \text{R2H,} \\
& \text{R2H} = \text{[mem32]} \times \text{[XAR7++]} ; \text{<--} \\
& \text{R2H and R3H are valid (note: no delay required) NOP}
\end{align*}
\]

**Example**

Cascading of RPT || VMAC is allowed as long as the first and subsequent counts are even. Cascading is useful for creating interruptible windows so that interrupts are not delayed too long by the RPT instruction. For example:

```c
; Example of cascaded VMAC instructions
; VCLEARALL ; Zero the accumulation registers
; Execute MACF32 N+1 (4) times
; RPT #3 || VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
; Execute MACF32 N+1 (6) times
; RPT #5 || VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
; Repeat MACF32 N+1 times where N+1 is even
; RPT #N || MACF32 R7H, R3H, *XAR6++, *XAR7++
ADD F32 VR7, VR6, VR5, VR4
```

**See also**
VCMPY VR3, VR2, VR1, VR0 — Complex Multiply

Operands

Before the operation, the inputs should be loaded into registers as shown below. Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR0L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3</td>
<td>16-bit integer representing the real part of the result: Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>16-bit integer representing the imaginary part of the result: Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0000 0000

Description

Complex 16 x 16 = 32-bit multiply operation.

If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.

```
// VR0 = X + Xj: VR0[31:16] = Re(X), VR0[15:0] = Im(X)
// VR1 = Y + Yj: VR1[31:16] = Re(Y), VR1[15:0] = Im(Y)
// Calculate: Z = (X + jX) * (Y + jY)
// VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
if(SAT == 1)
{
    sat32(VR3);
    sat32(VR2);
}
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline

This is a 2p-cycle instruction. The instruction following this one should not use VR3 or VR2.

Example

```
; Example 1
; X = 4 + 6j
; Y = 12 + 9j
;
; Z = X * Y
; Re(Z) = 4*12 - 6*9 = -6
; Im(Z) = 4*9 + 6*12 = 108

VSATOFF ; VSTATUS[SAT] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXVR0, #6
VMOVXVR0, #4 ; VR0 = X = 0x00040006 = 4 + 6j
VMOVXVR1, #9
VMOVXVR1, #12 ; VR1 = Y = 0x000C0009 = 12 + 9j
VCMPYVR3, VR2, VR1, VR0 ; VR3 = Re(Z) = 0xFFFFFFFF = -6
                      ; VR2 = Im(Z) = 0x0000006C = 108
```
VCMPY VR3, VR2, VR1, VR0 — Complex Multiply

<instruction 1> ; <- Must not use VR2, VR3
<instruction 2> ; <- VCMPY completes, VR2, VR3 valid

See also
VCLROVF1
VCLROVF2
VCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSATON
VSATOFF
VCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa — Complex Multiply with Parallel Store

Operands
Before the operation, the inputs should be loaded into registers as shown below. Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR0L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit integer representing the real part of the second input: Re(Y)</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit integer representing the imaginary part of the second input: Im(Y)</td>
</tr>
<tr>
<td>VRa</td>
<td>Value to be stored.</td>
</tr>
</tbody>
</table>

The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3</td>
<td>16-bit integer representing the real part of the result: Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>16-bit integer representing the imaginary part of the result: Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)</td>
</tr>
</tbody>
</table>

[mem32] Contents of VRa. VRa can be VR0-VR7. VRa cannot be VR8.

Opcode
LSW: 1110 0010 1100 1010
MSW: 0000 0000 mem16

Description
Complex 16 x 16 = 32-bit multiply operation with parallel register load.

If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.

```c
// VR0 = X + jX: VR0[31:16] = Re(X), VR0[15:0] = Im(X)
// VR1 = Y + jY: VR1[31:16] = Re(Y), VR1[15:0] = Im(Y)
//
// Calculate: Z = (X + jX) * (Y + jY)
//
VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
if(SAT == 1)
{
    sat32(VR3);
    sat32(VR2);
}
VRa = [mem32];
```

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline
This is a 2p/1-cycle instruction. The multiply operation takes 2p cycles and the VMOV operation completes in a single cycle. The instruction following this one must not use VR2 or VR3.

Example
```c
; Example 1
; X = 4 + 6j
; Y = 12 + 9j
;
; Z = X * Y
; Re(Z) = 4*12 - 6*9 = -6
; Im(Z) = 4*9 + 6*12 = 108
;
VSATOFF ; VSTATUS[SAT] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
```
VCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa — Complex Multiply with Parallel Store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOVXI VR0, #6</td>
<td>VR0 = X = 0x00040006 = 4 + 6j</td>
</tr>
<tr>
<td>VMOVIX VR0, #4</td>
<td>VR0 = X = 0x00000004 = 4j</td>
</tr>
<tr>
<td>VMOVXI VR1, #9</td>
<td>VR1 = Y = 0x000C0009 = 12 + 9j</td>
</tr>
<tr>
<td>VMOVIX VR1, #12</td>
<td>VR1 = Y = 0x00000012 = 12j</td>
</tr>
<tr>
<td>VCMPY VR3, VR2, VR1, VR0</td>
<td>VR2 = Im(Z) = 0x0000000C = 108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV32 *XAR7, VR3</td>
<td>Location XAR7 points to = VR3 (before multiply)</td>
</tr>
</tbody>
</table>

See also

- VCLROVFI
- VCLROVFR
- VCMAC VR5, VR4, VR3, VR2, VR1, VR0
- VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
- VSATON
- VSATOFF
VCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32  Complex Multiply with Parallel Load

Operands

Before the operation, the inputs should be loaded into registers as shown below. Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0H</td>
<td>16-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR0L</td>
<td>16-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3</td>
<td>16-bit integer representing the real part of the result: Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>16-bit integer representing the imaginary part of the result: Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)</td>
</tr>
<tr>
<td>VRa</td>
<td>32-bit value pointed to by [mem32]. VRa can not be VR2, VR3 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 0110
MSW: 0000 aaaa mem32

Description

Complex 16 x 16 = 32-bit multiply operation with parallel register load.

If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.

```c
// VR0 = X + jX: VR0H[31:16] = Re(X), VR0L[15:0] = Im(X)
// VR1 = Y + jY: VR1H[31:16] = Re(Y), VR1L[15:0] = Im(Y)
// Calculate: Z = (X + jX) * (Y + jY)
// VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
if(SAT == 1)
{
    sat32(VR3);
    sat32(VR2);
}
VRa = [mem32];
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline

This is a 2p/1-cycle instruction. The multiply operation takes 2p cycles and the VMOV operation completes in a single cycle. The instruction following this one must not use VR2 or VR3.

Example

```c
; Example 1
; X = 4 + 6j
; Y = 12 + 9j
;
; Z = X * Y
; Re(Z) = 4*12 - 6*9 = -6
; Im(Z) = 4*9 + 6*12 = 108
;
VSATOFF       ; VSTATUS[SAT] = 0
VCLEARALL     ; VR0, VR1...VR8 == 0
```
VCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 — Complex Multiply with Parallel Load

```
VMOVXI VR0, #6
VMOVIX VR0, #4 ; VR0 = X = 0x00040006 = 4 + 6j
VMOVXI VR1, #9
VMOVIX VR1, #12 ; VR1 = Y = 0x000C0009 = 12 + 9j
VCMPY VR3, VR2, VR1, VR0 ; VR3 = Re(Z) = 0xFFFFFFFF = -6
VMOV32 VR3, VR2, VR1, VR0 ; VR2 = Im(Z) = 0x0000006C = 108

| VR0, *XAR7 | VR0 = contents of location XAR7 points to |
| <instruction 1> | <- Must not use VR2, VR3 |
| <instruction 2> | <- VCMPY completes, VR2, VR3 valid |
| Can use VR2, VR3 |
```

See also

- VCLROVFI
- VCLROVFR
- VCMAC VR5, VR4, VR3, VR2, VR1, VR0
- VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
- VSATON
- VSATOFF
**VNEG VRa — Two's Complement Negate**

### Operands

| VRa can be VR0 - VR7. VRa can not be VR8. |

### Opcode

LSW: 1110 0101 0001 aaaa

### Description

Complex add operation.

```c
// SAT is VSTATUS[SAT]
//
if (VRa == 0x800000000)
{
    if(SAT == 1)
    {
        VRa = 0x7FFFFFFF;
    }
    else
    {
        VRa = 0x80000000;
    }
}
else
{
    VRa = - VRa
}
```

### Flags

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the input to the operation is 0x80000000.

### Pipeline

This is a single-cycle instruction.

### See also

- VCLROVFR
- VSATON
- VSATOFF
VCSUB VR5, VR4, VR3, VR2  Complex 32 - 32 = 32 Subtraction

Operands

Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) - (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) - (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0000 0011

Description

Complex 32 - 32 = 32-bit subtraction operation.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFTR] bits before the subtraction. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]

if (RND == 1)
{
    VR5 = VR5 - round(VR3 >> SHIFTR);
    VR4 = VR4 - round(VR2 >> SHIFTR);
} else
{
    VR5 = VR5 - (VR3 >> SHIFTR);
    VR4 = VR4 - (VR2 >> SHIFTR);
}
if (SAT == 1)
{
    sat32(VR5);
    sat32(VR4);
}
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR5 computation (real part) overflows or underflows.
- OVFI is set if the VR6 computation (imaginary part) overflows or underflows.

Pipeline

This is a single-cycle instruction.

Example

See also

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCLROVFI
VCSUB VR5, VR4, VR3, VR2 — Complex 32 - 32 = 32 Subtraction

VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR #5-bit
VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32  Complex Subtraction

Operands

Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) - (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) - (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VRa</td>
<td>contents of the memory pointed to by [mem32]. VRa can not be VR5, VR4 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1100 1010
MSW: 0000 0000 mem16

Description

Complex 32 - 32 = 32-bit subtraction operation with parallel load.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFTR] bits before the subtraction. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in . If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
//
// if (RND == 1)
// {
//     VR5 = VR5 - round(VR3 >> SHIFTR);
//     VR4 = VR4 - round(VR2 >> SHIFTR);
// } else
// {
//     VR5 = VR5 - (VR3 >> SHIFTR);
//     VR4 = VR4 - (VR2 >> SHIFTR);
// }
// if (SAT == 1)
// {
//     sat32(VR5);
//     sat32(VR4);
// }
VRa = [mem32];
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR5 computation (real part) overflows or underflows.
- OVFI is set if the VR6 computation (imaginary part) overflows or underflows.

Pipeline

This is a single-cycle instruction.
VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex Subtraction

See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCSUB VR5, VR4, VR3, VR2
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETS #5-bit
### 3.6.4 Cyclic Redundancy Check (CRC) Instructions

The instructions are listed alphabetically, preceded by a summary.

**Table 3-12. CRC Instructions**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRC8H_1 mem16 — CRC8, High Byte</td>
<td>428</td>
</tr>
<tr>
<td>VCRC8L_1 mem16 — CRC8, Low Byte</td>
<td>429</td>
</tr>
<tr>
<td>VCRC16P1H_1 mem16 — CRC16, Polynomial 1, High Byte</td>
<td>430</td>
</tr>
<tr>
<td>VCRC16P1L_1 mem16 — CRC16, Polynomial 1, Low Byte</td>
<td>431</td>
</tr>
<tr>
<td>VCRC16P2H_1 mem16 — CRC16, Polynomial 2, High Byte</td>
<td>432</td>
</tr>
<tr>
<td>VCRC16P2L_1 mem16 — CRC16, Polynomial 2, Low Byte</td>
<td>433</td>
</tr>
<tr>
<td>VCRC32H_1 mem16 — CRC32, High Byte</td>
<td>434</td>
</tr>
<tr>
<td>VCRC32L_1 mem16 — CRC32, Low Byte</td>
<td>435</td>
</tr>
<tr>
<td>VCRCCLR — Clear CRC Result Register</td>
<td>436</td>
</tr>
<tr>
<td>VMOV32 mem32, VCRC — Store the CRC Result Register</td>
<td>437</td>
</tr>
<tr>
<td>VMOV32 VCRC, mem32 — Load the CRC Result Register</td>
<td>438</td>
</tr>
</tbody>
</table>
VCRC8H_1 mem16  —  CRC8, High Byte

Operands

| mem16     | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 1100
MSW: 0000 0000 mem16

Description

This instruction uses CRC8 polynomial == 0x07.

Calculate the CRC8 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

\[ \text{VCRC} = \text{CRC8} (\text{VCRC}, \text{mem16}[15:8]) \]

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VCRC8L_1 mem16

See also

VCRC8L_1 mem16
VCRC8L_1 mem16  

**CRC8, Low Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

- LSW: 1110 0010 1100 1011
- MSW: 0000 0000 mem16

**Description**

This instruction uses CRC8 polynomial == 0x07.

Calculate the CRC8 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

\[
\text{VCRC} = \text{CRC8} (\text{VCRC}, \text{mem16}[7:0])
\]

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData; // Start of data
    uint16_t CRCLen; // Length of data in bytes
} CRC_CALC;

CRC_CALC mycrc;
...
CRC8(&mycrc);
...
```

```assembly
; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words
.global _CRC8

_CRC8
  VCRCCLR ; Clear the result register
  MOV AL, *+XAR4[4] ; AL = CRCLen
  ASR AL, 2 ; AL = CRCLen/4
  SUBB AL, #1 ; AL = CRCLen/4 - 1
  MOVL XAR7, *+XAR4[2] ; XAR7 = 4CRCData
  .align 2
  NOP ; Align RPTB to an odd address
  RPTB _CRC8_done, AL ; Execute block of code AL + 1 times
  VCRC8L_1 *XAR7 ; Calculate CRC for 4 bytes
  VCRC8L_1 *XAR7++ ; ...
  VCRC8L_1 *XAR7 ; ...
  VCRC8H_1 *XAR7++ ; ...
  _CRC8_done
  MOVL XAR7, *+XAR4[0] ; XAR7 = 4CRCResult
  MOV32 ++XAR7[0], VCRC ; Store the result
  LRETR ; return to caller
```

**See also**

VCRC8H_1 mem16
**VCRC16P1H_1 mem16  **  
**CRC16, Polynomial 1, High Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

| LSW: 1110 0010 1100 1111 |
| MSW: 0000 0000  mem16 |

**Description**

This instruction uses CRC16 polynomial 1 == 0x8005.

Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

\[
\text{VCRC} = \text{CRC16} (\text{VCRC}, \text{mem16}[15:8])
\]

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VCRC16P1L_1 mem16.

**See also**

- VCRC16P1L_1 mem16
- VCRC16P2H_1 mem16
- VCRC16P2L_1 mem16
VCRC16P1L_1 mem16  CRC16, Polynomial 1, Low Byte

Operands

| mem16  | 16-bit memory location |

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 1100 1110</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 mem16</td>
</tr>
</tbody>
</table>

Description

This instruction uses CRC16 polynomial 1 == 0x8005.

Calculate the CRC16 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

\[ VCRC = CRC16 (VCRC, \text{mem16}[7:0]) \]

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```c
typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData;   // Start of data
    uint16_t CRCLen;     // Length of data in bytes
} CRC_CALC;

CRC_CALC mycrc;
...
CRC16P1(&mycrc);
...
```

```assembly
; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words
; .global _CRC16P1
 CRC16P1
  VCRCCLR ; Clear the result register
  MOV  AL, *+XAR4[4] ; AL = CRCLen
  ASR  AL, 2       ; AL = CRCLen/4
  SUBB AL, #1       ; AL = CRCLen/4 - 1
  MOVL XAR7, *+XAR4[2] ; XAR7 = 4CRCData
  .align 2
  NOP ; Align RPTB to an odd address
  RPTB _CRC16P1_done, AL ; Execute block of code AL + 1 times
  VCRC16P1L_1 *XAR7     ; Calculate CRC for 4 bytes
  VCRC16P1H_1 *XAR7++   ; ...
  VCRC16P1L_1 *XAR7     ; ...
  VCRC16P1H_1 *XAR7++   ; ...
  _CRC16P1_done
  MOVL XAR7, *+XAR4[0] ; XAR7 = 4CRCResult
  MOVSZ *+XAR7[0], VCRC ; Store the result
  LRET ; return to caller
```

See also

- VCRC16P1H_1 mem16
- VCRC16P2H_1 mem16
- VCRC16P2L_1 mem16
## VCRC16P2H_1 mem16  
**CRC16, Polynomial 2, High Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

LSW: 1110 0010 1110 1111  
MSW: 0001 0000 mem16

**Description**

This instruction uses CRC16 polynomial 2== 0x1021. Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

\[ VCRC = \text{CRC16}(VCRC, \text{mem16}[15:8]) \]

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VCRC16P2L_1 mem16.

**See also**

- VCRC16P2L_1 mem16
- VCRC16P1H_1 mem16
- VCRC16P1L_1 mem16
VCRC16P2L_1 mem16  CRC16, Polynomial 2, Low Byte

Operands

| mem16  | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 1110
MSW: 0001 0000 mem16

Description

This instruction uses CRC16 polynomial 2== 0x1021.
Calculate the CRC16 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

VCRC = CRC16 (VCRC, mem16[7:0])

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData; // Start of data
    uint16_t CRCLen; // Length of data in bytes
} CRC_CALC;

CRC_CALC mycrc;
...
CRC16P2(&mycrc);
...

; -------------------
; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words
; .global _CRC16P2
_CRC16P2
   _CRC16P2
   VCRCCLR ; Clear the result register
   MOV AL, *+XAR4[4] ; AL = CRCLen
   ASR AL, 2 ; AL = CRCLen/4
   SUBB AL, #1 ; AL = CRCLen/4 - 1
   MOVL XAR7, *+XAR4[2] ; XAR7 = 4CRCData
   .align 2
   NOP ; Align RPTB to an odd address
   RPTB _CRC16P2_done, AL ; Execute block of code AL + 1 times
   VCRC16P2L_1 *XAR7 ; Calculate CRC for 4 bytes
   VCRC16P2H_1 *XAR7++ ; ...
   VCRC16P2L_1 *XAR7 ; ...
   VCRC16P2H_1 *XAR7++ ; ...
   _CRC16P2_done
   MOVL XAR7, *+XAR4[0] ; XAR7 = 4CRCResult
   MOV32 ++XAR7[0], VCRC ; Store the result
   LRETR ; return to caller

See also

VCRC16P2H_1 mem16
VCRC16P1H_1 mem16
VCRC16P1L_1 mem16
VCRC32H_1 mem16  —  CRC32, High Byte

Operands

| mem16  | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 0010
MSW: 0000 0000 mem16

Description

This instruction uses CRC32 polynomial $1 = 0x04C11DB7$
Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it
with the value in the VCRC register. Store the result in VCRC.

$VCRC = CRC16(VCRC, \text{mem16}[15:8])$

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VCRC32L_1 mem16.

See also

VCRC32L_1 mem16
VCRC32L_1 mem16  CRC32, Low Byte

Operands

| mem16     | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 0001
MSW: 0000 0000 mem16

Description

This instruction uses CRC32 polynomial $1 = 0x04C11DB7$

Calculate the CRC32 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

$$VCRC = CRC32 (VCRC, \text{mem16}[7:0])$$

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```c
typedef struct {
  uint32_t  *CRCResult;  // Address where result should be stored
  uint16_t   *CRCData;   // Start of data
  uint16_t   CRCLen;     // Length of data in bytes
}CRC_CALC;

CRC_CALC mycrc;
...
CRC32(&mycrc);
...
```

`; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words `;
.global _CRC32

```
_CRC32
   VCRCCLR        ; Clear the result register
   MOV  AL,  *+XAR4[4]  ; AL = CRCLen
   ASR  AL,  2        ; AL = CRCLen/4
   SUBB AL, #1        ; AL = CRCLen/4 - 1
   MOVL XAR7,  *+XAR4[2] ; XAR7 = 4CRCData
   .align 2
   NOP              ; Align RPTB to an odd address
   RPTB _CRC16P2_done, AL ; Execute block of code AL + 1 times
   VCRC32_1  *XAR7    ; Calculate CRC for 4 bytes
   VCRC32_1  *XAR7++
   VCRC32_1  *XAR7
   VCRC32_1  *XAR7++
   _CRC32_done
   MOVL XAR7,  *+XAR4[0] ; XAR7 = 4CRCResult
   MOV32  ++XAR7[0], VCRC ; Store the result
   LRETR; return to caller
```

See also

VCRC32H_1 mem16
### VCRCLR — Clear CRC Result Register

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th>mem16</th>
<th>16-bit memory location</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0010 0100</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Clear the VCRC register. VCRC = 0x0000</td>
<td></td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>This instruction does not modify any flags in the VSTATUS register.</td>
<td></td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>This is a single-cycle instruction.</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>Refer to the example for VCRC32L_1 mem16.</td>
<td></td>
</tr>
</tbody>
</table>
| **See also** | VMOV32 mem32, VCRC  
VMOV32 VCRC, mem32 |
VMOV32 mem32, VCRC  Store the CRC Result Register

**Operands**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>32-bit memory destination</td>
</tr>
<tr>
<td>VCRC</td>
<td>CRC result register</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW: 1110 0010 0000 0110</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 mem32</td>
</tr>
</tbody>
</table>

**Description**

Store the VCRC register.

\[ \text{[mem32]} = \text{VCRC} \]

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

X

See also

VCRCCLR
VMOV32 VCRC, mem32
VMOV32 VCRC, mem32 — Load the CRC Result Register

Operands

<table>
<thead>
<tr>
<th>mem32</th>
<th>32-bit memory destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRC</td>
<td>CRC result register</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 0011 1111 0110
- MSW: 0000 0000 mem32

Description

Load the VCRC register.

\[ VCRC = [\text{mem32}] \]

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

\[ \text{VMOV32 mem32, VCRC} \]

See also

- VCRCLR
- VMOV32 mem32, VCRC
### 3.6.5 Viterbi Instructions

The instructions are listed alphabetically, preceded by a summary.

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITBM2 VR0</td>
<td>440</td>
</tr>
<tr>
<td>VITBM2 VR0</td>
<td></td>
</tr>
<tr>
<td>VITBM3 VR0, VR1, VR2</td>
<td>442</td>
</tr>
<tr>
<td>VITBM3 VR0, VR1, VR2</td>
<td></td>
</tr>
<tr>
<td>VITDHADDSSUB VR4, VR3, VR2, VRa</td>
<td>444</td>
</tr>
<tr>
<td>VITDHADDSSUB VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITDHSUBADD VR4, VR3, VR2, VRa</td>
<td>447</td>
</tr>
<tr>
<td>VITDHSUBADD VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITDLADDSSUB VR4, VR3, VR2, VRa</td>
<td>449</td>
</tr>
<tr>
<td>VITDLADDSSUB VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITDLSUBADD VR4, VR3, VR2, VRa</td>
<td>451</td>
</tr>
<tr>
<td>VITDLSUBADD VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITHSEL VRa, VRb, VR4, VR3</td>
<td>453</td>
</tr>
<tr>
<td>VITHSEL VRa, VRb, VR4, VR3</td>
<td></td>
</tr>
<tr>
<td>VITLSEL VRa, VRb, VR4, VR3</td>
<td>455</td>
</tr>
<tr>
<td>VITLSEL VRa, VRb, VR4, VR3</td>
<td></td>
</tr>
<tr>
<td>VTCLEAR</td>
<td>457</td>
</tr>
<tr>
<td>VTRACE mem32, VR0, VT0, VT1</td>
<td>458</td>
</tr>
<tr>
<td>VTRACE VR1, VR0, VT0, VT1</td>
<td>460</td>
</tr>
</tbody>
</table>
VITBM2 VR0 — Code Rate 1:2 Branch Metric Calculation

Operands

Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit decoder input 1</td>
</tr>
</tbody>
</table>

The result of the operation is also stored in VR0 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR0H</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L - VR0L</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0000 1100

Description

Branch metric calculation for code rate = 1/2.

```c
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR0H is decoder input 1

// Calculate the branch metrics by performing 16-bit signed addition and subtraction
//
// VR0L = VR0L + VR0H; // VR0L = branch metric 0
// VR0H = VR0L - VR0L; // VR0H = branch metric 1
//
// if (SAT == 1)
// {
//    sat16(VR0L);        // VR0L = branch metric 0
//    sat16(VR0H);        // VR0H = branch metric 1
// }
```

Flags

This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

Pipeline

This is a single-cycle instruction.

Example

See also

VITBM2 VR0 || VMOV32 VR2, mem32
VITBM3 VR0, VR1, VR2
VITBM2 VR0 || VMOV32 VR2, mem32  

**Code Rate 1:2 Branch Metric Calculation with Parallel Load**

**Operands**

Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit decoder input 1</td>
</tr>
<tr>
<td>[mem32]</td>
<td>pointer to 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR0 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR0H</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L - VR0L</td>
</tr>
<tr>
<td>VR2</td>
<td>contents of memory pointed to by [mem32]</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0011 1111 1100  
MSW: 0000 aaaa mem32

**Description**

Branch metric calculation for a code rate of 1/2 with parallel register load.

```c
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR0H is decoder input 1
//
// Calculate the branch metrics by performing 16-bit signed addition and subtraction
//
// VR0L = VR0L + VR0H;  // VR0L = branch metric 0
// VR0H = VR0L - VR0L;  // VR0H = branch metric 1
//
if (SAT == 1)  
{
    sat16(VR0L);
    sat16(VR0H);
}
VR2 = [mem32]  // Load VR2L and VR2H with the next state metrics
```

**Flags**

This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

**Pipeline**

Both operations complete in a single cycle.

**Example**

See also

VITBM2 VR0
VITBM3 VR0, VR1, VR2
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32
VITBM3 VR0, VR1, VR2 — Code Rate 1:3 Branch Metric Calculation

Operands
Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit decoder input 1</td>
</tr>
<tr>
<td>VR2L</td>
<td>16-bit decoder input 2</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR0 and VR1 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR1L + VR2L</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L + VR1L - VR2L</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit branch metric 2 = VR0L - VR1L + VR2L</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit branch metric 3 = VR0L - VR1L - VR2L</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0000 1101

Description
Calculate the four branch metrics for a code rate of 1/3.

```c
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR1L is decoder input 1
// VR2L is decoder input 2

// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction

VR0L = VR0L + VR1L + VR2L; // VR0L = branch Metric 0
VR0H = VR0L + VR1L - VR2L; // VR0H = branch Metric 1
VR1L = VR0L - VR1L + VR2L; // VR1L = branch Metric 2
VR1H = VR0L - VR1L - VR2L; // VR1H = branch Metric 3
if(SAT == 1)
{
    sat16(VR0L);
    sat16(VR0H);
    sat16(VR1L);
    sat16(VR1H);
}
```

Flags
This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

Pipeline
This is a 2p-cycle instruction. The instruction following VITBM3 must not use VR0 or VR1.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also
VITBM2 VR0
VITBM2 VR0 || VMOV32 VR2, mem32
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32  

Code Rate 1:3 Branch Metric Calculation with Parallel Load

### Operands

Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit decoder input 1</td>
</tr>
<tr>
<td>[mem32]</td>
<td>pointer to a 32-bit memory location</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR0 and VR1 and VR2 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR1L + VR2L</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L + VR1L - VR2L</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit branch metric 2 = VR0L - VR1L + VR2</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit branch metric 3 = VR0L - VR1L - VR2L</td>
</tr>
<tr>
<td>VR2</td>
<td>Contents of the memory pointed to by [mem32]</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW:</th>
<th>MSW:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0011 1111</td>
<td>1101 0000 aaaa mem32</td>
</tr>
</tbody>
</table>

### Description

Calculate the four branch metrics for a code rate of 1/3 with parallel register load.

```c
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR1L is decoder input 1
// VR2L is decoder input 2
//
// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction
//
// VR0L = VR0L + VR1L + VR2L; // VR0L = branch Metric 0
// VR0H = VR0L + VR1L - VR2L; // VR0H = branch Metric 1
// VR1L = VR0L - VR1L + VR2;  // VR1L = branch Metric 2
// VR1H = VR0L - VR1L - VR2L; // VR1H = branch Metric 3
if(SAT == 1)
{
    sat16(VR0L);
    sat16(VR0H);
    sat16(VR1L);
    sat16(VR1H);
}
VR2 = [mem32];
```

### Flags

This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

### Pipeline

This is a 2p/1-cycle instruction. The VBITM3 operation takes 2p cycles and the VMOV32 completes in a single cycle. The next instruction must not use VR0 or VR1.

### Example

Refer to the example for VITDADDSSUB VR4, VR3, VR2, VRa.

### See also

- VITBM2 VR0
- VITBM2 VR0 || VMOV32 VR2, mem32
VITDHADDSUB VR4, VR3, VR2, VRa — Viterbi Double Add and Subtract, High

Operands
Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaH.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaH</td>
<td>Branch metric 1. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaH</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0111 aaaa

Description
Viterbi high add and subtract. This instruction is used to calculate four path metrics.

```
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaH with the branch metric.
//
// VR3L = VR2L + VRaH  // Path metric 0
// VR3H = VR2H - VRaH  // Path metric 1
// VR4L = VR2L - VRaH  // Path metric 2
// VR4H = VR2H + VRaH  // Path metric 3
```

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
```
; Example Viterbi decoder code fragment
; Viterbi butterfly calculations
; Loop once for each decoder input pair
;
; Branch metrics = BM0 and BM1
; XAR5 points to the input stream to the decoder
...
...
_loop:
  VMOV32 VR0, *XAR5++         ; Load two inputs into VR0L, VR0H
  VITBM2 VR0                 ; VR0L = BM0   VR0H = BM1
  || VMOV32 VR2, *XAR1++      ; Load previous state metrics

;
; 2 cycle Viterbi butterfly
;
  VITDLADDSUB VR4,VR3,VR2,VR0; Perform add/sub
  VITLSEL VR6,VR5,VR4,VR3    ; Perform compare/select
  || VMOV32 VR2, *XAR1++      ; Load previous state metrics

;
; 2 cycle Viterbi butterfly, next stage
;
  VITDHADDSUB VR4,VR3,VR2,VR0
```
VITHSEL VR6, VR5, VR4, VR3
|| VMOV32 VR2, *XAR1++

; 2 cycle Viterbi butterfly, next stage
;
VITDLADDSUB VR4, VR3, VR2, VR0
|| VMOV32 *XAR2++, VR5
...
VITDHADDSUB VR4, VR3, VR2, VRa || mem32 VRb — Viterbi Add and Subtract High with Parallel Store

Operands
Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaH.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaH</td>
<td>Branch metric 1. VRa must be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Value to be stored. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaH</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0000 1001
MSW: bbbb aaaa mem32

Description
Viterbi high add and subtract. This instruction is used to calculate four path metrics.

// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaH with the branch metric.
//
// VR3L = VR2L + VRaH    // Path metric 0
// VR3H = VR2H - VRaH    // Path metric 1
// VR4L = VR2L - VRaH    // Path metric 2
// VR4H = VR2H + VRaH    // Path metric 3

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example

See also
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa — Viterbi Add and Subtract Low

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H + VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H - VRaL</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1111  aaaa

Description

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

```c
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaL with the branch metric.
//
// VR3L = VR2L - VRaL  // Path metric 0
// VR3H = VR2H + VRaL  // Path metric 1
// VR4L = VR2L + VRaL  // Path metric 2
// VR4H = VR2H - VRaL  // Path metric 3
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHDADDUB VR4, VR3, VR2, VRa.

See also

VITDHDADDUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa || mem32 VRb — Viterbi Subtract and Add, High with Parallel Store

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaH.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaH</td>
<td>Branch metric 1. VRa must be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Contents to be stored. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H + VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H - VRaH</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 0101
MSW: bbbb aaaa mem32

Description

Viterbi high subtract and add. This instruction is used to calculate four path metrics.

```
// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaH with the branch metric.
// [mem32] = VRb // Store VRb to memory
VR3L = VR2L - VRaH  // Path metric 0
VR3H = VR2H + VRaH  // Path metric 1
VR4L = VR2L + VRaH  // Path metric 2
VR4H = VR2H - VRaH  // Path metric 3
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VITDHADDSUB VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa  

Viterbi Add and Subtract Low

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaL</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0011 aaaa

Description

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaL with the branch metric.
//
// VR3L = VR2L + VRaL // Path metric 0
// VR3H = VR2H - VRaL // Path metric 1
// VR4L = VR2L - VRaL // Path metric 2
// VR4H = VR2H + VRaL // Path metric 3

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa || mem32 VRb — Viterbi Add and Subtract Low with Parallel Load

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa can be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Contents to be stored to memory</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaL</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 1000
MSW: bbbb aaaa mem32

Description

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

```c
// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaL with the branch metric.
// [mem32] = VRb          // Store VRb
VR3L = VR2L + VRaL      // Path metric 0
VR3H = VR2H - VRaL      // Path metric 1
VR4L = VR2L - VRaL      // Path metric 2
VR4H = VR2H + VRaL      // Path metric 3
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa — Viterbi Subtract and Add Low

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H + VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H - VRaL</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1110  aaaa

Description

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaH with the branch metric.
//
// VR3L = VR2L - VRaL   // Path metric 0
// VR3H = VR2H + VRaL   // Path metric 1
// VR4L = VR2L + VRaL   // Path metric 2
// VR4H = VR2H - VRaL   // Path metric 3

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADD SUB VR4, VR3, VR2, VRa.

See also

VITDHADD SUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
**VITDLSUBADD VR4, VR3, VR2, VRa || mem32 VRb — Viterbi Subtract and Add, Low with Parallel Store**

**Operands**

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Value to be stored. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaL</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H + VRaL</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H - VRaL</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW: 1110 0010 0000</th>
<th>1010</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: bbbb aaaa</td>
<td>mem32</td>
</tr>
</tbody>
</table>

**Description**

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

```c
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaH with the branch metric.
//
// [mem32] = VRb // Store VRb into mem32
VR3L = VR2L - VRaL // Path metric 0
VR3H = VR2H + VRaL // Path metric 1
VR4L = VR2L + VRaL // Path metric 2
VR4H = VR2H - VRaL // Path metric 3
```

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

**See also**

- VITDHADDSUB VR4, VR3, VR2, VRa
- VITDHSUBADD VR4, VR3, VR2, VRa
- VITDLADDSUB VR4, VR3, VR2, VRa
VITHSEL VRa, VRb, VR4, VR3  

Viterbi Select High

Operands

Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaH</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbH</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1111 0111  
MSW: 0000 0000 bbbb aaaa

Description

This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16-bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITLSEL instruction.

```
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbH = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
} else
{
    VRbH = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}
T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaH = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
} else
{
    VRaH = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITLSEL VRa, VRb, VR4, VR3
VITHSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 — Viterbi Select High with Parallel Load

Operands

Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
<tr>
<td>[mem32]</td>
<td>pointer to 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaH</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbH</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VR2</td>
<td>Contents of the memory pointed to by [mem32].</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 1111
MSW: bbbb aaaa mem32

Description

This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16-bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITLSEL instruction.

```
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbH = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
}
else
{
    VRbH = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}
T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaH = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
}
else
{
    VRaH = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
VR2 = [mem32]; // Load VR2
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITLSEL VRa, VRb, VR4, VR3
VITLSEL VRa, VRb, VR4, VR3  Viterbi Select, Low Word

Operands
Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaL</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbL</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0110 1111 0110
MSW: 0000 0000 bbbb aaaa

Description
This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16-bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITHSEL instruction.

```
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbL = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
}
else
{
    VRbL = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}

T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaL = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
}
else
{
    VRaL = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
```

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also
VITHSEL VRa, VRb, VR4, VR3
VITSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 — Viterbi Select Low with Parallel Load

Operands

Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaL</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbL</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VR2</td>
<td>Contents of 32-bit memory pointed to by mem32.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 1111
MSW: bbbb aaaa mem32

Description

This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16-bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITHSEL instruction. In parallel the VR2 register is loaded with the contents of memory pointed to by [mem32].

```plaintext
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbL = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
}
else
{
    VRbL = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}
T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaL = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
}
else
{
    VRaL = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
VR2 = [mem32]
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITHSEL VRa, VRb, VR4, VR3
VTCLEAR — Clear Transition Bit Registers

Operands
none

Opcode
LSW: 1110 0101 0010 1001

Description
Clear the VT0 and VT1 registers.
VT0 = 0;
VT1 = 0;

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example

See also
VCLEARALL
VCLEAR VRa
VTRACE mem32, VR0, VT0, VT1  —  Viterbi Traceback, Store to Memory

Operands

Before the operation, the path metrics are loaded into the registers as shown below using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT0</td>
<td>transition bit register 0</td>
</tr>
<tr>
<td>VT1</td>
<td>transition bit register 1</td>
</tr>
<tr>
<td>VR0</td>
<td>Initial value is zero. After the first VTRACE, this contains information from the previous trace-back.</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[mem32]</td>
<td>Traceback result from the transition bits.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 1100
MSW: 0000 0000 mem32

Description

Trace-back from the transition bits stored in VT0 and VT1 registers. Write the result to memory. The transition bits in the VT0 and VT1 registers are stored in the following format by the VITLSEL and VITHSEL instructions:

| VT0[31]         | Transition bit [State 0] |
| VT0[30]         | Transition bit [State 1] |
| VT0[29]         | Transition bit [State 2] |
| ...             | ...                      |
| VT0[0]          | Transition bit [State 31]|
| VT1[31]         | Transition bit [State 32]|
| VT1[30]         | Transition bit [State 33]|
| VT1[29]         | Transition bit [State 34]|
| ...             | ...                      |
| VT1[0]          | Transition bit [State 63]|

// Calculate the decoder output bit by performing a traceback from the transition bits stored in the VT0 and VT1 registers
S = VR0[5:0];
VR0[31:6] = 0;
if (S < 32)
{
    temp[0] = VT0[31-S];
}
else
{
    temp[0] = VT1[63-S];
}
*[mem32][0] = temp;
*[mem32][31:1] = 0;
VR0[5:0] = 2*VR0[5:0] + temp[0];

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

// Example traceback code fragment
//
// XAR5 points to the beginning of Decoder Output array
//
VCLEAR VR0
MOVL XAR5, *+XAR4[0]

//
// To retrieve each original message:
// Load VT0/VT1 with the stored transition values
// and use VTRACE instruction
//
VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE *XAR5++, VR0, VT0, VT1

VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE *XAR5++, VR0, VT0, VT1
...
...etc for each VT0/VT1 pair

See also

VTRACE VR1, VR0, VT0, VT1
VTRACE VR1, VR0, VT0, VT1 — *Viterbi Traceback, Store to Register*

**Operands**
Before the operation, the path metrics are loaded into the registers as shown below using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT0</td>
<td>transition bit register 0</td>
</tr>
<tr>
<td>VT1</td>
<td>transition bit register 1</td>
</tr>
<tr>
<td>VR0</td>
<td>Initial value is zero. After the first VTRACE, this contains information from the previous trace-back.</td>
</tr>
</tbody>
</table>

The result of the operation is the output of the decoder stored in VR1:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR1</td>
<td>Traceback result from the transition bits.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0101 0010 1000

**Description**
Trace-back from the transition bits stored in VT0 and VT1 registers. Write the result to VR1. The transition bits in the VT0 and VT1 registers are stored in the following format by the VITLSEL and VITHSEL instructions:

<table>
<thead>
<tr>
<th>VT0[i]</th>
<th>Transition bit [State i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT0[31]</td>
<td>Transition bit [State 0]</td>
</tr>
<tr>
<td>VT0[30]</td>
<td>Transition bit [State 1]</td>
</tr>
<tr>
<td>VT0[29]</td>
<td>Transition bit [State 2]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>VT0[0]</td>
<td>Transition bit [State 31]</td>
</tr>
<tr>
<td>VT1[i]</td>
<td>Transition bit [State i+32]</td>
</tr>
<tr>
<td>VT1[31]</td>
<td>Transition bit [State 32]</td>
</tr>
<tr>
<td>VT1[30]</td>
<td>Transition bit [State 33]</td>
</tr>
<tr>
<td>VT1[29]</td>
<td>Transition bit [State 34]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>VT1[0]</td>
<td>Transition bit [State 63]</td>
</tr>
</tbody>
</table>

//
// Calculate the decoder output bit by performing a
// traceback from the transition bits stored in the VT0 and VT1 registers
//
S = VR0[5:0];
VR0[31:6] = 0;
if (S < 32)
{
    temp[0] = VT0[31-S];
}
else
{
    temp[0] = VT1[63-S];
}
VR1[0] = temp;
VR1[31:1] = 0;
VR0[5:0] = 2*VR0[5:0] + temp[0];

**Flags**
This instruction does not modify any flags in the VSTATUS register.

**Pipeline**
This is a single-cycle instruction.

**Example**

See also
VTRACE mem32, VR0, VT0, VT1
3.7 Rounding Mode

This section details the rounding operation as applied to a right shift. When the rounding mode is enabled in the VSTATUS register, .5 will be added to the right shifted intermediate value before truncation. If rounding is disabled the right shifted value is only truncated. **Table 3-14** shows the bit representation of two values, 11.0 and 13.0. The columns marked Bit-1, Bit-2 and Bit-3 hold temporary bits resulting from the right shift operation.

### Table 3-14. Example: Values Before Shift Right

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit-3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>13.000</td>
</tr>
</tbody>
</table>

**Table 3-14** Shows the intermediate values after the right shift has been applied to Val B. The columns marked Bit-1, Bit-2 and Bit-3 hold temporary bits resulting from the right shift operation.

### Table 3-15. Example: Values after Shift Right

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit-3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.625</td>
</tr>
</tbody>
</table>

When the rounding mode is enabled, .5 will be added to the intermediate result before truncation. **Table 3-16** shows the bit representation of Val A + Val (B >> 3) operation with rounding. Notice .5 is added to the intermediate shifted right value. After the addition, the bits in Bit-1, Bit-2 and Bit-3 are removed. In this case the result of the operation will be 13 which is the truncated value after rounding.

### Table 3-16. Example: Addition with Right Shift and Rounding

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit-3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.625</td>
</tr>
<tr>
<td>.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.500</td>
</tr>
<tr>
<td>Val A + Val B &gt;&gt; 3 + .5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>13.125</td>
</tr>
</tbody>
</table>

When the rounding mode is disabled, the value is simply truncated. **Table 3-17** shows the bit representation of the operation Val A + (Val B >> 3) without rounding. After the addition, the bits in Bit-1, Bit-2 and Bit-3 are removed. In this case the result of the operation will be 12 which is the truncated value without rounding.

### Table 3-17. Example: Addition with Rounding After Shift Right

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit-3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.625</td>
</tr>
<tr>
<td>Val A + Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>12.625</td>
</tr>
</tbody>
</table>

**Table 3-18** shows more examples of the intermediate shifted value along with the result if rounding is enabled or disabled. In each case, the truncated value is without .5 added and the rounded value is with .5 added.

### Table 3-18. Shift Right Operation With and Without Rounding

<table>
<thead>
<tr>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit -1</th>
<th>Bit -2</th>
<th>Value</th>
<th>Result with RND = 0</th>
<th>Result with RND = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.00</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.75</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1.50</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 3-18. Shift Right Operation With and Without Rounding (continued)

<table>
<thead>
<tr>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Value</th>
<th>Result with RND = 0</th>
<th>Result with RND = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.25</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.75</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.50</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-0.50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-0.75</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1.00</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-1.25</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1.50</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1.75</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-2.00</td>
<td>-2</td>
<td>-2</td>
</tr>
</tbody>
</table>
This chapter provides an overview of the architectural structure and instruction set of the CRC Unit (VCRC) and describes the architecture, pipeline, instruction set, and interrupts. The VCRC is a fully-programmable block.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Overview</td>
<td>464</td>
</tr>
<tr>
<td>4.2 VCRC Code Development</td>
<td>464</td>
</tr>
<tr>
<td>4.3 Components of the C28x Plus VCRC</td>
<td>464</td>
</tr>
<tr>
<td>4.4 Register Set</td>
<td>467</td>
</tr>
<tr>
<td>4.5 Pipeline</td>
<td>469</td>
</tr>
<tr>
<td>4.6 Instruction Set</td>
<td>470</td>
</tr>
</tbody>
</table>
4.1 Overview

The C28x with VCRC (C28x+VCRC) processor extends the capabilities of the C28x CPU by adding registers and instructions to support CRC. CRC algorithms provide a straightforward method for verifying data integrity over large data blocks, communication packets, or code sections. The C28x+VCRC can perform 8-, 16-, 24-, and 32-bit CRCs.

4.2 VCRC Code Development

When developing C28x VCRC code for C28x+VCRC, use Code Composer Studio 8.0, or later. The TI C28x C/C++ Compiler v18.9.0.STS or later is required, use the compiler switches: -v28 and --vcu_support=vcrc. The support for intrinsic for VCRC will be provided in the compiler 19.6.0.STS release.

4.3 Components of the C28x Plus VCRC

The VCRC extends the capabilities of the C28x CPU by adding additional instructions. No changes have been made to existing instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x are completely compatible with the C28x+VCRC. All of the features of the C28x documented in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430) apply to the C28x+VCRC. Figure 4-1 shows the block diagram of the C28x+VCRC.

Figure 4-1. C28x + VCRC Block Diagram
The C28x+VCRC contains the same features as the C28x fixed-point CPU:

- A central processing unit for generating data and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory.
- Emulation logic for monitoring and controlling various parts and functions of the device and for testing device operation. This logic is identical to that on the C28x fixed-point CPU.
- Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts. This logic is identical to the C28x fixed-point CPU.
- Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- Address register arithmetic unit (ARAU). The ARAU generates data memory addresses and increments or decrements pointers in parallel with ALU operations.
- Fixed-Point instructions are pipeline protected. This pipeline for fixed-point instructions is identical to that on the C28x fixed-point CPU. The CPU implements an 8-phase pipeline that prevents a write to and a read from the same location from occurring out of order.
- Barrel shifter. This shifter performs all left and right shifts of fixed-point data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.
- Fixed-Point Multiplier. The multiplier performs 32-bit × 32-bit 2s-complement multiplication with a 64-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.

The VCRC adds the following features:

- Instructions to support Cyclic Redundancy Check (CRC) or a polynomial code checksum are categorized into 2 categories.
  - Fixed polynomial fixed data size (8 bits) instructions that execute in one pipeline cycle (CRC8, CRC16, CRC32, CRC24)
  - Configurable polynomial configurable data size instructions that execute in three pipeline cycles.
- Clocked at the same rate as the main CPU (SYSCLKOUT).
- VCRC instructions can perform CRC calculation on the data stored in C28x ROM, RAMs and Flash to check their integrity during application runtime. CRC can be computed by C28x application code by using the CRC related VCRC instructions described in this section.
- Some VCRC instructions require pipeline alignment. This alignment is done through software to allow the user to improve performance by taking advantage of required delay slots. See Section 4.5 for more information.

### 4.3.1 Emulation Logic

The emulation logic is identical to that on the C28x fixed-point CPU. This logic includes the following features. For more details about these features, refer to the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430):

- Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline.
- A counter for performance benchmarking.
- Multiple debug events. Any of the following debug events can cause a break in program execution:
  - A breakpoint initiated by the ESTOP0 or ESTOP1 instruction.
  - An access to a specified program-space or data-space location. When a debug event causes the C28x to enter the debug-halt state, the event is called a break event.
- Real-time mode of operation.
4.3.2 Memory Map

Like the C28x, the C28x+VCRC uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x+VCRC designs are uniformly mapped to both program and data space. For specific details about each of the map segments, see the device-specific data manual.

4.3.3 CPU Interrupt Vectors

The C28x+VCRC interrupt vectors are identical to those on the C28x CPU. Sixty-four addresses in program space are set aside for a table of 32 CPU interrupt vectors. For more information about the CPU vectors, see TMS320C28x CPU and Instruction Set Reference Guide (literature number SPRU430). Typically the CPU interrupt vectors are only used during the boot up of the device by the boot ROM. Once an application has taken control it should initialize and enable the peripheral interrupt expansion block (PIE).

4.3.4 Memory Interface

The C28x+VCRC memory interface is identical to that on the C28x. The C28x+VCRC memory map is accessible outside the CPU by the memory interface, which connects the CPU logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed. The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the CPU supports special byte-access instructions that can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.

4.3.5 Address and Data Buses

Like the C28x, the memory interface has three address buses:

- PAB: Program address bus: The 22-bit PAB carries addresses for reads and writes from program space.
- DRAB: Data-read address bus: The 32-bit DRAB carries addresses for reads from data space.
- DWAB: Data-write address bus: The 32-bit DWAB carries addresses for writes to data space.

The memory interface also has three data buses:

- PRDB: Program-read data bus: The 32-bit PRDB carries instructions during reads from program space.
- DRDB: Data-read data bus: The 32-bit DRDB carries data during reads from data space.
- DWDB: Data-/Program-write data bus: The 32-bit DWDB carries data during writes to data space or program space.

A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time. This behavior is identical to the C28x CPU.
4.3.6 Alignment of 32-Bit Accesses to Even Addresses

The C28x+VCRC expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU will begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the CPU.

You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space.

4.4 Register Set

Devices with the C28x+VCRC include the standard C28x register set plus an additional set of VCRC specific registers. Figure 4-2 shows a diagram of both register sets and Section 4.4.1 shows a register summary.

![Figure 4-2. C28x + VCRC Registers](image)

- ACC (32-bit)
- P (32-bit)
- XT (32-bit)
- VSTATUS (32-bit)
- VCRCPOLY (32-bit)
- VCRCSIZE (32-bit)
- VCR (32-bit)
- VCUREV (32-bit)
- VSTATUS (32-bit)
- VCRCPOLY (32-bit)
- VCRCSIZE (32-bit)
- VCR (32-bit)
- VCUREV (32-bit)
- XAR0 (32-bit)
- XAR1 (32-bit)
- XAR2 (32-bit)
- XAR3 (32-bit)
- XAR4 (32-bit)
- XAR5 (32-bit)
- XAR6 (32-bit)
- XAR7 (32-bit)
- PC (22-bit)
- RPC (22-bit)
- DP (16-bit)
- SP (16-bit)
- ST0 (16-bit)
- ST1 (16-bit)
- IER (16-bit)
- IFR (16-bit)
- DBGIER (16-bit)
### 4.4.1 VCRC Register Set

#### Table 4-1. VCRC Status (VSTATUS) Register Field Descriptions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>CRCMSGF</td>
<td>LIP</td>
<td>CRC Message Flip&lt;br&gt;This bit affects the order in which the bits in the message are taken for CRC calculation by all the CRC instructions.</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>Message bits are taken starting from most-significant to least-significant for CRC computation. In this case, bytes loaded from memory are fed directly for CRC computation.</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>Message bits are taken starting from least-significant to most-significant for CRC computation. In this case, bytes loaded from memory are “flipped” and then fed for CRC computation.</td>
</tr>
<tr>
<td>30-0</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

#### Table 4-2. VCRC: The CRC result register for unsecured memories

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RESULT</td>
<td></td>
<td>The CRC result gets updated in this register. When using a polynomial value less than 32 bits wide, the VCRC.RESULT will be right justified with the upper bits reading as zero. This register can be cleared by executing the VCRCCLR instruction.</td>
</tr>
</tbody>
</table>

#### Table 4-3. VCRCPOLY: The CRC Polynomial register for generic CRC instructions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>POLY</td>
<td></td>
<td>This register defines the polynomial value used by the generic CRC VCRCL/VCRCH instructions. This register is right justified.</td>
</tr>
</tbody>
</table>

#### Table 4-4. VCRCSIZE: The CRC Polynomial and Data Size register for generic CRC instructions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>DSIZE</td>
<td></td>
<td>This bit field defines the size of the data value used by the generic CRC VCRCL/VCRCH instructions. The VCRCL/H instructions always expect the data to be right justified and ignore the upper bits. 0x0: Data size is 1 bit 0x1: Data size is 2 bits 0x2: Data size is 3 bits … 0x7: Data size is 8 bits</td>
</tr>
<tr>
<td>15:3</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>20:16</td>
<td>PSIZE</td>
<td></td>
<td>This bit field defines the size of the polynomial value used by the generic CRC VCRCL/VCRCH instructions. 0x00: Polynomial size is 1 bit 0x01: Polynomial size is 2 bits 0x02: Polynomial size is 3 bits … 0x1F: Polynomial size is 32 bits</td>
</tr>
<tr>
<td>31:21</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 4-5. VCUREV: VCU revision register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>VCUREV</td>
<td></td>
<td>0: Indicates VCU-I 1: Indicates VCU-II 2: Indicates VCU2.1 3: Indicates VCRC</td>
</tr>
</tbody>
</table>
4.5 Pipeline

This section describes the VCRC pipeline stages and presents cases where pipeline alignment must be considered.

4.5.1 Pipeline Overview

The C28x VCRC pipeline is identical to the C28x pipeline for all standard C28x instructions. In the decode2 stage (D2), it is determined if an instruction is a C28x instruction or a VCRC instruction. C28x VCRC instructions are single cycle or three cycle instructions. The rest of this section will describe when delay cycles are required. Keep in mind that the assembly tools for the C28x+VCRC will issue an error if a delay slot has not been handled correctly.

4.5.2 General Guidelines for VCRC Pipeline Alignment

The majority of the VCRC instructions do not require any special pipeline considerations. This section lists the few operations that do require special consideration. While the C28x+VCRC assembler will issue errors for pipeline conflicts, you may still find it useful to understand when software delays are required.

C28x fixed-point instructions can be used in VCRC instruction delay slots as long as source and destination register conflicts are avoided. The C28x+VCRC assembler will issue an error anytime you use a conflicting instruction within a delay slot.

Following are the careabouts related to multi-cycle pipelined instructions:

1. All fixed polynomial VCRC instructions are executed in single cycle. However if fixed polynomial VCRC instructions is followed by an instruction which updates VCRC register then a NOP is necessary before update of VCRC register.

   For example - write of VCRC after CRC calculation (Illegal scenario):
   
   VCRC16P1L_1 *XAR7++
   VMOV32 VCRC, *XAR6++

   To make the above legal, insert a NOP:
   
   VCRC16P1L_1 *XAR7++
   NOP
   VMOV32 VCRC, *XAR6++

   For example - read of VCRC after CRC calculation (Legal scenario):
   
   VCRC16P1L_1 *XAR7++
   VMOV32 *XAR6++, VCRC

2. Configurable polynomial instructions are executed in 3 cycles and hence appropriate NOPs must be inserted after VCRC instructions for proper execution.

   For example - Storing VCRC register to memory (Illegal scenario):
   
   VCRCL *XAR7++
   VMOV32 *XAR6++, VCRC

   To make the above legal, insert two NOPs
   
   VCRCL *XAR7++
   NOP
   NOP
   VMOV32 *XAR6++, VCRC
4.6 Instruction Set

This section describes the assembly language instructions of the VCRC. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are independent from C28x and C28x+FPU instruction sets.

4.6.1 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Operands
- Opcode
- Description
- Exceptions
- Pipeline
- Examples
- See also
The example INSTRUCTION is shown to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information. VCRC instructions follow the same format as the C28x; the source operand(s) are always on the right and the destination operand(s) are on the left.

The explanations for the syntax of the operands used in the instruction descriptions for the C28x VCRC are given in Table 4-6.

### Table 4-6. Operand Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>16-bit immediate (hex or float) value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FHiHex</td>
<td>16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32Fhex</td>
<td>32-bit immediate value that represents an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32Ffloat</td>
<td>Immediate float value represented in floating-point representation</td>
</tr>
<tr>
<td>#0.0</td>
<td>Immediate zero</td>
</tr>
<tr>
<td>#5-bit</td>
<td>5-bit immediate unsigned value</td>
</tr>
<tr>
<td>addr</td>
<td>Opcode field indicating the addressing mode</td>
</tr>
<tr>
<td>Im(X), Im(Y)</td>
<td>Imaginary part of the input X or input Y</td>
</tr>
<tr>
<td>Im(Z)</td>
<td>Imaginary part of the output Z</td>
</tr>
<tr>
<td>Re(X), Re(Y)</td>
<td>Real part of the input X or input Y</td>
</tr>
<tr>
<td>Re(Z)</td>
<td>Real part of the output Z</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
<tr>
<td>VRa</td>
<td>VR0 - VR8 registers. Some instructions exclude VR8. Refer to the instruction description for details.</td>
</tr>
<tr>
<td>VR0H, VR1H...VR7H</td>
<td>VR0 - VR7 registers, high half.</td>
</tr>
<tr>
<td>VROL, VR1L,...VR7L</td>
<td>VR0 - VR7 registers, low half.</td>
</tr>
<tr>
<td>VT0, VT1</td>
<td>Transition bit register VT0 or VT1.</td>
</tr>
<tr>
<td>VSMn+1: VSMn</td>
<td>Pair of State Metric Registers (n = 0 : 62, n is even)</td>
</tr>
<tr>
<td>VRx.By</td>
<td>32 bit Aliased address space for each byte of the VRx registers (x=0:7, y =0:3)</td>
</tr>
</tbody>
</table>

Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).

### Table 4-7. INSTRUCTION dest, source1, source2 Short Description

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dest1</td>
<td>Description for the 1st operand for the instruction</td>
</tr>
<tr>
<td>source1</td>
<td>Description for the 2nd operand for the instruction</td>
</tr>
<tr>
<td>source2</td>
<td>Description for the 3rd operand for the instruction</td>
</tr>
<tr>
<td>Opcode</td>
<td>This section shows the opcode for the instruction</td>
</tr>
<tr>
<td>Description</td>
<td>Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Any constraints on the operands or use of the instruction imposed by the processor are discussed.</td>
</tr>
<tr>
<td>Pipeline</td>
<td>This section describes the instruction in terms of pipeline cycles as described in Section 4.5.</td>
</tr>
<tr>
<td>Example</td>
<td>Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution. Some examples are code fragments while other examples are full tasks that assume the VCU is correctly configured and the main CPU has passed it data.</td>
</tr>
<tr>
<td>Operands</td>
<td>Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).</td>
</tr>
</tbody>
</table>
4.6.2 General Instructions

The instructions are listed alphabetically, preceded by a summary.

### Table 4-8. General Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV32 VCRC, mem32 —32bit write of CRC result register (VCRC)</td>
<td>473</td>
</tr>
<tr>
<td>VMOV32 mem32, VCRC —32bit read of CRC result register (VCRC)</td>
<td>474</td>
</tr>
<tr>
<td>VNOP —No operation</td>
<td>475</td>
</tr>
<tr>
<td>VMOV32 VSTATUS, mem32 —32bit load of VSTATUS register from memory</td>
<td>476</td>
</tr>
<tr>
<td>VMOV32 mem32, VSTATUS —32bit store of VSTATUS register to memory</td>
<td>477</td>
</tr>
<tr>
<td>VSETCRCMSGFLIP —Set CRCMSGFLIP bit in the VSTATUS Register</td>
<td>478</td>
</tr>
<tr>
<td>VCLRRCMSGFLIP —Clear CRCMSGFLIP bit in the VSTATUS</td>
<td>479</td>
</tr>
<tr>
<td>VRCRC8L_1 mem16 — CRC8, Lowbyte</td>
<td>480</td>
</tr>
<tr>
<td>VRCRC8H_1 mem16 —CRC8, High Byte</td>
<td>481</td>
</tr>
<tr>
<td>VRCRC16P1L_1 mem16 —CRC16, Polynomial 1, Low Byte</td>
<td>482</td>
</tr>
<tr>
<td>VRCRC16P1H_1 mem16 —CRC16, Polynomial 1, High Byte</td>
<td>483</td>
</tr>
<tr>
<td>VRCRC16P2L_1 mem16 —CRC16, Polynomial 2, Low Byte</td>
<td>484</td>
</tr>
<tr>
<td>VRCRC16P2H_1 mem16 —CRC16, Polynomial 2, High Byte</td>
<td>485</td>
</tr>
<tr>
<td>VRCRC32L_1 mem16 —CRC32, Polynomial 1, Low Byte</td>
<td>486</td>
</tr>
<tr>
<td>VRCRC32H_1 mem16 —CRC32, Polynomial 1, High Byte</td>
<td>487</td>
</tr>
<tr>
<td>VRCRC32P2L_1 mem16 —CRC32, Polynomial 2, Low Byte</td>
<td>488</td>
</tr>
<tr>
<td>VRCRC32P2H_1 mem16 —CRC32, Polynomial 2, High Byte</td>
<td>489</td>
</tr>
<tr>
<td>VRCCLR —Clear CRC Result Register</td>
<td>490</td>
</tr>
<tr>
<td>VMOV32 loc32,&quot;(0:16bitAddr) —VCRC to CPU register Move</td>
<td>491</td>
</tr>
<tr>
<td>VMOV32 &quot;(0:16bitAddr),loc32 —CPU to VCRC register Move</td>
<td>492</td>
</tr>
<tr>
<td>VRCRC24L_1 mem16 —CRC24, Polynomial 1, Low Byte</td>
<td>493</td>
</tr>
<tr>
<td>VRCRC24H_1 mem16 —CRC24, Polynomial 1, High Byte</td>
<td>494</td>
</tr>
<tr>
<td>VMOVZI VCRCPOLY, #16L —16-bit immediate Lower load to VCRCPOLY</td>
<td>495</td>
</tr>
<tr>
<td>VMOVIX VCRCPOLY, #16L —16-bit immediate Upper load to VCRCPOLY</td>
<td>496</td>
</tr>
<tr>
<td>VMOV16 VCRCSIZE, mem16 —16-bit write of the DSIZE half of the VCRCSIZE register from memory</td>
<td>497</td>
</tr>
<tr>
<td>VMOV16 VCRCSIZE, mem16 —16-bit write of the PSIZE half of the VCRCSIZE register from memory</td>
<td>498</td>
</tr>
<tr>
<td>VSETCRCSIZE #1!#3 —VCRCSIZE.DSIZE bit field to a 3-bit value and the VCRCSIZE.PSIZE bit field to a 5-bit value</td>
<td>499</td>
</tr>
<tr>
<td>VCRCL mem16 —compute CRC on VCRCSIZE bits using polynomial VCRCPOLY of size VCRCSIZE</td>
<td>500</td>
</tr>
<tr>
<td>VCRCH mem16 —compute CRC on VCRCSIZE bits using polynomial VCRCPOLY of size VCRCSIZE</td>
<td>501</td>
</tr>
<tr>
<td>VSWAPCRC —Byte Swap VCRCL</td>
<td>502</td>
</tr>
<tr>
<td>VMOV32 VCRCPOLY, mem32 —32-bit write of VCRCPOLY rom memory</td>
<td>503</td>
</tr>
<tr>
<td>VMOV32 VCRCSIZE, mem32 —32-bit write of VCRCSIZE from memory</td>
<td>504</td>
</tr>
<tr>
<td>VMOV32 mem32, VCRCPOLY —32-bit read of VCRCPOLY to memory</td>
<td>505</td>
</tr>
<tr>
<td>VMOV32 mem32, VCRCSIZE —32-bit read of VCRCSIZE register to memory</td>
<td>506</td>
</tr>
</tbody>
</table>
VMOV32 VCRC, mem32  32bit write of CRC result register (VCRC)

Operands

<table>
<thead>
<tr>
<th>VCRC</th>
<th>CRC result register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 0010
MSW: 0000 0000 mem32

Description

32bit write of CRC result register (VCRC).

VCRC = [mem32]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VMOV32 mem32, VCRC  

32bit read of CRC result register (VCRC)

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRC</td>
<td>CRC result register</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
</table>
| LSW: 1110 0010 0000 0110  
MSW: 0000 0000 mem32 | 32bit read of CRC result register (VCRC).  
[mem32] = VCRC |

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This instruction does not affect any flags in the VSTATUS register.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is a single-cycle instruction.</td>
</tr>
</tbody>
</table>
**VNOP** — *No operation*

<table>
<thead>
<tr>
<th><strong>VNOP</strong></th>
<th><em>No operation</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operand</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0010 0111</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>No operation.</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>This instruction does not affect any flags in the VSTATUS register.</td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>This is a single-cycle instruction.</td>
</tr>
</tbody>
</table>
### VMOV32 VSTATUS, mem32  32bit load of VSTATUS register from memory

#### Operands

<table>
<thead>
<tr>
<th>VSTATUS</th>
<th>VCRC status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32</td>
</tr>
</tbody>
</table>

#### Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0010</td>
<td>1011 0000 mem32</td>
</tr>
</tbody>
</table>

#### Description

32bit load of VSTATUS register from memory.

\[ VSTATUS = [\text{mem32}] \]

#### Flags

This instruction does not affect any flags in the VSTATUS register.

#### Pipeline

This is a single-cycle instruction.
## VMOV32 mem32, VSTATUS  
### 32bit store of VSTATUS register to memory

### Operands

<table>
<thead>
<tr>
<th>mem32</th>
<th>Pointer to a 32-bit memory location. This will be the source of the VMOV32</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSTATUS</td>
<td>VCRC status register</td>
</tr>
</tbody>
</table>

### Opcode

LSW: 1110 0010 0000 1101  
MSW: 0000 0000 mem32

### Description

32bit store of VSTATUS register to memory.  

\[ \text{[mem32]} = \text{VSTATUS} \]

### Flags

This instruction does not affect any flags in the VSTATUS register.

### Pipeline

This is a single-cycle instruction.
**VSETCRCMSGFLIP**  
*Set CRCMSGFLIP bit in the VSTATUS Register*

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0010 1100</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Set the CRCMSGFLIP bit in the VSTATUS register. This causes the VCRC to process message bits starting from least-significant to most-significant for CRC computation. In this case, bytes loaded from memory are “flipped” and then fed for CRC computation.</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>This instruction sets the CRCMSGFLIP bit in the VSTATUS register.</td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>This is a single-cycle instruction.</td>
</tr>
</tbody>
</table>
VCLRCRCMSGFLIP  —  Clear CRCMSGFLIP bit in the VSTATUS

Operands

none

Opcode

LSW: 1110 0101 0010 1101

Description

Clear the CRCMSGFLIP bit in the VSTATUS register. This causes the VCRC to process message bits starting from most-significant to least-significant for CRC computation. In this case, bytes loaded from memory are fed directly for CRC computation.

Flags

This instruction clears the CRCMSGFLIP bit in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VCRC8L_1 mem16  —  CRC8, Lowbyte

Operands

mem16  16-bit memory location

Opcode

LSW: 1110 0010 1100 1011
MSW: 0000 0000 mem16

Description

Compute CRC of one byte, Polynomial = 0x07. Calculate the CRC8 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

if VSTATUS.CRCMSGFLIP = 0
  temp[7:0] = [mem16][7:0]
else
  temp[7:0] = [mem16][0:7]

VCRC[7:0] = CRC8 (VCRC[7:0], temp[7:0])

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VCRC8H_1 mem16  **CRC8, High Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

LSW: 1110 0010 1100 1100  
MSW: 0000 0000 mem16

**Description**

Compute CRC of one byte, Polynomial = 0x07. Calculate the CRC8 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if VSTATUS.CRCMSGFLIP = 0
    temp[7:0] = [mem16][15:8]
else
    temp[7:0] = [mem16][8:15]

VCRC[7:0] = CRC8 (VCRC[7:0], temp[7:0])
```

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
VCRC16P1L_1 mem16  **CRC16, Polynomial 1, Low Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

| LSW: 1110 0010 1100 1110 |
| MSW: 0000 0000 mem16 |

**Description**

Compute CRC of one byte, Polynomial = 0x8005. Calculate the CRC16 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if VSTATUS.CRCMSGFLIP = 0
    temp[7:0] = [mem16][7:0]
else
    temp[7:0] = [mem16][0:7]

VCRC[15:0] = CRC16 (VCRC[15:0], temp[7:0])
```

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
VCRC16P1H_1 mem16  

**CRC16, Polynomial 1, High Byte**

**Operands**

<table>
<thead>
<tr>
<th>mem16</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

- **LSW:** 1110 0010 1100 1111
- **MSW:** 0000 0000 mem16

**Description**

Compute CRC of one byte, Polynomial = 0x8005. Calculate the CRC16 of the most significant byte pointed to by `mem16` and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if VSTATUS.CRCMSGFLIP = 0
    temp[7:0] = [mem16][15:8]
else
    temp[7:0] = [mem16][8:15]

VCRC[15:0] = CRC16(VCRC[15:0], temp[7:0])
```

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
VCRC16P2L_1 mem16 — CRC16, Polynomial 2, Low Byte

Operands

| mem16 | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 1110
MSW: 0000 0000 mem16

Description

Compute CRC of one byte, Polynomial = 0x1021. Calculate the CRC16 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

if VSTATUS.CRCMSGFLIP = 0
  temp[7:0] = [mem16][7:0]
else
  temp[7:0] = [mem16][0:7]

VCRC[15:0] = CRC16 (VCRC[15:0], temp[7:0])

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VCRC16P2H_1 mem16  CRC16, Polynomial 2, High Byte

Operands

| Mem16 | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 1111
MSW: 0000 0000 mem16

Description
Compute CRC of one byte, Polynomial = 0x1021. Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

if VSTATUS.CRCMSGFLIP = 0
  temp[7:0] = [mem16][15:8]
else
  temp[7:0] = [mem16][8:15]

VCRC[15:0] = CRC16(VCRC[15:0], temp[7:0])

Flags
This instruction does not affect any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.
**VCRC32L_1 mem16 — CRC32, Polynomial 1, Low Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

- **LSW:** 1110 0010 1100 0001
- **MSW:** 0000 0000 mem16

**Description**

Compute CRC of one byte, Polynomial = 0x04c11db7. Calculate the CRC32 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if VSTATUS.CRCMSGFLIP = 0
    temp[7:0] = [mem16][7:0]
else
    temp[7:0] = [mem16][0:7]

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
VCRC32H_1 mem16  CRC32, Polynomial 1, High Byte

Operands

| mem16 | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 0010
MSW: 0000 0000 mem16

Description

Compute CRC of one byte, Polynomial = 0x04c11db7. Calculate the CRC32 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if VSTATUS.CRCMSGFLIP = 0
  temp[7:0] = [mem16][15:8]
else
  temp[7:0] = [mem16][8:15]

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VCRC32P2L_1 mem16 — CRC32, Polynomial 2, Low Byte

VCRC32P2L_1 mem16  CRC32, Polynomial 2, Low Byte

Operands

| mem16 | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 1011
MSW: 0000 0000 mem16

Description

Compute CRC of one byte, Polynomial = 0x1edc6f41. Calculate the CRC32 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if VSTATUS.CRCMSGFLIP = 0
    temp[7:0] = [mem16][7:0]
else
    temp[7:0] = [mem16][0:7]

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VCRC32P2H_1 mem16  

**CRC32, Polynomial 2, High Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

| LSW: 1110 0010 1100 1011 |
| MSW: 0000 0000 mem16 |

**Description**

Compute CRC of one byte, Polynomial = 0x1edc6f41. Calculate the CRC32 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if VSTATUS.CRCMSGFLIP = 0
  temp[7:0] = [mem16][15:8]
else
  temp[7:0] = [mem16][8:15]

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
### VCRCLLR — Clear CRC Result Register

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0010 0100</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>VCRC = 0x0 Clear the VCRC register.</td>
</tr>
<tr>
<td></td>
<td>VCRC = 0x0000</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>This instruction does not affect any flags in the VSTATUS register.</td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>This is a single-cycle instruction.</td>
</tr>
</tbody>
</table>
VMOV32 loc32,*(0:16bitAddr)  VCRC to CPU register Move

**Operands**

<table>
<thead>
<tr>
<th>loc32</th>
<th>loc32 Destination Location (CPU register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*(0:16bitAddr)</td>
<td>Address of 32-bit Source Value (VCRC register)</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1011 1111 loc32  
MSW: IIII IIII IIII IIII

**Description**

VCRC to CPU register move is done using this instruction. Copy the 32-bit value referenced by 0:16bitAddr to the location indicated by loc32.

[optimization]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a two-cycle instruction.
VMOV32 *(0:16bitAddr),loc32 — CPU to VCRC register Move

Operands

<table>
<thead>
<tr>
<th>*(0:16bitAddr)</th>
<th>Address of 32-bit destination (VCRC register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc32</td>
<td>loc32 Source Location (CPU register)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1101 loc32
MSW: I I I I I I I I I I I I I I I I

Description

CPU to VCRC move is done using this instruction. Copy the 32-bit value referenced by loc32 to the location indicated by *(0:16bitAddr).

[0:16bitAddr] = [loc32]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a two-cycle instruction.
VCRC24L_1 mem16  **CRC24, Polynomial 1, Low Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

LSW: 1110 0010 1100 1011  
MSW: 0000 0001 mem16

**Description**

This instruction uses CRC24 polynomial == 0x5D6DCB. Calculate the CRC24 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if (VSTATUS[CRCMSGFLIP] == 0){
temp[7:0] = [mem16][7:0];
}else {
temp[7:0] = [mem16][0:7];
}
VCRC[23:0] = CRC24 (VCRC[23:0], temp[7:0])
```

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
**VCRC24H_1 mem16**  
**CRC24, Polynomial 1, High Byte**

**Operands**
- mem16: 16-bit memory location

**Opcode**
- LSW: 1110 0010 1100 1011
- MSW: 0000 0010 mem16

**Description**
This instruction uses CRC24 polynomial == 0x5D6DCB. Calculate the CRC24 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```c
if (VSTATUS[CRCMSGFLIP] == 0)
    temp[7:0] = [mem16][15:8];
else {
    temp[7:0] = [mem16][8:15];
}
VCRC[23:0] = CRC24 (VCRC[23:0], temp[7:0])
```

**Flags**
- This instruction does not affect any flags in the VSTATUS register.

**Pipeline**
- This is a single-cycle instruction.
VMOVZI VCRCPOLY, #16I  — 16-bit immediate Lower load to VCRCPOLY

**Operands**

| #16I | 16-bit immediate value |

**Opcode**

LSW: 1110 0111 0101 0100  
MSW: IIII IIII IIII IIII

**Description**

Load the lower 16-bits of the VCRCPOLY register with an immediate value. Clear the upper 16-bits of the register VCRCPOLY.

VCRCPOLY[31:16] = 0x0000 ;  
VCRCPOLY[15:0] = #16I;

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
VMOVIX VCRCPOLY, #16I  —  16-bit immediate Upper load to VCRCPOLY

Operands

#16I  16-bit immediate value

Opcode

LSW: 1110 0111 0101 0101
MSW: IIII IIII IIII IIII

Description

Load the upper 16-bits of the VCRCPLOY register with an immediate value. Leave the lower 16-bits of the register unchanged.

VCRCPOLY[31:16] = #16I ;
VCRCPOLY[15:0] = unchanged ;

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VMOV16 VCRCDSIZE, mem16 — 16-bit write of the DSIZE half of the VCRCSIZE register from memory

Operands

<table>
<thead>
<tr>
<th>VCRCDSIZE</th>
<th>VCRCDSIZE Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>Pointer to a 16-bit memory location. This will be the source for the VMOV16</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1100 1011
MSW: 0000 0101 mem16

Description

16-bit write of the DSIZE half of the VCRCSIZE register from memory.

VCRCSIZE.DSIZE = [mem16]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VMOV16 VCRCPSIZE, mem16  

16-bit write of the PSIZE half of the VCRCSIZE register from memory

Operands

| VCRCPSIZE | VCRCPSIZE register |
| mem16     | Pointer to a 16-bit memory location. This will be the source for the VMOV16 |

Opcode

LSW: 1110 0010 1100 1011  
MSW: 0000 0100 mem16

Description

16-bit write of the PSIZE half of the VCRCSIZE register from memory.

VCRCSIZE.PSIZE = [mem16]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VSETCRCSIZE #5I:#3I

VCRCSIZE.DSIZE bit field to a 3-bit value and the VCRCSIZE.PSIZE bit field to a 5-bit value

Operands

<table>
<thead>
<tr>
<th>#5I</th>
<th>5-bit immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#3I</td>
<td>3-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0111 0101 0110
MSW: xxxx xiii xxxI IIII

Description

Sets the VCRCSIZE.DSIZE bit field to a 3-bit value (i) and the VCRCSIZE.PSIZE bit field to a 5-bit value (I).

VCRCSIZE.DSIZE = #3'bi ;
VCRCSIZE.PSIZE = #5'bI ;

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
**VCRCL mem16**  compute CRC on VCRCDSIZE bits using polynomial VCRCPOLY of size VCRCPSIZE

### Operands

| mem16       | Pointer to a 16-bit memory location |

### Opcode

- **LSW**: 1110 0010 1100 1011
- **MSW**: 0000 0110 mem16

### Description

Compute CRC on VCRCDSIZE number of bits of memory using polynomial VCRCPOLY of size VCRCPSIZE.

```
if VSTATUS.CRCMSGFLIP = 0
    temp[VCRCDSIZE:0] = [mem16][VCRCDSIZE:0]
else
    temp[VCRCDSIZE:0] = [mem16][0:VCRCDSIZE]

VCRC[VCRCPSIZE:0] = CRC(VCRC[VCRCPSIZE:0], temp[VCRCDSIZE:0])
```

### Flags

This instruction does not affect any flags in the VSTATUS register.

### Pipeline

This is a three-cycle instruction.
VCRCH mem16

compute CRC on VCRCDSIZE bits using polynomial VCRCPOLY of size VCRCPSIZE

Operands

mem16 Pointer to a 16-bit memory location

Opcode

LSW: 1110 0010 1100 1011
MSW: 0000 0111 mem16

Description

compute CRC on VCRCDSIZE number of bits of memory using polynomial VCRCPOLY of size VCRCPSIZE.

if VSTATUS.CRCMSGFLIP = 0
    temp[VCRCDSIZE:0] = [mem16][VCRCDSIZE+8:8]
else
    temp[VCRCDSIZE:0] = [mem16][8:VCRCDSIZE+8]

VCRC[VCRCPSIZE:0] = CRC(VCRC[VCRCPSIZE:0], temp[VCRCDSIZE:0])

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a three-cycle instruction.
### VSWAPCRC — Byte Swap VCRCL

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0010 1110</td>
</tr>
</tbody>
</table>
| **Description** | Byte swap VCRCL register.  
VCRC[31:16] = unchanged;  
Swap VCRC[15:8] with VCRC[7:0] |
| **Flags**    | This instruction does not affect any flags in the VSTATUS register. |
| **Pipeline** | This is a single-cycle instruction. |
VMOV32 VCRCPOLY, mem32  32-bit write of VCRCPOLY rom memory

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRCPOLY</td>
<td>VCRCPOLY Register (Destination)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>0011</td>
<td>1111 0010</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>mem32</td>
</tr>
</tbody>
</table>

Description

32-bit write of VCRCPOLY register from memory.

VCRCPOLY = [mem32];

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
VMOV32 VCRCSIZE, mem32 — 32-bit write of VCRCSIZE from memory

**Operands**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRCSIZE</td>
<td>VCRCSIZE Register (Destination)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW</td>
<td>1110 0011 1111 0010</td>
</tr>
<tr>
<td>MSW</td>
<td>0000 0010 mem32</td>
</tr>
</tbody>
</table>

**Description**

32-bit write of VCRCSIZE register from memory.

VCRCSIZE ← [mem32];

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.
VMOV32 mem32, VCRCPOLY  

32-bit read of VCRCPOLY to memory

Operands

<table>
<thead>
<tr>
<th>Mem32</th>
<th>Pointer to a 32-bit memory location. This will be the destination of the VMOV32.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRCPOLY</td>
<td>VCRCPOLY Register (Source).</td>
</tr>
</tbody>
</table>

Opcode

| LSW: 1110 0010 0000 0110 |
| MSW: 0000 0001 mem32 |

Description

32-bit read of VCRCPOLY register to memory.

\[[mem32] = VCRCPOLY;\]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.
### VMOV32 mem32, VCRCSIZE  32-bit read of VCRCSIZE register to memory

#### Operands

<table>
<thead>
<tr>
<th>Mem32</th>
<th>Pointer to a 32-bit memory location. This will be the destination of the VMOV32.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRCSIZE</td>
<td>VCRCSIZE Register (Source)</td>
</tr>
</tbody>
</table>

#### Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0010 0000 0110</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 0010 mem32</td>
</tr>
</tbody>
</table>

#### Description

32-bit read of VCRCSIZE register to memory.

\[
[\text{mem32}] = \text{VCRCSIZE};
\]

#### Flags

This instruction does not affect any flags in the VSTATUS register.

#### Pipeline

This is a single-cycle instruction.
This chapter provides an overview of the architectural structure and instruction set of the Viterbi, Complex Math and CRC Unit (VCU-II) and describes the architecture, pipeline, instruction set, and interrupts. The VCU is a fully-programmable block which accelerates the performance of communications-based algorithms. In addition to eliminating the need for a second processor to manage the communications link, the performance gains of the VCU provides headroom for future system growth and higher bit rates or, conversely, enables devices to operate at a lower MHz to reduce system cost and power consumption.

Any references to VCU or VCU-II in this chapter relate to Type 2 specifically. Information pertaining to an older VCU will have the module type listed explicitly. See the TMS320x28xx, 28xxx DSP Peripheral Reference Guide (SPRU566) for a list of all devices with a VCU module of the same type, to determine the differences between the types, and for a list of device-specific differences within a type.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Overview</td>
<td>508</td>
</tr>
<tr>
<td>5.2 Components of the C28x Plus VCU</td>
<td>509</td>
</tr>
<tr>
<td>5.3 Register Set</td>
<td>513</td>
</tr>
<tr>
<td>5.4 Pipeline</td>
<td>521</td>
</tr>
<tr>
<td>5.5 Instruction Set</td>
<td>526</td>
</tr>
<tr>
<td>5.6 Rounding Mode</td>
<td>746</td>
</tr>
</tbody>
</table>
5.1 Overview

The C28x with VCU (C28x+VCU) processor extends the capabilities of the C28x fixed-point or floating-point CPU by adding registers and instructions to support the following algorithm types:

- **Viterbi decoding**
  
  Viterbi decoding is commonly used in baseband communications applications. The viterbi decode algorithm consists of three main parts: branch metric calculations, compare-select (viterbi butterfly) and a traceback operation. shows a summary of the VCU performance for each of these operations.

  **Table 5-1. Viterbi Decode Performance**

<table>
<thead>
<tr>
<th>Viterbi Operation</th>
<th>VCU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Metric Calculation (code rate = 1/2)</td>
<td>1</td>
</tr>
<tr>
<td>Branch Metric Calculation (code rate = 1/3)</td>
<td>2p</td>
</tr>
<tr>
<td>Viterbi Butterfly (add-compare-select)</td>
<td>2 (1)</td>
</tr>
<tr>
<td>Traceback per Stage</td>
<td>3 (2)</td>
</tr>
</tbody>
</table>

(1) C28x CPU takes 15 cycles per butterfly.
(2) C28x CPU takes 22 cycles per stage.

- **Cyclic redundancy check (CRC)**
  
  CRC algorithms provide a straightforward method for verifying data integrity over large data blocks, communication packets, or code sections. The C28x+VCU can perform 8-, 16-, 24-, and 32-bit CRCs. For example, the VCU can compute the CRC for a block length of 10 bytes in 10 cycles. A CRC result register contains the current CRC which is updated whenever a CRC instruction is executed.

- **Complex math**
  
  Complex math is used in many applications. The VCU A few of which are:
  
  - Fast Fourier transform (FFT)
    
    The complex FFT is used in spread spectrum communications, as well in many signal processing algorithms.
  
  - Complex filters
    
    Complex filters improve data reliability, transmission distance, and power efficiency. The C28x+VCU can perform a complex I and Q multiply with coefficients (four multiplies) in a single cycle. In addition, the C28x+VCU can read/write the real and imaginary parts of 16-bit complex data to memory in a single cycle.

  **Table 5-2** shows a summary of the VCU operations enabled by the VCU:

  **Table 5-2. Complex Math Performance**

<table>
<thead>
<tr>
<th>Complex Math Operation</th>
<th>VCU Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Or Subtract</td>
<td>1</td>
<td>32 +/- 32 = 32-bit (Useful for filters)</td>
</tr>
<tr>
<td>Add or Subtract</td>
<td>1</td>
<td>16 +/- 32 = 15-bit (Useful for FFT)</td>
</tr>
<tr>
<td>Multiply</td>
<td>2p</td>
<td>16 x 16 = 32-bit</td>
</tr>
<tr>
<td>Multiply &amp; Accumulate (MAC)</td>
<td>2p</td>
<td>32 + 32 = 32-bit, 16 x 16 = 32-bit</td>
</tr>
<tr>
<td>RPT MAC</td>
<td>2p+N</td>
<td>Repeat MAC. Single cycle after the first operation.</td>
</tr>
</tbody>
</table>

This C28x+VCU draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The C2000 features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and modified Harvard architecture (usable in Von Neumann mode). The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation. The modified Harvard architecture of the CPU enables instruction and data fetches to be performed in parallel. The CPU can read instructions and data while it writes data simultaneously to maintain the single-cycle instruction operation across the pipeline. The CPU does this over six separate address/data buses.
Throughout this document the following notations are used:

- C28x refers to the C28x fixed-point CPU.
- C28x plus Floating-Point and C28x+FPU both refer to the C28x CPU with enhancements to support IEEE single-precision floating-point operations.
- C28x plus VCU and C28x+VCU both refer to the C28x CPU with enhancements to support viterbi decode, complex math, forward error correcting algorithms, and CRC.
- Some devices have both the FPU and the VCU. These are referred to as C28x+FPU+VCU.

5.2 Components of the C28x Plus VCU

The VCU extends the capabilities of the C28x CPU and C28x+FPU processors by adding additional instructions. No changes have been made to existing instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x are completely compatible with the C28x+VCU. All of the features of the C28x documented in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430) apply to the C28x+VCU. All features documented in the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUE02) apply to the C28x+FPU+VCU. Figure 5-1 shows the block diagram of the VCU.

Figure 5-1. C28x + VCU Block Diagram
The C28x+VCU contains the same features as the C28x fixed-point CPU:

- A central processing unit for generating data and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory.
- Emulation logic for monitoring and controlling various parts and functions of the device and for testing device operation. This logic is identical to that on the C28x fixed-point CPU.
- Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts. This logic is identical to the C28x fixed-point CPU.
- Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- Address register arithmetic unit (ARAU). The ARAU generates data memory addresses and increments or decrements pointers in parallel with ALU operations.
- Fixed-Point instructions are pipeline protected. This pipeline for fixed-point instructions is identical to that on the C28x fixed-point CPU. The CPU implements an 8-phase pipeline that prevents a write to and a read from the same location from occurring out of order.
- Barrel shifter. This shifter performs all left and right shifts of fixed-point data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.
- Fixed-Point Multiplier. The multiplier performs 32-bit × 32-bit 2s-complement multiplication with a 64-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.

The VCU adds the following features:

- Instructions to support Cyclic Redundancy Check (CRC) or a polynomial code checksum
  - CRC8
  - CRC16
  - CRC32
  - CRC24
- Clocked at the same rate as the main CPU (SYSCLKOUT).
- Instructions to support a software implementation of a Viterbi Decoder of constraint length 4 - 7 and code rates of 1/2 and 1/3
  - Branch metrics calculations
  - Add-Compare Select or Viterbi Butterfly
  - Traceback
- Complex Math Arithmetic Unit
  - Add or Subtract
  - Multiply
  - Multiply and Accumulate (MAC)
  - Repeat MAC (RPT || MAC).
- Independent register space. These registers function as source and destination registers for VCU instructions.
- Some VCU instructions require pipeline alignment. This alignment is done through software to allow the user to improve performance by taking advantage of required delay slots. See Section 5.4 for more information.

Devices with the floating-point unit also include:

- Floating point unit (FPU). The 32-bit FPU performs IEEE single-precision floating-point operations.
- Dedicated floating-point registers.
5.2.1 Emulation Logic

The emulation logic is identical to that on the C28x fixed-point CPU. This logic includes the following features. For more details about these features, refer to the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430):

- Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline.
- A counter for performance benchmarking.
- Multiple debug events. Any of the following debug events can cause a break in program execution:
  - A breakpoint initiated by the ESTOP0 or ESTOP1 instruction.
  - An access to a specified program-space or data-space location. When a debug event causes the C28x to enter the debug-halt state, the event is called a break event.
- Real-time mode of operation.

5.2.2 Memory Map

Like the C28x, the C28x+VCU uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x+VCU designs are uniformly mapped to both program and data space. For specific details about each of the map segments, see the device-specific data manual.

5.2.3 CPU Interrupt Vectors

The C28x+VCU interrupt vectors are identical to those on the C28x CPU. Sixty-four addresses in program space are set aside for a table of 32 CPU interrupt vectors. For more information about the CPU vectors, see TMS320C28x CPU and Instruction Set Reference Guide (literature number SPRU430). Typically the CPU interrupt vectors are only used during the boot up of the device by the boot ROM. Once an application has taken control it should initialize and enable the peripheral interrupt expansion block (PIE).

5.2.4 Memory Interface

The C28x+VCU memory interface is identical to that on the C28x. The C28x+VCU memory map is accessible outside the CPU by the memory interface, which connects the CPU logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed. The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the CPU supports special byte-access instructions that can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.

5.2.5 Address and Data Buses

Like the C28x, the memory interface has three address buses:

- PAB: Program address bus: The 22-bit PAB carries addresses for reads and writes from program space.
- DRAB: Data-read address bus: The 32-bit DRAB carries addresses for reads from data space.
- DWAB: Data-write address bus: The 32-bit DWAB carries addresses for writes to data space.

The memory interface also has three data buses:

- PRDB: Program-read data bus: The 32-bit PRDB carries instructions during reads from program space.
- DRDB: Data-read data bus: The 32-bit DRDB carries data during reads from data space.
- DWDB: Data-/Program-write data bus: The 32-bit DWDB carries data during writes to data space or program space.
A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time. This behavior is identical to the C28x CPU.

5.2.6 Alignment of 32-Bit Accesses to Even Addresses

The C28x+VCU expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU will begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the CPU.

You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space.
## 5.3 Register Set

Devices with the C28x+VCU include the standard C28x register set plus an additional set of VCU specific registers. The additional VCU registers are the following:
- Result registers: VR0, VR1... VR8
- Traceback registers: VT0, VT1
- Configuration and status register: VSTATUS
- CRC result register: VCRC
- Repeat block register: RB

Figure 5-2 shows the register sets for the 28x CPU, the FPU and the VCU. The following section discusses the VCU register set in detail.

### Figure 5-2. C28x + FPU + VCU Registers

<table>
<thead>
<tr>
<th>Standard C28x Register Set</th>
<th>Additional 32-bit FPU Registers</th>
<th>Standard VCU Register Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC (32-bit)</td>
<td>R0H (32-bit)</td>
<td>VR0</td>
</tr>
<tr>
<td>P (32-bit)</td>
<td>R1H (32-bit)</td>
<td>VR1</td>
</tr>
<tr>
<td>XT (32-bit)</td>
<td>R2H (32-bit)</td>
<td>VR2</td>
</tr>
<tr>
<td>XAR0 (32-bit)</td>
<td>R3H (32-bit)</td>
<td>VR3</td>
</tr>
<tr>
<td>XAR1 (32-bit)</td>
<td>R4H (32-bit)</td>
<td>VR4</td>
</tr>
<tr>
<td>XAR2 (32-bit)</td>
<td>R5H (32-bit)</td>
<td>VR5</td>
</tr>
<tr>
<td>XAR3 (32-bit)</td>
<td>R6H (32-bit)</td>
<td>VR6</td>
</tr>
<tr>
<td>XAR4 (32-bit)</td>
<td>R7H (32-bit)</td>
<td>VR7</td>
</tr>
<tr>
<td>XAR5 (32-bit)</td>
<td></td>
<td>VR8</td>
</tr>
<tr>
<td>XAR6 (32-bit)</td>
<td>FPU Status Register (STF)</td>
<td>VT0</td>
</tr>
<tr>
<td>XAR7 (32-bit)</td>
<td>Repeat Block Register (RB)</td>
<td>VT1</td>
</tr>
<tr>
<td>PC (22-bit)</td>
<td></td>
<td>VSTATUS</td>
</tr>
<tr>
<td>RPC (22-bit)</td>
<td></td>
<td>VCRC</td>
</tr>
<tr>
<td>DP (16-bit)</td>
<td></td>
<td>VSM0</td>
</tr>
<tr>
<td>SP (16-bit)</td>
<td></td>
<td>VSM1</td>
</tr>
<tr>
<td>ST0 (16-bit)</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>ST1 (16-bit)</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>IER (16-bit)</td>
<td></td>
<td>VSM63</td>
</tr>
<tr>
<td>IFR (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGIER (16-bit)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FPU registers R0H - R7H and STF are shadowed for fast context save and restore.
5.3.1 VCU Register Set

Table 5-3 describes the VCU module register set. The last three columns indicate whether the particular module within the VCU can make use of the register.

Table 5-4 lists the CPU registers available on devices with the C28x, the C28x+FPU, the C28x+VCU and the C28x+FPU+VCU.

**Table 5-3. VCU Register Set**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Size</th>
<th>Description</th>
<th>Viterbi</th>
<th>Complex Math</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>32 bits</td>
<td>General purpose register 0</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR1</td>
<td>32 bits</td>
<td>General purpose register 1</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR2</td>
<td>32 bits</td>
<td>General purpose register 2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR3</td>
<td>32 bits</td>
<td>General purpose register 3</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR4</td>
<td>32 bits</td>
<td>General purpose register 4</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR5</td>
<td>32 bits</td>
<td>General purpose register 5</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR6</td>
<td>32 bits</td>
<td>General purpose register 6</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR7</td>
<td>32 bits</td>
<td>General purpose register 7</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VR8</td>
<td>32 bits</td>
<td>General purpose register 8</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VT0</td>
<td>32 bits</td>
<td>32-bit transition bit register 0</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VT1</td>
<td>32 bits</td>
<td>32-bit transition bit register 1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VSTATUS</td>
<td>32 bits</td>
<td>VCU status and configuration register (1)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VCRC</td>
<td>32 bits</td>
<td>Cyclic redundancy check (CRC) result register</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>VSM0-VSM63</td>
<td>32 bits</td>
<td>Viterbi Decoding State Metric registers</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VRx.By</td>
<td>32 bits</td>
<td>Aliased address space for each byte of the VRx registers, left-shifted by one</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

(1) Debugger writes are not allowed to the VSTATUS register.
## Table 5-4. 28x CPU Register Summary

<table>
<thead>
<tr>
<th>Register</th>
<th>C28x CPU</th>
<th>C28x+FPU</th>
<th>C28x+VCU</th>
<th>C28x+FPU+VCU</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Fixed-point accumulator</td>
</tr>
<tr>
<td>AH</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High half of ACC</td>
</tr>
<tr>
<td>AL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of ACC</td>
</tr>
<tr>
<td>XAR0 - XAR7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Auxiliary register 0 - 7</td>
</tr>
<tr>
<td>AR0 - AR7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of XAR0 - XAR7</td>
</tr>
<tr>
<td>DP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Data-page pointer</td>
</tr>
<tr>
<td>IFR</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Interrupt flag register</td>
</tr>
<tr>
<td>IER</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Interrupt enable register</td>
</tr>
<tr>
<td>DBGIER</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Debug interrupt enable register</td>
</tr>
<tr>
<td>P</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Fixed-point product register</td>
</tr>
<tr>
<td>PH</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High half of P</td>
</tr>
<tr>
<td>PL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of P</td>
</tr>
<tr>
<td>PC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Program counter</td>
</tr>
<tr>
<td>RPC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Return program counter</td>
</tr>
<tr>
<td>SP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>ST0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Status register 0</td>
</tr>
<tr>
<td>ST1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Status register 1</td>
</tr>
<tr>
<td>XT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Fixed-point multiplicand register</td>
</tr>
<tr>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High half of XT</td>
</tr>
<tr>
<td>TL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low half of XT</td>
</tr>
<tr>
<td>ROH - R7H</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Floating-point Unit result registers</td>
</tr>
<tr>
<td>STF</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Floating-point Uint status register</td>
</tr>
<tr>
<td>RB</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Repeat block register</td>
</tr>
<tr>
<td>VR0 - VR8</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>VCU general purpose registers</td>
</tr>
<tr>
<td>VT0, VT1</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>VCU transition bit register 0 and 1</td>
</tr>
<tr>
<td>VSTATUS</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>VCU status and configuration</td>
</tr>
<tr>
<td>CRC</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>CRC result register</td>
</tr>
<tr>
<td>VSM0-VSM63</td>
<td>No</td>
<td>No</td>
<td>Yes(^{(1)})</td>
<td>Yes(^{(1)})</td>
<td>Viterbi State Metric Registers</td>
</tr>
<tr>
<td>VRx.By</td>
<td>No(^{(1)})</td>
<td>No(^{(1)})</td>
<td>Yes(^{(1)})</td>
<td>Yes(^{(1)})</td>
<td>Aliased address space for each byte of the VRx registers, left-shifted by one</td>
</tr>
</tbody>
</table>

\(^{(1)}\) Present on Type-2 VCU only
5.3.2 VCU Status Register (VSTATUS)

The VCU status register (VSTATUS) register is described in Figure 5-3. There is no single instruction to directly transfer the VSTATUS register to a C28x register. To transfer the contents:
1. Store VSTATUS into memory using VMOV32 mem32, VSTATUS instruction
2. Load the value from memory into a main C28x CPU register.

Configuration bits within the VSTATUS registers are set or cleared using VCU instructions.

![Figure 5-3. VCU Status Register (VSTATUS)](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>CRCMSGFL</td>
<td>0</td>
<td>CRC Message Flip&lt;br&gt;This bit affects the order in which the bits in the message are taken for CRC calculation by all the CRC instructions. Message bits are taken starting from most-significant to least-significant for CRC computation. In this case, bytes loaded from memory are fed directly for CRC computation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Message bits are taken starting from least-significant to most-significant for CRC computation. In this case, bytes loaded from memory are “flipped” and then fed for CRC computation.</td>
</tr>
<tr>
<td>30</td>
<td>DIVE</td>
<td>0</td>
<td>Divide-by-zero Error&lt;br&gt;Indicates whether a “divide by zero” occurred during a VMOD32 computation. This bit is cleared by executing the VCLRDIVE instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>29-27</td>
<td>K</td>
<td>0x7</td>
<td>Constraint Length for Viterbi Decoding&lt;br&gt;This field sets the constraint length for the Viterbi decoding algorithm. It accepts values of 4 to 7. Values outside this range will be treated as 7 by the hardware.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>26-24</td>
<td>GFORDER</td>
<td>0x7</td>
<td>Galois Field Polynomial Order&lt;br&gt;This field holds the Order of the polynomial for all the Galois Field instructions. This field is initialized by the VGFINIT mem16 instruction. The actual order of the polynomial is GFORDER+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>23-16</td>
<td>GFPOLY</td>
<td>0</td>
<td>Galois Field Polynomial&lt;br&gt;This field holds the Polynomial for all the Galois Field instructions. This field is initialized by the VGFINIT mem16 instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>OPACK</td>
<td>0</td>
<td>Viterbi Traceback Packing Order&lt;br&gt;This bit affects the packing order of the traceback output bits (using the VTRACE instructions)&lt;br&gt;Big-endian (compatible with VCU Type-0 output packing order)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Little-endian (VCU Type-2 mode)</td>
</tr>
<tr>
<td>14</td>
<td>CPACK</td>
<td>0</td>
<td>Complex Packing Order&lt;br&gt;This bit affects the packing order of the 16-bit real and 16-bit imaginary part of a complex numbers inside the 32-bit general purpose VRx register.&lt;br&gt;VRx[31:16] holds Real part, VRx[15:0] holds Imaginary part (VCU-I compatible mode)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>VRx[31:16] holds Imaginary part; VRx[15:0] holds Real part</td>
</tr>
</tbody>
</table>

(1) Present on Type-2 VCU only.
### Table 5-5. VCU Status (VSTATUS) Register Field Descriptions (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 13   | OVRI  | 0     | Overflow or Underflow Flag: Imaginary Part 
|      |       | 1     | Indicates an overflow or underflow has occurred during the computation of the imaginary part of operations shown in Table 10-6. This bit will be set regardless of the value of the VSTATUS[SAT] bit. OVRI bit will remain set until it is cleared by executing the VCLROVFI instruction. |
| 12   | OVFR  | 0     | Overflow or Underflow Flag: Real Part 
|      |       | 1     | Indicates overflow or underflow has occurred during a real number calculation for operations shown in Table 5-6. This bit will be set regardless of the value of the VSTATUS[SAT] bit. This bit will remain set until it is cleared by executing the VCLROVFR instruction. |
| 11   | RND   | 0     | Rounding when a right-shift operation is performed the lower bits of the value will be lost. The RND bit determines if the shifted value is rounded or if the shifted-out bits are simply truncated. This is described in Section 5.3.2. Operations which use right-shift and rounding are shown in Table 5-6. The RND bit is set by the VRNDON instruction and cleared by the VRNDOFF instruction. 
|      |       | 1     | Rounding is performed. Bits shifted out right are truncated. |
| 10   | SAT   | 0     | Saturation  
|      |       | 1     | Saturation is performed. |
| 9-5  | SHIFTL| 0     | Left Shift  
|      |       | 0x01-0x1F | Operations which use left-shift are shown in Table 5-6. The shift SHIFTL field can be set or cleared by the VSETSHL instruction. |
|      |       | 1     | Left Shift  
|      |       | 0x01-0x1F | Refer to the instruction description for information on how the operation is affected by the shift value. During the left-shift, the lower bits are filled with 0's. |
| 4-0  | SHIFTR| 0     | Right Shift 
|      |       | 0x01-0x1F | Operations which use right-shift and rounding are shown in Table 5-6. The shift SHIFTR field can be set or cleared by the VSETSHR instruction. |
|      |       | 1     | Right Shift 
|      |       | 0x01-0x1F | Refer to the instruction descriptions for information on how the operation is affected by the shift value. During the right-shift, the lower bits are lost, and the shifted value is sign extended. If rounding is enabled (VSTATUS[RND] == 1), then the value will be rounded instead of truncated. |

Table 5-6 shows a summary of the operations that are affected by or modify bits in the VSTATUS register.

### Table 5-6. Operation Interaction With VSTATUS Bits

<table>
<thead>
<tr>
<th>Operation(1)</th>
<th>Description</th>
<th>OVFI</th>
<th>OVFR</th>
<th>RND</th>
<th>SAT</th>
<th>SHIFTL</th>
<th>SHIFTR</th>
<th>CPACK</th>
<th>QPACk</th>
<th>DIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITDLADDSUB</td>
<td>Viterbi Add and Subtract Low</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VITDHADDSUB</td>
<td>Viterbi Add and Subtract High</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VITDLSUBADD</td>
<td>Viterbi Subtract and Add Low</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VITDHSUBADD</td>
<td>Viterbi Subtract and Add High</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VITBM2</td>
<td>Viterbi Branch Metric CR 1/2</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VITBM3</td>
<td>Viterbi Branch Metric CR 1/3</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VTRACE(2)</td>
<td>Viterbi Trace-back</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Some parallel instructions also include these operations. In this case, the operation will also modify, or be affected by, VSTATUS bits as when used as part of a parallel instruction.

(2) Present on Type-2 VCU only.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>OVFI</th>
<th>OVFR</th>
<th>RND</th>
<th>SAT</th>
<th>SHIFTL</th>
<th>SHIFTR</th>
<th>CPACK</th>
<th>OPACK</th>
<th>DIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITSTAGE[25]</td>
<td>Viterbi Compute 32 Butterfly</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCADD</td>
<td>Complex 32 + 32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCDADD16</td>
<td>Complex 16 + 32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCDSUB16</td>
<td>Complex 16 - 32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCMAC</td>
<td>Complex 32 + 32 = 32, 16 x 16 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCMAC[25]</td>
<td>Complex Conjugate 32 + 32 = 32, 16 x 16 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCMPY</td>
<td>Complex 16 x 16 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCCMPY[25]</td>
<td>Complex Conjugate 16 x 16 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCSUB</td>
<td>Complex 32 - 32 = 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCSHL16[25]</td>
<td>Complex Shift Left</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCHR16[25]</td>
<td>Complex Shift Right</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCMAG[20]</td>
<td>Complex Number Magnitude</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VNNEG</td>
<td>Two’s Complement Negation</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VASHR32[25]</td>
<td>Arithmetic Shift Right</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VASHL32[25]</td>
<td>Arithmetic Shift Left</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VMPYADD[25]</td>
<td>Arithmetic Multiply Add 16 + ((16 x 16) &gt;&gt; SHR) = 16</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VCFFT[10]</td>
<td>Complex FFT calculation step (x = 1 – 10)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VMOD32</td>
<td>Modulo 32 % 16 = 16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>
5.3.3 Repeat Block Register (RB)

The repeat block instruction (RPTB) applies to devices with the C28x+FPU and the C28x+VCU. This instruction allows you to repeat a block of code as shown in Example 5-1.

Example 5-1. The Repeat Block (RPTB) Instruction uses the RB Register

```
; find the largest element and put its address in XAR6
;
; This example makes use of floating-point (C28x + FPU) instructions
;
MOV32 R0H, *XAR0++
.align 2 ; Aligns the next instruction to an even address
NOP ; Makes RPTB odd aligned - required for a block size of 8
RPTB VECTOR_MAX_END, AR7 ; RA is set to 1
MOVL ACC,XAR0
MOV32 R1H, *XAR0++ ; RSIZE reflects the size of the RPTB block
MAXF32 R0H, R1H ; in this case the block size is 8
MOVS0 NF, 2F
MOVL XAR6, ACC, LT
VECTOR_MAX_END: ; RE indicates the end address. RA is cleared
```

The C28x FPU or VCU automatically populates the RB register based on the execution of a RPTB instruction. This register is not normally read by the application and does not accept debugger writes.

![Figure 5-4. Repeat Block Register (RB)](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RAS</td>
<td>0</td>
<td>Repeat Block Active Shadow Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Repeat block was active when the interrupt was taken.</td>
</tr>
<tr>
<td>30</td>
<td>RA</td>
<td>0</td>
<td>Repeat Block Active Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>This bit is set when the RPTB instruction is executed to indicate that a RPTB is currently active.</td>
</tr>
<tr>
<td>29-23</td>
<td>RSIZE</td>
<td>0-7</td>
<td>Repeat Block Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8/9-0x7F</td>
<td>A RPTB block that starts at an even address must include at least 9 16-bit words and a block that starts at an odd address must include at least 8 16-bit words. The maximum block size is 127 16-bit words. The codegen assembler will check for proper block size and alignment.</td>
</tr>
<tr>
<td>Bits</td>
<td>Field</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 22-16 | RE    |       | Repeat Block End Address  
This 7-bit value specifies the end address location of the repeat block. The RE value is calculated by hardware based on the RSIZE field and the PC value when the RPTB instruction is executed.  
RE = lower 7 bits of (PC + 1 + RSIZE) |
| 15-0  | RC    | 0     | Repeat Count  
The block will not be repeated; it will be executed only once. In this case the repeat active, RA, bit will not be set.  
This 16-bit value determines how many times the block will repeat. The counter is initialized when the RPTB instruction is executed and is decremented when the PC reaches the end of the block. When the counter reaches zero, the repeat active bit is cleared and the block will be executed one more time. Therefore the total number of times the block is executed is RC+1. |
5.4 Pipeline

This section describes the VCU pipeline stages and presents cases where pipeline alignment must be considered.

5.4.1 Pipeline Overview

The C28x VCU pipeline is identical to the C28x pipeline for all standard C28x instructions. In the decode2 stage (D2), it is determined if an instruction is a C28x instruction, a FPU instruction, or a VCU instruction. The pipeline flow is shown in Figure 5-5.

Notice that stalls due to normal C28x pipeline stalls (D2) and memory waitstates (R2 and W) will also stall any C28x VCU instruction. Most C28x VCU instructions are single cycle and will complete in the VCU E1 or W stage which aligns to the C28x pipeline. Some instructions will take an additional execute cycle (E2). For these instructions you must wait a cycle for the result from the instruction to be available. The rest of this section will describe when delay cycles are required. Keep in mind that the assembly tools for the C28x+VCU will issue an error if a delay slot has not been handled correctly.

Figure 5-5. C28x + FCU + VCU Pipeline
5.4.2 General Guidelines for VCU Pipeline Alignment

The majority of the VCU instructions do not require any special pipeline considerations. This section lists the few operations that do require special consideration.

While the C28x+VCU assembler will issue errors for pipeline conflicts, you may still find it useful to understand when software delays are required. This section describes three guidelines you can follow when writing C28x+VCU assembly code.

VCU instructions that require delay slots have a ‘p’ after their cycle count. For example ‘2p’ stands for 2 pipelined cycles. This means that an instruction can be started every cycle, but the result of the instruction will only be valid one instruction later.

Table 5-8 outlines the instructions that need delay slots.

<table>
<thead>
<tr>
<th>Operation(1)</th>
<th>Description</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITBM3</td>
<td>Viterbi Branch Metric CR 1/3</td>
<td>2p/2(2)</td>
</tr>
<tr>
<td>VCMAC</td>
<td>Complex 32 + 32 = 32, 16 x 16 = 32</td>
<td>2p</td>
</tr>
<tr>
<td>VCCMAC(3)</td>
<td>Complex Conjugate 32 + 32 = 32, 16 x 16 = 32</td>
<td>2p</td>
</tr>
<tr>
<td>VCMPY</td>
<td>Complex 16 x 16 = 32</td>
<td>2p</td>
</tr>
<tr>
<td>VCCMPY(3)</td>
<td>Complex Conjugate 16 x 16 = 32</td>
<td>2p</td>
</tr>
<tr>
<td>VCMAG(3)</td>
<td>Complex Number Magnitude</td>
<td>2</td>
</tr>
<tr>
<td>VCFFTx(3)</td>
<td>Complex FFT calculation step (x = 1 – 10)</td>
<td>2p/2(2)</td>
</tr>
<tr>
<td>VMOD32</td>
<td>Modulo 32 % 16 = 16</td>
<td>9p</td>
</tr>
<tr>
<td>VMPYADD(3)</td>
<td>Arithmetic Multiply Add 16 + ((16 x 16) &gt;&gt; SHR) = 16</td>
<td>2p</td>
</tr>
</tbody>
</table>

(1) Some parallel instructions also include these operations. In this case, the operation will also modify, or be affected by, VSTATUS bits as when used as part of a parallel instruction.
(2) Variations of the instruction execute differently. In these cases, the user is referred to the description Example 5-2 of the instruction(s) in Section 5.5.
(3) Present on Type-2 VCU only.

An example of the complex multiply instruction is shown in Example 5-2. VCMPY is a 2p instruction and therefore requires one delay slot. The destination registers for the operation, VR2 and VR3, will be updated one cycle after the instruction for a total of two cycles. Therefore, a NOP or instruction that does not use VR2 or VR3 must follow this instruction.

Any memory stall or pipeline stall will also stall the VCU. This keeps the VCU aligned with the C28x pipeline and there is no need to change the code based on the waitstates of a memory block.

Example 5-2. 2p Instruction Pipeline Alignment

```
VCMPY VR3, VR2, VR1, VR0 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
; <-- VCMPY completes, VR2 and VR3 updated
NOP ; Any instruction
```
5.4.3 Parallel Instructions

Parallel instructions are single opcodes that perform two operations in parallel. The guidelines provided in Section 5.4.2 apply to parallel instructions as well. In this case the cycle count will be given for both operations. For example, a branch metric calculation for code rate of 1/3 with a parallel load takes 2p/1 cycles. This means the branch metric portion of the operation takes two pipelined cycles while the move portion of the operation is single cycle. NOPs or other non conflicting instructions must be inserted to align the branch metric calculation portion of the operation as shown in Example 5-4.

Example 5-3. Branch Metric CR 1/2 Calculation with Parallel Load

```
; VITBM2 || VMOV32 instruction: branch metrics calculation with parallel load
; VBITM2 is a 1 cycle operation (code rate = 1/2)
; VMOV32 is a 1 cycle operation
;
VITBM2 VR0 ; Load VR0 with the 2 branch metrics
|| VMOV32 VR2, @Val ; VR2 gets the contents of Val
; <-- VMOV32 completes here (VR2 is valid)
; <-- VITBM2 completes here (VR0 is valid)
<istruction 2> ; Any instruction, can use VR2 and/or VR0
```

Example 5-4. Branch Metric CR 1/3 Calculation with Parallel Load

```
; VITBM3 || VMOV32 instruction: branch metrics calculation with parallel load
; VBITM3 is a 2p cycle operation (code rate = 1/3)
; VMOV32 is a 1 cycle operation
;
VITBM3 VR0, VR1, VR2 ; Load VR0 and VR1 with the 4 branch metrics
|| VMOV32 VR2, @Val ; VR2 gets the contents of Val
; <-- VMOV32 completes here (VR2 is valid)
; <-- VITBM3 completes here (VR0, VR1 are valid)
<istruction 2> ; Must not use VR0 or VR1. Can use VR2.
<istruction 3> ; Any instruction, can use VR2 and/or VR0
```

5.4.4 Invalid Delay Instructions

All VCU, FPU and fixed-point instructions can be used in VCU instruction delay slots as long as source and destination register conflicts are avoided. The C28x+VCU assembler will issue an error anytime you use an conflicting instruction within a delay slot. The following guidelines can be used to avoid these conflicts.

NOTE: Destination register conflicts in delay slots:

Any operation used for pipeline alignment delay must not use the same destination register as the instruction requiring the delay. See Example 5-5.

In Example 5-5 the VCMPY instruction uses VR2 and VR3 as its destination registers. The next instruction should not use VR2 or VR3 as a destination. Since the VMOV32 instruction uses the VR3 register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than VR2 for the VMOV32 instruction as shown in Example 5-6.
Example 5-5. Destination Register Conflict

```
; Invalid delay instruction.
; Both instructions use the same destination register (VR3)
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VMOV32 VR3, mem32         ; Invalid delay instruction
                        ; <-- VCMPY completes, VR3, VR2 are valid
```

Example 5-6. Destination Register Conflict Resolved

```
; Valid delay instruction
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VMOV32 VR7, mem32         ; Valid delay instruction
```

NOTE:  Instructions in delay slots cannot use the instruction’s destination register as a source register.

Any operation used for pipeline alignment delay must not use the destination register of the instruction requiring the delay as a source register as shown in Example 5-7. For parallel instructions, the current value of a register can be used in the parallel operation before it is overwritten as shown in Example 5-9.

In Example 5-7 the VCMPY instruction again uses VR3 and VR2 as its destination registers. The next instruction should not use VR3 or VR2 as its source since the VCMPY will take an additional cycle to complete. Since the VCADD instruction uses the VR2 as a source register a pipeline conflict will be issued by the assembler. The use of VR3 will also cause a pipeline conflict. This conflict can be resolved by using a register other than VR2 or VR3 or by inserting a non-conflicting instruction between the VCMPY and VCADD instructions. Since the VNEG does not use VR2 or VR3 this instruction can be moved before the VCADD as shown in Example 5-8.

Example 5-7. Destination/Source Register Conflict

```
; Invalid delay instruction.
; VCADD should not use VR2 or VR3 as a source operand
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VCADD VR5, VR4, VR3, VR2 ; Invalid delay instruction
VNEG VR0               ; <-- VCMPY completes, VR3, VR2 valid
```

Example 5-8. Destination/Source Register Conflict Resolved

```
; Valid delay instruction.
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VNEG VR0                ; Non conflicting instruction or NOP
VCADD VR5, VR4, VR3, VR2 ; <-- VCMPY completes, VR3, VR2 valid
```
It should be noted that a source register for the second operation within a parallel instruction can be the same as the destination register of the first operation. This is because the two operations are started at the same time. The second operation is not in the delay slot of the first operation. Consider Example 5-9 where the VCMPY uses VR3 and VR2 as its destination registers. The VMOV32 is the second operation in the instruction and can freely use VR3 or VR2 as a source register. In the example, the contents of VR3 before the multiply will be used by MOV32.

**Example 5-9. Parallel Instruction Destination/Source Exception**

```plaintext
; Valid parallel operation.
;
VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
|| VMOV32 mem32, VR3 ; <-- Uses VR3 before the VCMPY update
|| NOP ; <-- mem32 updated
; <-- Delay for VCMPY
; <-- VR2, VR3 updated
```

Likewise, the source register for the second operation within a parallel instruction can be the same as one of the source registers of the first operation. The VCMPY operation in Example 5-10 uses the VR0 register as one of its sources. This register is also updated by the VMOV32 instruction. The multiplication operation will use the value in VR0 before the VMOV32 updates it.

**Example 5-10. Parallel Instruction Destination/Source Exception**

```plaintext
; Valid parallel operation.
VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
|| VMOV32 VR0, mem32 ; <-- Uses VR3 before the VCMPY update
|| NOP ; <-- mem32 updated
; <-- Delay for VCMPY
; <-- VR2, VR3 updated
```

---

**NOTE:** Operations within parallel instructions cannot use the same destination register.

When two parallel operations have the same destination register, the result is invalid.

For example, see Example 5-11.

If both operations within a parallel instruction try to update the same destination register as shown in Example 5-11 the assembler will issue an error.

**Example 5-11. Invalid Destination Within a Parallel Instruction**

```plaintext
; Invalid parallel instruction. Both operations use VR3 as a destination register
;
VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
|| VMOV32 VR3, mem32 ; <-- Invalid
```
5.5 Instruction Set

This section describes the assembly language instructions of the VCU. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are independent from C28x and C28x+FPU instruction sets.

5.5.1 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Operands
- Opcode
- Description
- Exceptions
- Pipeline
- Examples
- See also

The example INSTRUCTION is shown to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information. VCU instructions follow the same format as the C28x; the source operand(s) are always on the right and the destination operand(s) are on the left.

The explanations for the syntax of the operands used in the instruction descriptions for the C28x VCU are given in Table 5-9.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FH</td>
<td>16-bit immediate (hex or float) value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FHiHex</td>
<td>16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value.</td>
</tr>
<tr>
<td>#32Fhex</td>
<td>32-bit immediate value that represents an IEEE 32-bit floating-point value.</td>
</tr>
<tr>
<td>#32F</td>
<td>Immediate float value represented in floating-point representation.</td>
</tr>
<tr>
<td>#0.0</td>
<td>Immediate zero.</td>
</tr>
<tr>
<td>#5-bit</td>
<td>5-bit immediate unsigned value.</td>
</tr>
<tr>
<td>addr</td>
<td>Opcode field indicating the addressing mode.</td>
</tr>
<tr>
<td>Im(X)</td>
<td>Imaginary part of the input X</td>
</tr>
<tr>
<td>Im(Y)</td>
<td>Imaginary part of the input Y</td>
</tr>
<tr>
<td>Im(Z)</td>
<td>Imaginary part of the output Z</td>
</tr>
<tr>
<td>Re(X)</td>
<td>Real part of the input X</td>
</tr>
<tr>
<td>Re(Y)</td>
<td>Real part of the input Y</td>
</tr>
<tr>
<td>Re(Z)</td>
<td>Real part of the output Z</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location.</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location.</td>
</tr>
<tr>
<td>VRa</td>
<td>VR0 - VR8 registers. Some instructions exclude VR8. Refer to the instruction description for details.</td>
</tr>
<tr>
<td>VR0H, VR1H...VR7H</td>
<td>VR0 - VR7 registers, high half.</td>
</tr>
<tr>
<td>VR0L, VR1L...VR7L</td>
<td>VR0 - VR7 registers, low half.</td>
</tr>
<tr>
<td>VTO, VT1</td>
<td>Transition bit register VT0 or VT1.</td>
</tr>
<tr>
<td>VSMn+1: VSMn</td>
<td>Pair of State Metric Registers (n = 0 : 62, n is even)</td>
</tr>
<tr>
<td>VRx.By</td>
<td>32 bit Aliased address space for each byte of the VRx registers (x = 0:7, y = 0:3)</td>
</tr>
</tbody>
</table>

Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).
Table 5-10. INSTRUCTION dest, source1, source2 Short Description

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dest1</td>
<td>Description for the 1st operand for the instruction</td>
</tr>
<tr>
<td>source1</td>
<td>Description for the 2nd operand for the instruction</td>
</tr>
<tr>
<td>source2</td>
<td>Description for the 3rd operand for the instruction</td>
</tr>
<tr>
<td>Opcode</td>
<td>This section shows the opcode for the instruction</td>
</tr>
<tr>
<td>Description</td>
<td>Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Any constraints on the operands or use of the instruction imposed by the processor are discussed.</td>
</tr>
<tr>
<td>Pipeline</td>
<td>This section describes the instruction in terms of pipeline cycles as described in Section 5.4.</td>
</tr>
<tr>
<td>Example</td>
<td>Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution. Some examples are code fragments while other examples are full tasks that assume the VCU is correctly configured and the main CPU has passed it data.</td>
</tr>
<tr>
<td>Operands</td>
<td>Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).</td>
</tr>
</tbody>
</table>
5.5.2 General Instructions

The instructions are listed alphabetically, preceded by a summary.

Table 5-11. General Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP RB — Pop the RB Register from the Stack</td>
<td>529</td>
</tr>
<tr>
<td>PUSH RB — Push the RB Register onto the Stack</td>
<td>531</td>
</tr>
<tr>
<td>RPTB label, loc16 — Repeat A Block of Code</td>
<td>533</td>
</tr>
<tr>
<td>RPTB label, #RC — Repeat a Block of Code</td>
<td>535</td>
</tr>
<tr>
<td>VCLR VRa — Clear General Purpose Register</td>
<td>537</td>
</tr>
<tr>
<td>VCLRALL — Clear All General Purpose and Transition Bit Registers</td>
<td>538</td>
</tr>
<tr>
<td>VCLRCPACK — Clears CPACK bit in the VSTATUS Register</td>
<td>539</td>
</tr>
<tr>
<td>VCLRCRCMSGFLIP — Clears CRCMSGFLIP bit in the VSTATUS Register</td>
<td>540</td>
</tr>
<tr>
<td>VCLROPACK — Clears OPACK bit in the VSTATUS Register</td>
<td>541</td>
</tr>
<tr>
<td>VCLROVFI — Clear Imaginary Overflow Flag</td>
<td>542</td>
</tr>
<tr>
<td>VCLROVF — Clear Real Overflow Flag</td>
<td>543</td>
</tr>
<tr>
<td>VMOV16 mem16, VRaH — Store General Purpose Register, High Half</td>
<td>544</td>
</tr>
<tr>
<td>VMOV16 mem16, VRaL — Store General Purpose Register, Low Half</td>
<td>545</td>
</tr>
<tr>
<td>VMOV16 VRaH, mem16 — Load General Purpose Register, High Half</td>
<td>546</td>
</tr>
<tr>
<td>VMOV16 VRaL, mem16 — Load General Purpose Register, Low Half</td>
<td>547</td>
</tr>
<tr>
<td>VMOV32 *(0:16bitAddr), loc32 — Move the contents of loc32 to Memory</td>
<td>548</td>
</tr>
<tr>
<td>VMOV32 loc32, *(0:16bitAddr) — Move 32-bit Value from Memory to loc32</td>
<td>549</td>
</tr>
<tr>
<td>VMOV32 mem32, VRa — Store General Purpose Register</td>
<td>550</td>
</tr>
<tr>
<td>VMOV32 mem32, VSTATUS — Store VCU Status Register</td>
<td>551</td>
</tr>
<tr>
<td>VMOV32 mem32, VTa — Store Transition Bit Register</td>
<td>552</td>
</tr>
<tr>
<td>VMOV32 VRa, mem32 — Load 32-bit General Purpose Register</td>
<td>553</td>
</tr>
<tr>
<td>VMOV32 VRb, VRa — Move 32-bit Register to Register</td>
<td>554</td>
</tr>
<tr>
<td>VMOV32 VSTATUS, mem32 — Load VCU Status Register</td>
<td>555</td>
</tr>
<tr>
<td>VMOV32 VTa, mem32 — Load 32-bit Transition Bit Register</td>
<td>556</td>
</tr>
<tr>
<td>VMOV32 VRa, mem32 — Load Register with Data Move</td>
<td>557</td>
</tr>
<tr>
<td>VMOVIX VRa, #16I — Load Upper Half of a General Purpose Register with 16-bit Immediate</td>
<td>558</td>
</tr>
<tr>
<td>VMOVZI VRa, #16I — Load General Purpose Register with Immediate</td>
<td>559</td>
</tr>
<tr>
<td>VMOVXI VRa, #16I — Load Low Half of a General Purpose Register with Immediate</td>
<td>560</td>
</tr>
<tr>
<td>VRNDOFF — Disable Rounding</td>
<td>561</td>
</tr>
<tr>
<td>VRNDONE — Enable Rounding</td>
<td>562</td>
</tr>
<tr>
<td>VSETCPACK — Set CPACK bit in the VSTATUS Register</td>
<td>563</td>
</tr>
<tr>
<td>VSETCRCMSGFLIP — Set CRCMSGFLIP bit in the VSTATUS Register</td>
<td>564</td>
</tr>
<tr>
<td>VSETOPACK — Set OPACK bit in the VSTATUS Register</td>
<td>565</td>
</tr>
<tr>
<td>VSETSHL #5-bit — Initialize the Left Shift Value</td>
<td>566</td>
</tr>
<tr>
<td>VSETSHR #5-bit — Initialize the Left Shift Value</td>
<td>567</td>
</tr>
<tr>
<td>VXORMOV32 VRa, mem32 — 32-bit Load and XOR From Memory</td>
<td>568</td>
</tr>
<tr>
<td>VXORMOV32 VRa, mem32 — 32-bit Load and XOR From Memory</td>
<td>569</td>
</tr>
</tbody>
</table>
POP RB — Pop the RB Register from the Stack

**Operands**

| RB       | repeat block register |

**Opcode**

LSW: 1111 1111 1111 0001

**Description**

Restore the RB register from stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```c
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...
    PUSH RB ; Save RB register only if a RPTB block is used in the ISR
...
    ...  
    RPTB _BlockEnd, AL ; Execute the block AL+1 times
...
...
_BlockEnd ; End of block to be repeated
...
...
    POP RB ; Restore RB register ...
IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```c
; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...
    PUSH RB ; Always save RB register
...
    CLRC INTM ; Enable interrupts only after saving RB
...
...
; ISR may or may not include a RPTB block
...
...
    SETC INTM ; Disable interrupts before restoring RB
...
    POP RB ; Always restore RB register
...
IRET ; RA = RAS, RAS = 0
```
See also

- PUSH RB
- RPTB label, loc16
- RPTB label, #RC
**PUSH RB**  
*Push the RB Register onto the Stack*

**Operands**

| RB     | repeat block register |

**Opcode**

LSW: 1111 1111 1111 0000

**Description**

Save the RB register on the stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```assembly
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...  
PUSH RB ; Save RB register only if a RPTB block is used in the ISR
...
...
RPTB _BlockEnd, AL ; Execute the block AL+1 times
...
...
...
_BlockEnd ; End of block to be repeated
...
...
POP RB ; Restore RB register ...
IRET ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```assembly
; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt: ; RAS = RA, RA = 0
...
PUSH RB ; Always save RB register
...
CLRC INTM ; Enable interrupts only after saving RB
...
...
; ISR may or may not include a RPTB block
...
...
SETC INTM ; Disable interrupts before restoring RB
...
POP RB ; Always restore RB register
...
IRET ; RA = RAS, RAS = 0
```
See also

POP RB
RPTB label, loc16
RPTB label, #RC
RPTB label, loc16

Repeat A Block of Code

Operands

| label | This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block. |
| loc16 | 16-bit location for the repeat count value. |

Opcode

LSW: 1011 0101 0bbb bbbb
MSW: 0000 0000 loc16

Description

Initialize repeat block loop, repeat count from [loc16]

Restrictions

- The maximum block size is ≤127 16-bit words.
- An even aligned block must be ≥ 9 16-bit words.
- An odd aligned block must be ≥ 8 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This instruction takes four cycles on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

Example

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

; Repeat Block of 8 Words (Interruptible)
;
; Note: This example makes use of floating-point (C28x+FPU) instructions
;
; find the largest element and put its address in XAR6
.align 2
NOP
RPTB _VECTOR_MAX_END, AR7
; Execute the block AR7+1 times
MOV ACC,XAR0 MOV32 R1H,*XAR0++ ; min size = 8, 9 words
MAXF32 R0H,R1H
MOVST0 NF,ZF
MOVST0 NF,ZF
MOV ACC,XAR0
MOV32 R1H,*XAR0++ ; max size = 127 words
MOVST0 NF,ZF
MOVST0 NF,ZF
MOV XAR6,ACC,LT
VECTOR_MAX_END: ; label indicates the end
; RA is cleared

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the
interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
; Interrupt:
; RAS = RA, RA = 0
... PUSH RB ; Save RB register only if a RPTB block is used in the ISR 
... 
RPTB _BlockEnd, AL ; Execute the block AL+1 times 
... 
... 
... 
_BlockEnd ; End of block to be repeated 
... 
... 
PUSH RB ; Restore RB register ...
IRET ; RA = RAS, RAS = 0

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

; Repeat Block within a Low-Priority Interrupt (Interruptible)
; Interrupt:
; RAS = RA, RA = 0
... PUSH RB ; Always save RB register
... CLRC INTM ; Enable interrupts only after saving RB
... 
... 
... 
; ISR may or may not include a RPTB block
... 
... 
SETC INTM ; Disable interrupts before restoring RB
... POP RB ; Always restore RB register
... IRET ; RA = RAS, RAS = 0

See also
POP RB
PUSH RB
RPTB label, #RC
**RPTB label, **RC **— Repeat a Block of Code**

### Operands

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>This label is used by the assembler to determine the end of the repeat block and to calculate <strong>RSIZE</strong>. This label should be placed immediately after the last instruction included in the repeat block.</td>
</tr>
<tr>
<td>#RC</td>
<td>16-bit immediate value for the repeat count.</td>
</tr>
</tbody>
</table>

### Opcode

- **LSW:** 1011 0101 1bbb bbbb
- **MSW:** cccc cccc cccc cccc

### Description

Repeat a block of code. The repeat count is specified as a immediate value.

### Restrictions

- The maximum block size is \( \leq 127 \) 16-bit words.
- An even aligned block must be \( \geq 9 \) 16-bit words.
- An odd aligned block must be \( \geq 8 \) 16-bit words.
- Interrupts must be disabled when saving or restoring the **RB** register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

### Flags

This instruction does not affect any flags in the VSTATUS register.

### Pipeline

This instruction takes one cycle on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

### Example

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

```c
; Repeat Block of 8 Words (Interruptible)
;
; Note: This example makes use of floating-point (C28x+FPU) instructions
;
; find the largest element and put its address in XAR6
;
; .align 2
; NOP
; RPTB _VECTOR_MAX_END, AR7
; Execute the block AR7+1 times
MOV ACC,XAR0 MOV32 R1H,*XAR0++ ; min size = 8, 9 words
MAXF32 R0H,R1H ; max size = 127 words
MOVST0 NF,ZF
MOVL XAR6,ACC,LT
VECTOR_MAX_END: ; label indicates the end
; RA is cleared
```

When an interrupt is taken the repeat active (RA) bit in the **RB** register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the **RB** register must be saved if a RPTB block is used within the
interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
; Interrupt: ; RAS = RA, RA = 0
... PUSH RB ; Save RB register only if a RPTB block is used in the ISR
... RPTB _BlockEnd, #5 ; Execute the block AL+1 times
... _BlockEnd ; End of block to be repeated
... POP RB ; Restore RB register ...
IRET ; RA = RAS, RAS = 0

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

; Repeat Block within a Low-Priority Interrupt (Interruptible)
; Interrupt: ; RAS = RA, RA = 0
... PUSH RB ; Always save RB register
... CLRC INTM ; Enable interrupts only after saving RB
... ... ; ISR may or may not include a RPTB block
... SETC INTM ; Disable interrupts before restoring RB
... POP RB ; Always restore RB register
... IRET ; RA = RAS, RAS = 0

See also
POP RB
PUSH RB
RPTB label, loc16
VCLEAR VRa — Clear General Purpose Register

Operands

| VRa | General purpose register: VR0, VR1... VR8 |

Opcode

LSW: 1110 0110 1111 1000
MSW: 0000 0000 0000 aaaa

Description

Clear the specified general purpose register.

\[ VRa = 0x00000000; \]

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

\[
\begin{align*}
; & \text{Code fragment from a viterbi traceback} \\
; & \text{For the first iteration the previous state metric must be} \\
; & \text{initialized to zero (VR0).} \\
; \quad \text{VCLEAR VR0} & \quad \text{Clear the VR0 register} \\
; \quad \text{MOVL XAR5,} & \quad \text{Point XAR5 to an array} \\
; \quad \text{For first stage} & \\
; \quad \text{VMOV32 VT0,} & \quad \text{Uses VR0 (which is zero)} \\
; \quad \text{VMOV32 VT1,} & \\
; \quad \text{VTRACE *XAR5++,VR0,VT0,VT1} & \\
; \quad \text{etc...} & \\
\end{align*}
\]

See also

VCLEARALL
VTCLEAR
VCLEARALL — Clear All General Purpose and Transition Bit Registers

**Operands**

none

**Opcode**

LSW: 1110 0110 1111 1001  
MSW: 0000 0000 0000 0000

**Description**

Clear all of the general purpose registers (VR0, VR1... VR8) and the transition bit registers (VT0 and VT1).

VR0 = 0x00000000;  
VR1 = 0x00000000;  
VR2 = 0x00000000;  
VR3 = 0x00000000;  
VR4 = 0x00000000;  
VR5 = 0x00000000;  
VR6 = 0x00000000;  
VR7 = 0x00000000;  
VR8 = 0x00000000;  
VT0 = 0x00000000;  
VT1 = 0x00000000;  
VSM0 = 0x00000000  
VSM1 = 0x00000000  
...  
VSM63 = 0x00000000

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
; Context save all VCU VRa and VTa registers

VMOV32 *SP++, VR0
VMOV32 *SP++, VR1
VMOV32 *SP++, VR2
VMOV32 *SP++, VR3
VMOV32 *SP++, VR4
VMOV32 *SP++, VR5
VMOV32 *SP++, VR6
VMOV32 *SP++, VR7
VMOV32 *SP++, VR8
VMOV32 *SP++, VT0
VMOV32 *SP++, VT1

; Clear VR0 - VR8, VT0 and VT1, VSM0 - VSM63

VCLEARALL

; etc...
```

**See also**

VCLEAR VRa  
VTCLEAR
VCLRCPACK
— Clears CPACK bit in the VSTATUS Register

Operands

none

Opcode

LSW: 1110 0101 0010 0010
MSW: 0000 0000 0000 0000

Description

Clears the CPACK bit in the VSTATUS register. This causes the VCU to process complex data, in complex math operations, in the VRx registers as follows: VRx[31:16] holds Real part, VRx[15:0] holds Imaginary part

Flags

This instruction clears the CPACK bit in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

; complex conjugate multiply | (jb + a) * (jd + c) = (ac+bd) + j(bc-ad)
VCLRCPACK ; cpack = 0 real part in high word
VMOV32 VR0, *XAR4++ ; load 1st complex input | jb + a
VMOV32 VR1, *XAR4++ ; load second complex input | jd + c
VCCMPY VR3, VR2, VR1, VR0

See also

VSETCPACK
**VCLRCRCMSGFLIP** — Clears CRCMSGFLIP bit in the VSTATUS Register

Operands
- none

Opcode
- LSW: 1110 0101 0010 1101
- MSW: 0000 0000 0000 0000

Description
Clear the CRCMSGFLIP bit in the VSTATUS register. This causes the VCU to process message bits starting from most-significant to least-significant for CRC computation. In this case, bytes loaded from memory are fed directly for CRC computation.

Flags
This instruction clears the CRCMSGFLIP bit in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
```c
; Clear the CRCMSGFLIP bit to have the CRC routine process the
; input message in big-endian format. The CRCMSGFLIP bit is
; cleared on reset
;
VCLRCRCMSGFLIP
LCR _CRC_run8Bit
```

See also
- VSETCRCMSGFLIP
VCLROPACK  Clears OPACK bit in the VSTATUS Register

Operands
none

Opcode
LSW: 1110 0101 0010 0101
MSW: 0000 0000 0000 0000

Description
Clear the OPACK bit in the VSTATUS register. This bit affects the packing order of the traceback output bits (using the VTRACE instructions). When the bit is set to 0 it forces the bits generated from the traceback operation to be loaded through the LSb of the destination register (or memory location) with the older bits being left shifted.

Flags
This instruction clears the OPACK bit in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example

See also
VSETOPACK
VCLROVFI — Clear Imaginary Overflow Flag

Operands
none

Opcode
LSW: 1110 0101 0000 1011

Description
Clear the real overflow flag in the VSTATUS register. To clear the real flag, use the VCLROVFR instruction. The imaginary flag bit can be set by instructions shown in Table 5-6. Refer to individual instruction descriptions for details.
VSTATUS[OVFR] = 0;

Flags
This instruction clears the OVFI flag.

Pipeline
This is a single-cycle instruction.

Example

See also
VCLROVFR
VRNDON
VSATFOFF
VSATON
<table>
<thead>
<tr>
<th>VCLROVFR</th>
<th><strong>Clear Real Overflow Flag</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operands</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0000 1010</td>
</tr>
</tbody>
</table>
| **Description** | Clear the real overflow flag in the VSTATUS register. To clear the imaginary flag, use the VCLROVFI instruction. The imaginary flag bit can be set by instructions shown in Table 5-6. Refer to individual instruction descriptions for details.  
VSTATUS[OVFR] = 0; |
| **Flags** | This instruction clears the OVFR flag. |
| **Pipeline** | This is a single-cycle instruction. |
| **See also** | VCLROVFI  
VRNDON  
VSATFOFF  
VSATON |
**VMOV16 mem16, VRaH — Store General Purpose Register, High Half**

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>Pointer to a 16-bit memory location. This will be the source for the VMOV16.</td>
</tr>
<tr>
<td>VRaH</td>
<td>High word of a general purpose register: VR0H, VR1H...VR8H.</td>
</tr>
</tbody>
</table>

**Opcode**

- **LSW**: 1110 0010 0001 1000
- **MSW**: 0001 aaaa mem16

**Description**

Store the upper 16-bits of the specified general purpose register into the 16-bit memory location.

\[ [\text{mem16}] = \text{VRa}[31:6]; \]

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**See also**

VMOV16 VRaH, mem16
VMOV16 mem16, VRaL  Store General Purpose Register, Low Half

Operands

<table>
<thead>
<tr>
<th>mem16</th>
<th>Pointer to a 16-bit memory location. This will be the destination of the VMOV16.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaL</td>
<td>Low word of a general purpose register: VR0L, VR1L...VR8L.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0001 1000  
MSW: 0000 aaaa mem16

Description

Store the low 16-bits of the specified general purpose register into the 16-bit memory location.

[mem16] = VRa[15:0];

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VMOV16 VRaL, mem16
VMOV16 VRaH, mem16  —  Load General Purpose Register, High Half

Operands

<table>
<thead>
<tr>
<th>VRaH</th>
<th>High word of a general purpose register: VR0H, VR1H,...,VR8H</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>Pointer to a 16-bit memory location. This will be the source for the VMOV16.</td>
</tr>
</tbody>
</table>

Opcode

- **LSW**: 1110 0010 1100 1001
- **MSW**: 0001 aaaa mem16

Description

Load the upper 16 bits of the specified general purpose register with the contents of memory pointed to by mem16.

VRa[31:16] = [mem16];

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

; 1st Iteration
VMOV32 VR4, *+XAR3[0] ; VR4H = m, VR4L=n Load m,n
VMOV16 VR0H, *+XAR5[0] ; VR0H = J, VR0L = I Init I, J
VMOV32 VR1, *+XAR3[4] ; VR1H = u, VR1L = a load u, a
VMOV32 VR6, VR0 ; Save current {J,I} in VR6
; etc.

See also

VMOV16 mem16, VRaH
VMOV16 VRaL, mem16  Load General Purpose Register, Low Half

Operands

<table>
<thead>
<tr>
<th>VRaL</th>
<th>Low word of a general purpose register: VR0L, VR1L,...VR8L</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>Pointer to a 16-bit memory location. This will be the source for the VMOV16.</td>
</tr>
</tbody>
</table>

Opcode

| LSW: 1110 0010 1100 1001 |
| MSW: 0000 aaaa mem16     |

Description

Load the lower 16 bits of the specified general purpose register with the contents of memory pointed to by mem16.

VRa[15:0] = [mem16];

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```c
; Loop will run 106 times for 212 inputs to decoder
; Code fragment from viterbi decoder
;
_LOOP:
;
; Calculate the branch metrics for code rate = 1/3
; Load VR0L, VR1L and VR2L with inputs
to the decoder from the array pointed to by XAR5
;
; VMOV16 VR0L, *XAR5++
VMOV16 VR1L, *XAR5++
VMOV16 VR2L, *XAR5++
;
; VR0L = BM0
; VR0H = BM1
; VR1L = BM2
; VR1H = BM3
; VR2L = pt_old[0]
; VR2H = pt_old[1]
;
VITBM3 VR0, VR1, VR2
VMOV32 VR2, *XAR3++
; etc...
```

See also

VMOV16 mem16, VRaL
VMOV32 *(0:16bitAddr), loc32 — Move the contents of loc32 to Memory

Operands

<table>
<thead>
<tr>
<th>*(0:16bitAddr)</th>
<th>Address of 32-bit Destination Location (VCU register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc32</td>
<td>Source Location (CPU register)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1101 loc32
MSW: IIII IIII IIII IIII

Description

Move the 32-bit value in loc32 to the memory location addressed by 0:16bitAddr. The EALLOW bit in the ST1 register is ignored by this operation.

\[ (0:16\text{bitAddr}) = [\text{loc32}] \]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a two-cycle instruction.

Example

```
; EALLOW ignored on write
; Four NOPs are needed after the operation so that the write to
; the VCU register takes effect before it is used in
; subsequent operations, for example
VMOV32 VRa, @ACC ; VRa = ACC
NOP ; Pipeline alignment
NOP ; Pipeline alignment
NOP ; Pipeline alignment
NOP ; Pipeline alignment
VMOV32 *XAR7++, VRa ; [*XAR] = VRa
```

See also

VMOV32 VRa, mem32
VMOV32 VRb, VRa
VMOV32 loc32, *(0:16bitAddr)
VMOV32 loc32, *(0:16bitAddr) — Move 32-bit Value from Memory to loc32

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc32</td>
<td>Destination Location (CPU register)</td>
</tr>
<tr>
<td>*(0:16bitAddr)</td>
<td>Address of 32-bit Source Value (VCU register)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1011 1111 loc32
MSW: IIII IIII IIII IIII

Description

Copy the 32-bit value referenced by 0:16bitAddr to the location indicated by loc32

[loc32] = [0:16bitAddr]

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is two-cycle instruction.

Example

; A single NOP is needed before the operation so as to read the
; correct VCU's VRx register value
    VMOV32 VRa,*XAR7++ ; VRa = [*XAR7]
    NOP                ; Pipeline alignment
    VMOV32 @ACC, VRa   ; ACC = VRa

; Two NOPs are needed before the operation so as to read the
; correct VCU's VSMx or VRx.By register value.
    VMOV32 VSM1: VSM0, *XAR7 ; VSM1:VSM0 = [*XAR7]
    NOP                ; Pipeline alignment
    NOP                ; Pipeline alignment
    VMOV32 @ACC, VSM0  ; AH:AL = VSM1:VSM0

See also

VMOV32 VRa, mem32
VMOV32 VRb, VRa
VMOV32 *(0:16bitAddr), loc32
VMOV32 mem32, VRa — Store General Purpose Register

Operands

| mem32 | Pointer to a 32-bit memory location. This will be the destination of the VMOV32. |
| VRa   | General purpose register VR0, VR1... VR8 |

Opcode

LSW: 1110 0010 0000 0100
MSW: 0000 aaaa mem32

Description

Store the 32-bit contents of the specified general purpose register into the memory location pointed to by mem32.

[mem32] = VRa;

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VRa, mem32
VMOV32 VTa, mem32
VMOV32 mem32, VSTATUS  Store VCU Status Register

Operands

| mem32  | Pointer to a 32-bit memory location. This will be the destination of the VMOV32. |
| VSTATUS | VCU status register. |

Opcode

| LSW: 1110 0010 0000 1101  |
| MSW: 0000 0000 mem32     |

Description

Store the VSTATUS register into the memory location pointed to by mem32.

\[[\text{mem32}] = \text{VSTATUS};\]

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

- VMOV32 mem32, VRa
- VMOV32 mem32, VTa
- VMOV32 VRa, mem32
- VMOV32 VSTATUS, mem32
- VMOV32 VTa, mem32
VMOV32 mem32, VTa — Store Transition Bit Register

Operands

| mem32 | pointer to a 32-bit memory location. This will be the destination of the VMOV32. |
| VTa   | Transition bits register VT0 or VT1 |

Opcode

LSW: 1110 0010 0000 0101
MSW: 0000 00tt mem32

Description

Store the 32-bits of the specified transition bits register into the memory location pointed to by mem32.

\[ [\text{mem32}] = \text{VTa}; \]

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

- `VMOV32 mem32, VRa`
- `VMOV32 mem32, VSTATUS`
- `VMOV32 VRa, mem32`
- `VMOV32 VSTATUS, mem32`
- `VMOV32 VTa, mem32`
VMOV32 VRa, mem32  

**Load 32-bit General Purpose Register**

**Operands**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRa</td>
<td>General purpose register VR0, VR1,...VR8</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW:</th>
<th>1110 0011 1111 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW:</td>
<td>0000 aaaa mem32</td>
</tr>
</tbody>
</table>

**Description**

Load the specified general purpose register with the 32-bit value in memory pointed to by mem32.

VRa = [mem32];

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

See also

- VMOV32 mem32, VRa
- VMOV32 mem32, VSTATUS
- VMOV32 mem32, VTa
- VMOV32 VSTATUS, mem32
- VMOV32 VTa, mem32
VMOV32 VRb, VRa — Move 32-bit Register to Register

Operands

| VRa | General purpose destination register VR0....VR8 |
| VRb | General purpose source register VR0...VR8   |

Opcode

LSW: 1110 0110 1111 0010
MSW: 0000 0010 bbbb aaaa

Description

Move a 32-bit value from one general purpose VCU register to another.

VRa = [mem32];

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

; Swap VR0 and VR1 using VR2 as temporary storage
; VMOV32 VR2, VR1
VMOV32 VR1, VR0
VMOV32 VR0, VR2

See also

VMOV32 mem32, VRa
VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VTa, mem32
VMOV32 VSTATUS, mem32  

**Load VCU Status Register**

**Operands**

<table>
<thead>
<tr>
<th>VSTATUS</th>
<th>VCU status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 1011 0000  
MSW: 0000 0000  mem32

**Description**

Load the VSTATUS register with the 32-bit value in memory pointed to by mem32.

\[ \text{VSTATUS} = [\text{mem32}] \]

**Flags**

This instruction modifies all bits within the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

See also

- VMOV32 mem32, VSTATUS
- VMOV32 mem32, VTa
- VMOV32 VRa, mem32
- VMOV32 VTa, mem32
VMOV32 VTa, mem32  **Load 32-bit Transition Bit Register**

**Operands**

<table>
<thead>
<tr>
<th>VTa</th>
<th>Transition bit register: VT0, VT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOV32.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0011 1111 0001  
MSW: 0000 00tt mem32

**Description**

Load the specified transition bit register with the 32-bit value in memory pointed to by mem32.

VTa = [mem32];

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**See also**

- VMOV32 mem32, VSTATUS
- VMOV32 mem32, VTa
- VMOV32 VRa, mem32
- VMOV32 VSTATUS, mem32
# VMOVD32 VRa, mem32

*Load Register with Data Move*

## Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register, VR0, VR1..., VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location. This will be the source of the VMOVD32.</td>
</tr>
</tbody>
</table>

## Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0010 0010 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 aaaa mem32</td>
</tr>
</tbody>
</table>

## Description

Load the specified general purpose register with the 32-bit value in memory pointed to by mem32. In addition, copy the next 32-bit value in memory to the location pointed to by mem32.

VRa = [mem32];
[mem32 + 2] = [mem32];

## Flags

This instruction does not modify any flags in the VSTATUS register.

## Pipeline

This is a single-cycle instruction.

## See also
## VMOVIX VRa, #16I

**Load Upper Half of a General Purpose Register with 16-bit Immediate**

### Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register, VR0, VR1... VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16I</td>
<td>16-bit immediate value</td>
</tr>
</tbody>
</table>

### Opcode

| LSW: 1110 0111 1110 IIII  |
| MSW: IIII IIII IIII aaaa |

### Description

Load the upper 16-bits of the specified general purpose register with an immediate value. Leave the upper 16-bits of the register unchanged.

VRa[15:0] = unchanged;
VRa[31:16] = #16I;

### Flags

This instruction does not modify any flags in the VSTATUS register.

### Pipeline

This is a single-cycle instruction.

### Example

See also:

- VMOVZI VRa, #16I
- VMOVXI VRa, #16I
VMOVZI VRa, #16I — Load General Purpose Register with Immediate

**Operands**
- **VRa**: General purpose register, VR0, VR1...VR8
- **#16I**: 16-bit immediate value

**Opcode**
- LSW: 1110 0111 1111 IIII
- MSW: IIII IIII IIII aaaa

**Description**
Load the lower 16-bits of the specified general purpose register with an immediate value. Clear the upper 16-bits of the register.

VRa[15:0] = #16I;
VRa[31:16] = 0x0000;

**Flags**
This instruction does not modify any flags in the VSTATUS register.

**Pipeline**
This is a single-cycle instruction.

**Example**
- VMOVIX VRa, #16I
- VMOVXI VRa, #16I

**See also**
- VMOVIX VRa, #16I
- VMOVXI VRa, #16I
### VMOVXI VRa, #16I — Load Low Half of a General Purpose Register with Immediate

<table>
<thead>
<tr>
<th>Operands</th>
<th>VRa</th>
<th>General purpose register, VR0 - VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#16I</td>
<td>16-bit immediate value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>LSW: 1110 0111 0111 IIII</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSW: IIII IIII IIII aaaa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>Load the lower 16-bits of the specified general purpose register with an immediate value. Leave the upper 16 bits unchanged.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VRa[15:0] = #16I;</td>
</tr>
<tr>
<td></td>
<td>VRa[31:16] = unchanged;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags</th>
<th>This instruction does not modify any flags in the VSTATUS register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline</td>
<td>This is a single-cycle instruction.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>See also</th>
</tr>
</thead>
</table>

VMOVIX VRa, #16I

VMOVZI VRa, #16I
VRNDOFF — Disable Rounding

Operands

none

Opcode

LSW: 1110 0101 0000 1001

Description

This instruction disables the rounding mode by clearing the RND bit in the VSTATUS register. When rounding is disabled, the result of the shift right operation for addition and subtraction operations will be truncated instead of rounded. The operations affected by rounding are shown in Table 5-6. Refer to the individual instruction descriptions for information on how rounding affects the operation. To enable rounding use the VRNDON instruction.

For more information on rounding, refer to Section 5.3.2.

VSTATUS[RND] = 0;

Flags

This instruction clears the RND bit in the VSTATUS register. It does not change any flags.

Pipeline

This is a single-cycle instruction.

Example

See also

VCLROVFI
VCLROVFR
VRNDON
VSATFOFF
VSATON
VRNDON — Enable Rounding

**Operands**
none

**Opcode**
LSW: 1110 0101 0000 1000

**Description**
This instruction enables the rounding mode by setting the RND bit in the VSTATUS register. When rounding is enabled, the result of the shift right operation for addition and subtraction operations will be rounded instead of being truncated. The operations affected by rounding are shown in Table 5-6. Refer to the individual instruction descriptions for information on how rounding effects the operation. To disable rounding use the VRNDOFF instruction.

For more information on rounding, refer to Section 5.3.2.

VSTATUS[RND] = 1;

**Flags**
This instruction sets the RND bit in the VSTATUS register. It does not change any flags.

**Pipeline**
This is a single-cycle instruction.

**Example**
VCLROVFI
VCLROVFR
VRNDOFF
VSATFOFF
VSATON

See also
VCLROVFI
VCLROVFR
VRNDOFF
VSATFOFF
VSATON
## VSATOFF  —  Disable Saturation

**Operands**

none

**Opcode**

LSW: 1110 0101 0000 0111

**Description**

This instruction disables the saturation mode by clearing the SAT bit in the VSTATUS register. When saturation is disabled, results of addition and subtraction are allowed to overflow or underflow. When saturation is enabled, results will instead be set to a maximum or minimum value instead of being allowed to overflow or underflow. To enable saturation use the VSATON instruction.

\[ \text{VSTATUS[SAT]} = 0 \]

**Flags**

This instruction clears the the SAT bit in the VSTATUS register. It does not change any flags.

**Pipeline**

This is a single-cycle instruction.

**Example**

See also VCLROVFI, VCLROVFR, VRNDOFF, VRNDON, VSATON
**VSATON — Enable Saturation**

**Operands**
none

**Opcode**
LSW: 1110 0101 0000 0110

**Description**
This instruction enables the saturation mode by setting the SAT bit in the VSTATUS register. When saturation is enabled, results of addition and subtraction are not allowed to overflow or underflow. Results will, instead, be set to a maximum or minimum value. To disable saturation use the **VSATOFF** instruction.

VSTATUS[SAT] = 1

**Flags**
This instruction sets the SAT bit in the VSTATUS register. It does not change any flags.

**Pipeline**
This is a single-cycle instruction.

**See also**
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATOFF
VSETCPACK

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0010 0001</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Set the CPACK bit in the VSTATUS register. This causes the VCU to process complex data, in complex math operations, in the VRx registers as follows: VRx[31:16] holds the Imaginary part, VRx[15:0] holds the Real part</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>This instruction sets the CPACK bit in the VSTATUS register.</td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>This is a single-cycle instruction.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>; complex conjugate multiply</td>
</tr>
<tr>
<td></td>
<td>VSETCPACK ; cpack = 1 imag part in low word</td>
</tr>
<tr>
<td></td>
<td>VMOV32 VR0, *XAR4++ ; load 1st complex input</td>
</tr>
<tr>
<td></td>
<td>VMOV32 VR1, *XAR4++ ; load second complex input</td>
</tr>
<tr>
<td></td>
<td>VCCMPY VR3, VR2, VR1, VR0</td>
</tr>
</tbody>
</table>

**See also**

VCLRCPACK
VSETCRCMSGFLIP — Set CRCMSGFLIP bit in the VSTATUS Register

**Operands**

none

**Opcode**

LSW: 1110 0101 0010 1100

**Description**

Set the CRCMSGFLIP bit in the VSTATUS register. This causes the VCU to process message bits starting from least-significant to most-significant for CRC computation. In this case, bytes loaded from memory are "flipped" and then fed for CRC computation.

**Flags**

This instruction sets the CRCMSGFLIP bit in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

```
; Set the CRCMSGFLIP bit, each word has all its bits reversed
; prior to the CRC being calculated
;
VSETCRCMSGFLIP
LCR  _CRC_run8Bit
VCLRCRCMSGFLIP
```

**See also**

VCLRCRCMSGFLIP
VSETOPACK  —  Set OPACK bit in the VSTATUS Register

Operands

none

Opcode

LSW: 1110 0101 0010 0011

Description

Set the OPACK bit in the VSTATUS register. This bit affects the packing order of the traceback output bits (using the instructions). When the bit is set to 1 it forces the bits generated from the traceback operation to be loaded through the MSb of the destination register (or memory location) with the older bits being right-shifted. This instruction sets the OPACK bit in the VSTATUS register.

Flags

This instruction sets the OPACK bit in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

VSETOPACK ; VSTATUS.OPACK = 1, start packing the decoded
; bits from trace back into VT1 starting from the
; MSb, this obviates the need to manually flip the
; result each time
; etc.

See also

VCLROPACK
VSETSHL #5-bit — Initialize the Left Shift Value

Operands

#5-bit
5-bit, unsigned, immediate value

Opcode
LSW: 1110 0101 110s ssss

Description
Load VSTATUS[SHIFTL] with an unsigned, 5-bit, immediate value. The left shift value specifies the number of bits an operand is shifted by. A value of zero indicates no shift will be performed. The left shift is used by the and VCDSUB16 and VCDADD16 operations. Refer to the description of these instructions for more information. To load the right shift value use the VSETSHR #5-bit instruction.

VSTATUS[VSHIFTL] = #5-bit

Flags
This instruction changes the VSHIFTL value in the VSTATUS register. It does not change any flags.

Pipeline
This is a single-cycle instruction.

Example

See also
VSETSHR #5-bit
## VSETSHR #5-bit

### Initialize the Left Shift Value

**Operands**

- #5-bit 5-bit, unsigned, immediate value

**Opcode**

LSW: 1110 0101 010s ssss

**Description**

Load VSTATUS[SHIFTR] with an unsigned, 5-bit, immediate value. The right shift value specifies the number of bits an operand is shifted by. A value of zero indicates no shift will be performed. The right shift is used by the VCADD, VCSUB, VCDADD16 and VCDSUB16 operations. It is also used by the addition portion of the VCMAC. Refer to the description of these instructions for more information.

VSTATUS[VSHIFTR] = #5-bit

**Flags**

This instruction changes the VSHIFTR value in the VSTATUS register. It does not change any flags.

**Pipeline**

This is a single-cycle instruction.

**Example**

See also VSETSHL #5-bit
VSWAP32 VRb, VRa — 32-bit Register Swap

Operands

| VRb  | General purpose register VR0...VR8 |
| VRab | General purpose register VR0...VR8 |

Opcode

LSW: 1110 0110 1111 0010
MSW: 0000 0011 bbbb aaaa

Description

Swap the contents of the 32-bit general purpose VCU registers VRa and VRb.

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```
; Swap VR0 and VR1 using VSWAP32 instruction
```

See also

VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VRa, mem32
VMOV32VRbVRa
VMOV32VTamem32
**VXORMOV32 VRa, mem32 — 32-bit Load and XOR From Memory**

**Operands**

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRa</td>
<td>General purpose register VR0...VR8</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0011 1111 0000  
MSW: 0000 aaaa MMMM MMMM

**Description**

XOR the contents of the VRa register with a long word from memory and store the result back into VRa

VRa = VRa ^ mem32

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

VXORMOV32 VR0, *+XAR4[0] ;VR0=VR0 ^ *XAR4[0]

**See also**
### 5.5.3 Arithmetic Math Instructions

The instructions are listed alphabetically, preceded by a summary.

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VASHL32 VRa &lt;&lt; #5-bit</td>
<td>573</td>
</tr>
<tr>
<td>VASHR32 VRa &gt;&gt; #5-bit</td>
<td>574</td>
</tr>
<tr>
<td>VBITFLIP VRa</td>
<td>575</td>
</tr>
<tr>
<td>VLSHL32 VRa &lt;&lt; #5-bit</td>
<td>576</td>
</tr>
<tr>
<td>VLSHR32 VRa &gt;&gt; #5-bit</td>
<td>577</td>
</tr>
<tr>
<td>VNEG VRa</td>
<td>578</td>
</tr>
</tbody>
</table>

---

*Table 5-12. Arithmetic Math Instructions*
VASHL32 VRa << #5-bit  —  Arithmetic Shift Left

### Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>VRa can be VR0 - VR7. VRa can not be VR8.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5-bit</td>
<td>5-bit unsigned immediate value</td>
</tr>
</tbody>
</table>

### Opcode

| LSW: 1110 0110 1111 0010 |
| MSW: 0000 0111 IIII Iaaa |

### Description

Arithmetic left shift of VRa

```c
If(VSTATUS[SAT] == 1){
    VRa = sat(VRa << #5-bit Immediate)
}else {
    VRa = VRa << #5-bit Immediate
}
```

### Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the 32-bit signed result after the shift left operation overflows

### Pipeline

This is a single-cycle instruction

### Example

```
VASHL32 VR4 << #16 ; VR4 := VR4 << 16
```

### See also

VASHR32 VRa>> #5-bit
**VASHR32 VRa >> #5-bit — Arithmetic Shift Right**

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
<th>Flags</th>
<th>Pipeline</th>
<th>Example</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRa</td>
<td>VRa can be VR0 - VR7. VRa can not be VR8.</td>
<td>This instruction does not affect any flags in the VSTATUS register</td>
<td>This is a single-cycle instruction</td>
<td>VASHR32 VR1 &gt;&gt; #16 ; VR1 := VR1 &gt;&gt; 16 (sign extended)</td>
<td>VASHL32 VRa#5-bit</td>
</tr>
<tr>
<td>#5-bit</td>
<td>5-bit unsigned immediate value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0110 1111 0010

MSW: 0000 1000 IIII Iaaa

**Description**

Arithmetic right shift of VRa

If(VSTATUS[RND] == 1){
    VRa = rnd(VRa >> #5-bit Immediate)
} else {
    VRa = VRa >> #5-bit Immediate
}

**Flags**
# VBITFLIP VRa — Bit Flip

## Operands

| VRa | General purpose register VR0...VR8 |

## Opcode

`LSW: 1010 0001 0010 aaaa`

## Description

Reverse the bit order of VRa register

VRa[31:0] = VRa[0:31]

## Flags

This instruction does not affect any flags in the VSTATUS register

## Pipeline

This is a single-cycle instruction

## Example

```
VBITFLIP VR1 ; VR1(31:0) := VR1(0:31)
```
VLSHL32 VRa << #5-bit — Logical Shift Left

VLSHL32 VRa << #5-bit Logical Shift Left

Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>VRa can be VR0 - VR7. VRa can not be VR8.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5-bit</td>
<td>5-bit unsigned immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1111 0010  
MSW: 0000 0101 IIII Iaaa

Description

Logical right shift of VRa  
VRa = VRa << #5-bit Immediate

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

VLSHL32 VR0 << #16 ; VR0 := VR0 << 16

See also

VLSHL32 VRa>> #5-bit
VLSHR32 VRa >> #5-bit — Logical Shift Right

Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>VRa can be VR0 - VR7. VRa can not be VR8.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5-bit</td>
<td>5-bit unsigned immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1111 0010
MSW: 0000 0110 IIII Iaaa

Description

Logical right shift of VRa

VRa = VRa >> #5-bit Immediate

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

VLSHR32 VR0 >> #16 ; VR0 := VR0 >> 16 (no sign extension)

See also

VLSHL32 VRa#5-bit
### VNEG VRa  
**Two’s Complement Negate**

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th>VRa can be VR0 - VR7. VRa can not be VR8.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Opcode</strong></th>
<th>LSW: 1110 0101 0001 aaaa</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th>Complex add operation.</th>
</tr>
</thead>
</table>

```c
// SAT is VSTATUS[SAT]
//
if (VRa == 0x80000000) {
  if(SAT == 1) {
    VRa = 0x7FFFFFFF;
  } else {
    VRa = 0x80000000;
  }
} else {
  VRa = - VRa
}
```

<table>
<thead>
<tr>
<th><strong>Flags</strong></th>
<th>This instruction modifies the following bits in the VSTATUS register:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• OVFR is set if the input to the operation is 0x80000000.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Pipeline</strong></th>
<th>This is a single-cycle instruction.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Example</strong></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>See also</strong></th>
<th>VCLROVFR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSATON</td>
</tr>
<tr>
<td></td>
<td>VSATOFF</td>
</tr>
</tbody>
</table>
### 5.5.4 Complex Math Instructions

The instructions are listed alphabetically, preceded by a summary.

**Table 5-13. Complex Math Instructions**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCADD VR5, VR4, VR3, VR2 — Complex 32 + 32 = 32 Addition</td>
<td>580</td>
</tr>
<tr>
<td>VCADD VR5, VR4, VR3, VR2</td>
<td></td>
</tr>
<tr>
<td>VCADD VR7, VR6, VR5, VR4 — Complex 32 + 32 - Addition</td>
<td>584</td>
</tr>
<tr>
<td>VCMAC VR5, VR4, VR3, VR2, VR1, VR0 — Complex Conjugate Multiply and Accumulate</td>
<td>586</td>
</tr>
<tr>
<td>VCMAC VR5, VR4, VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ — Complex Conjugate Multiply and Accumulate</td>
<td>590</td>
</tr>
<tr>
<td>VCCMPY VR3, VR2, VR1, VR0 — Complex Conjugate Multiply</td>
<td>593</td>
</tr>
<tr>
<td>VCCMPY VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCCMPY VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCCMACK VR5, VR4, VR3, VR2, VR1, VR0 — Complex Conjugate Multiply with Parallel Load</td>
<td>599</td>
</tr>
<tr>
<td>VCCON VRa — Complex Conjugate</td>
<td>601</td>
</tr>
<tr>
<td>VCDADD16 VR5, VR4, VR3, VR2 — Complex 16 + 32 = 16 Addition</td>
<td>602</td>
</tr>
<tr>
<td>VCDADD16 VR5, VR4, VR3, VR2</td>
<td></td>
</tr>
<tr>
<td>VCDSUB16 VR6, VR4, VR3, VR2 — Complex 16-32 = 16 Subtract</td>
<td>609</td>
</tr>
<tr>
<td>VCDSUB16 VR6, VR4, VR3, VR2</td>
<td></td>
</tr>
<tr>
<td>VCFLIP VRa — Swap Upper and Lower Half of VCU Register</td>
<td>616</td>
</tr>
<tr>
<td>VCMAC VR5, VR4, VR3, VR2, VR1, VR0 — Complex Multiply and Accumulate</td>
<td>617</td>
</tr>
<tr>
<td>VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ — Complex Multiply and Accumulate</td>
<td>619</td>
</tr>
<tr>
<td>VCMAC VR5, VR4, VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCMAG VRb, VRa — Magnitude of a Complex Number</td>
<td>625</td>
</tr>
<tr>
<td>VCMY VR3, VR2, VR1, VR0 — Complex Multiply</td>
<td>626</td>
</tr>
<tr>
<td>VCMY VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCMY VR3, VR2, VR1, VR0</td>
<td></td>
</tr>
<tr>
<td>VCSHL16 VRa &lt;&lt; #4-bit — Complex Shift Left</td>
<td>632</td>
</tr>
<tr>
<td>VCSHR16 VRa &gt;&gt; #4-bit — Complex Shift Right</td>
<td>633</td>
</tr>
<tr>
<td>VCSUB VR5, VR4, VR3, VR2 — Complex 32 - 32 = 32 Subtraction</td>
<td>634</td>
</tr>
<tr>
<td>VCSUB VR5, VR4, VR3, VR2</td>
<td></td>
</tr>
</tbody>
</table>
VCADD VR5, VR4, VR3, VR2 — Complex 32 + 32 = 32 Addition

Operands
Before the operation, the inputs should be loaded into registers as shown below. Each operand for this instruction includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| VR5             | 32-bit integer representing the real part of the result:  
|                 | Re(Z) = Re(X) + (Re(Y) >> SHIFTR) |
| VR4             | 32-bit integer representing the imaginary part of the result:  
|                 | Im(Z) = Im(X) + (Im(Y) >> SHIFTR) |

Opcode
LSW: 1110 0101 0000 0010

Description
Complex 32 + 32 = 32-bit addition operation.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFR] bits before the addition. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 3.4.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
//
// X: VR5 = Re(X) VR4 = Im(X)
// Y: VR3 = Re(Y) VR2 = Im(Y)
//
// Calculate Z = X + Y
//
// if (RND == 1)
// {
//     VR5 = VR5 + round(VR3 >> SHIFTR); // Re(Z)
//     VR4 = VR4 + round(VR2 >> SHIFTR); // Im(Z)
// }
// else
// {
//     VR5 = VR5 + (VR3 >> SHIFTR);       // Re(Z)
//     VR4 = VR4 + (VR2 >> SHIFTR);       // Im(Z)
// }
// if (SAT == 1)
// {
//    sat32(VR5);
//    sat32(VR4);
// }
```

Flags
This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR5 computation (real part) overflows or underflows.
- OVFI is set if the VR4 computation (imaginary part) overflows or underflows.

Pipeline
This is a single-cycle instruction.
Example

See also

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCLROVF1
VCLROVFR
VRNDOFF
VRN DON
VSATON
VSATOFF
VSETSHR #5-bit
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex 32+32 = 32 Add with Parallel Load

Operands

Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the result:</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = Re(X) + (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the result:</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = Im(X) + (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VRa</td>
<td>contents of the memory pointed to by [mem32]. VRa can not be VR5, VR4 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 1000
MSW: 0000 aaaa mem32

Description

Complex 32 + 32 = 32-bit addition operation with parallel register load.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFTR] bits before the addition. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 5.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

In parallel with the addition, VRa is loaded with the contents of memory pointed to by mem32.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]

// VR5 = Re(X) VR4 = Im(X)
// VR3 = Re(Y) VR2 = Im(Y)
// Z = X + Y

if (RND == 1)
{
    VR5 = VR5 + round(VR3 >> SHIFTR); // Re(Z)
    VR4 = VR4 + round(VR2 >> SHIFTR); // Im(Z)
}
else
{
    VR5 = VR5 + (VR3 >> SHIFTR); // Re(Z)
    VR4 = VR4 + (VR2 >> SHIFTR); // Im(Z)
}
if (SAT == 1)
{
    sat32(VR5);
    sat32(VR4);
}
VRa = [mem32];
```
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex 32+32 = 32 Add with Parallel Load

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR5 computation (real part) overflows.
- OVFI is set if the VR4 computation (imaginary part) overflows.

Pipeline
Both operations complete in a single cycle (1/1 cycles).

Example

See also
VCADD VR7, VR6, VR5, VR4
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR #5-bit
VCADD VR7, VR6, VR5, VR4  Complex 32 + 32 = 32- Addition

Operands
Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR6</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR7 and VR6 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR6</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) + (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR7</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) + (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0010 1010

Description
Complex 32 + 32 = 32-bit addition operation.

The second input operand (stored in VR5 and VR4) is shifted right by VSTATUS[SHIFTR] bits before the addition. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 5.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```c
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
//
// VR5 = Re(X) VR4 = Im(X)
// VR3 = Re(Y) VR2 = Im(Y)
//
// Z = X + Y
//
// if (RND == 1)
// {
//     VR7 = VR7 + round(VR5 >> SHIFTR); // Re(Z)
//     VR6 = VR6 + round(VR4 >> SHIFTR); // Im(Z)
// }
// else
// {
//     VR7 = VR5 + (VR5 >> SHIFTR); // Re(Z)
//     VR6 = VR4 + (VR4 >> SHIFTR); // Im(Z)
// }
// if (SAT == 1)
// {
//     sat32(VR7);
//     sat32(VR6);
// }
```

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR7 computation (real part) overflows.
- OVFI is set if the VR6 computation (imaginary part) overflows.

Pipeline
This is a single-cycle instruction.
Example

See also

VCADD VR5, VR4, VR3, VR2
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR #5-bit
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 — Complex Conjugate Multiply and Accumulate

Operands

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>VR4</td>
<td>Imaginary part of the accumulation</td>
</tr>
<tr>
<td>VR5</td>
<td>Real part of the accumulation</td>
</tr>
</tbody>
</table>

(1) The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.

Opcode

LSW: 1110 0101 0000 1111

Description

Complex Conjugate Multiply Operation

// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
//
// VR0 = X + jX: VR0[31:16] = X, VR0[15:0] = jX
//
// Perform add
//
// if (RND == 1)
// {
//   VR5 = VR5 + round(VR3 >> SHIFTR);
//   VR4 = VR4 + round(VR2 >> SHIFTR);
// }
// else
// {
//   VR5 = VR5 + (VR3 >> SHIFTR);
//   VR4 = VR4 + (VR2 >> SHIFTR);
// }
//
// Perform multiply (X + jX) * (Y - jY)
//
// If(VSTATUS[CPACK] == 0){
//   VR3 = VR0H * VR1H + VR0L * VR1L; Real result
//   VR2 = VR0H * VR1L - VR0L * VR1H; Imaginary result
// }
// else
// {
//   VR3 = VR0L * VR1L + VR0H * VR1H; Real result
//   VR2 = VR0L * VR1H - VR0H * VR1L; Imaginary result
// }
// if(SAT == 1)
// {
//   sat32(VR3);
//   sat32(VR2);
// }

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline

This is a 2p-cycle instruction.

See also

VCLROVFI
VCLROVFR
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VSATON
VSATOFF
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32  
**Complex Conjugate Multiply and Accumulate with Parallel Load**

### Operands

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>VR4</td>
<td>Imaginary part of the accumulation</td>
</tr>
<tr>
<td>VR5</td>
<td>Real part of the accumulation</td>
</tr>
<tr>
<td>VRa</td>
<td>Contents of the memory pointed to by mem32. VRa cannot be VR5, VR4 or VR8</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Note: The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.

### Opcode

LSW: 1110 0011 1111 0111  
MSW: 0001 aaaa mem32

### Description

Complex Conjugate Multiply Operation with parallel load.

```c
// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
// // VR0 = X + jX: VR0[31:16] = X, VR0[15:0] = jX
// // Perform add
// if (RND == 1)
// {  
//     VR5 = VR5 + round(VR3 >> SHIFTR);
//     VR4 = VR4 + round(VR2 >> SHIFTR);
// } else
// {
//     VR5 = VR5 + (VR3 >> SHIFTR);
//     VR4 = VR4 + (VR2 >> SHIFTR);
// }
// // Perform multiply (X + jX) * (Y - jY)
// if(VSTATUS[CPACK] == 0){
//     VR3 = VR0H * VR1H + VR0L * VR1L; Real result
//     VR2 = VR0H * VR1L - VR0L * VR1H; Imaginary result
// } else
// {
//     VR3 = VR0L * VR1L + VR0H * VR1H; Real result
//     VR2 = VR0L * VR1H - VR0H * VR1L; Imaginary result
// } if(SAT == 1)
// {  
//     sat32(VR3);
//     sat32(VR2);
// }
VRa = [mem32];
```
**Flags**

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

**Pipeline**

This is a 2p-cycle instruction.

**See also**

VCLROVFI
VCLROVFR
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VSATON
VSATOFF
VCCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++  
Complex Conjugate Multiply and Accumulate

Operands
The VMAC alternates which registers are used between each cycle. For odd cycles (1, 3, 5, and so on) the following registers are used:

<table>
<thead>
<tr>
<th>Odd Cycle Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>Previous real-part total accumulation: Re(odd_sum)</td>
</tr>
<tr>
<td>VR4</td>
<td>Previous imaginary-part total accumulation: Im(odd-sum)</td>
</tr>
<tr>
<td>VR1</td>
<td>Previous real result from the multiply: Re(odd-mpy)</td>
</tr>
<tr>
<td>VR0</td>
<td>Previous imaginary result from the multiply Im(odd-mpy)</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Pointer to a 32-bit memory location representing the first input to the multiply</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 0)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Re(X)</td>
</tr>
<tr>
<td></td>
<td>[mem32][15:0] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 1)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>mem32[15:0] = Re(X)</td>
</tr>
<tr>
<td>XAR7</td>
<td>Pointer to a 32-bit memory location representing the second input to the multiply</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 0)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[32:16] = Re(X)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[15:0] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 1)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[32:16] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>*XAR7 [15:0] = Re(X)</td>
</tr>
</tbody>
</table>

The result from the odd cycle is stored as shown below:

<table>
<thead>
<tr>
<th>Odd Cycle Output</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit real part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Re(odd_sum) = Re(odd_sum) + Re(odd_mpy)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit imaginary part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Im(odd_sum) = Im(odd_sum) + Im(odd_mpy)</td>
</tr>
<tr>
<td>VR1</td>
<td>32-bit real result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)</td>
</tr>
<tr>
<td>VR0</td>
<td>32-bit imaginary result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = Re(X)*Im(Y) - Re(Y)*Im(X)</td>
</tr>
</tbody>
</table>

For even cycles (2, 4, 6, and so on) the following registers are used:

<table>
<thead>
<tr>
<th>Even Cycle Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>Previous real-part total accumulation: Re(even_sum)</td>
</tr>
<tr>
<td>VR6</td>
<td>Previous imaginary-part total accumulation: Im(even-sum)</td>
</tr>
<tr>
<td>VR3</td>
<td>Previous real result from the multiply: Re(even-mpy)</td>
</tr>
<tr>
<td>VR2</td>
<td>Previous imaginary result from the multiply Im(even-mpy)</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Pointer to a 32-bit memory location representing the first input to the multiply</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 0)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Re(X)</td>
</tr>
<tr>
<td></td>
<td>[mem32][15:0] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 1)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Im(X)</td>
</tr>
</tbody>
</table>
Even Cycle Input  Value

mem32[15:0] = Re(X)

XAR7  Pointer to a 32-bit memory location representing the second input to the multiply
If(VSTATUS[CPACK] == 0)
  *XAR7[32:16] = Re(X)
  *XAR7[15:0] = Im(X)
If(VSTATUS[CPACK] == 1)
  *XAR7[32:16] = Im(X)
  *XAR7[15:0] = Re(X)

The result from even cycles is stored as shown below:

Even Cycle Output  Value

VR7  32-bit real part of the total accumulation
     Re(even_sum) = Re(even_sum) + Re(even_mpy)
VR6  32-bit imaginary part of the total accumulation
     Im(even_sum) = Im(even_sum) + Im(even_mpy)
VR3  32-bit real result from the multiplication:
     Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
VR2  32-bit imaginary result from the multiplication:
     Im(Z) = Re(X)*Im(Y) - Re(Y)*Im(X)

Opcode

LSW: 1110 0010 0101 0001
MSW: 0010 1111 mem32

Description  Perform a repeated complex conjugate multiply and accumulate operation. This
instruction must be used with the single repeat instruction (RPT ||). The destination
of the accumulate will alternate between VR7/VR6 and VR5/VR4 on each cycle.

// Cycle 1:
// // Perform accumulate
// if(RND == 1)
// {
//   VR5 = VR5 + round(VR1 >> SHIFTR)
//   VR4 = VR4 + round(VR0 >> SHIFTR)
// }
// else
// {
//   VR5 = VR5 + (VR1 >> SHIFTR)
//   VR4 = VR4 + (VR0 >> SHIFTR)
// }
// // X and Y array element 0
// // VR1 = Re(X)*Re(Y) + Im(X)*Im(Y)
// VR0 = Re(X)*Im(Y) - Re(Y)*Im(X)
// // Cycle 2:
// // // Perform accumulate
// if(RND == 1)
// {
//   VR7 = VR7 + round(VR3 >> SHIFTR)
//   VR6 = VR6 + round(VR2 >> SHIFTR)
// }
else
{
VR7 = VR7 + (VR3 >> SHIFTR)
VR6 = VR6 + (VR2 >> SHIFTR)
}

// X and Y array element 1
//
VR3 = Re(X)*Re(Y) + Im(X)*Im(Y)
VR2 = Re(X)*Im(Y) - Re(Y)*Im(X)

// Cycle 3:
//
// Perform accumulate
//
if(RND == 1)
{
VR5 = VR5 + round(VR1 >> SHIFTR)
VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
VR5 = VR5 + (VR1 >> SHIFTR)
VR4 = VR4 + (VR0 >> SHIFTR)
}

// X and Y array element 2
//
VR1 = Re(X)*Re(Y) + Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) - Re(Y)*Im(X)

etc...

Restrictions
VR0, VR1, VR2, and VR3 will be used as temporary storage by this instruction.

Flags
The VSTATUS register flags are modified as follows:
• OVFR is set in the case of an overflow or underflow of the addition or subtraction operations.
• OVFI is set in the case an overflow or underflow of the imaginary part of the addition or subtraction operations.

Pipeline
The VCCMAC takes 2p + N cycles where N is the number of times the instruction is repeated. This instruction has the following pipeline restrictions:

`<instruction1>` ; No restriction
`<instruction2>` ; Cannot be a 2p instruction that writes
; to VR0, VR1...VR7 registers
RPT #(N-1) ; Execute N times, where N is even
|| VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
`<instruction3>` ; No restrictions.
; Can read VR0, VR1... VR8

See also
VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++
VCCMPY VR3, VR2, VR1, VR0  Complex Conjugate Multiply

Operands
Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

Opcode
LSW: 1110 0101 0000 1110

Description
Complex Conjugate 16 x 16 = 32-bit multiply operation.

If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow. The following operation is carried out:

```
if(VSTATUS[CPACK] == 0){
    VR3 = VR0H * VR1H + VR0L * VR1L; //Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0H * VR1L - VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
} else{
    VR3 = VR0L * VR1L + VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0L * VR1H - VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
}
```

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline
This is a 2p-cycle instruction. The instruction following this one should not use VR3 or VR2.

```
VCLRCPACK ; cpack = 0 real part in high word
VMOV32 VR0, *XAR4++ ; load 1st complex input | jb + a
VMOV32 VR1, *XAR4++ ; load second complex input | sd + c
VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply
; (jb + a)*(jd + c)=(ac+bd)+j(bc-ad)
NOP
VMOV32 *XAR5++, VR3 ; store real part first
VMOV32 *XAR5++, VR2 ; store imag part next
VSETCPACK ; cpack = 1 imag part in low word
VMOV32 VR0, *XAR4++ ; load 1st complex input | a + jb
VMOV32 VR1, *XAR4++ ; load second complex input | c + jd
VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply
; (a + jb)*(c + jd)=(ac+bd)+j(bc-ad)
NOP
VMOV32 *XAR5++, VR3 ; store real part first
VMOV32 *XAR5++, VR2 ; store imag part next
```

Example

See also
VCLROVF
VCLROVFI
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCCMPY VR3, VR2, VR1, VR0 — Complex Conjugate Multiply

VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSETCPACK
VCLRCPACK
VSATON
VSATOFF
VCCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa — Complex Conjugate Multiply with Parallel Store

### Operands
Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VRa</td>
<td>Value to be stored</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

### Opcode
LSW: 1110 0011 0000 0111
MSW: 0001 aaaa mem32

### Description
Complex Conjugate 16 x 16 = 32-bit multiply operation.

If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow. The following operation is carried out:

```
if(VSTATUS[CPACK] == 0){
    VR3 = VR0H * VR1H + VR0L * VR1L; //Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0H * VR1L - VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
} else{
    VR3 = VR0L * VR1L + VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0L * VR1H - VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
}
```

[mem32] = VRa;

### Flags
This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

### Pipeline
This is a 2p/1-cycle instruction. The multiply operation takes 2p cycles and the VMOV operation completes in a single cycle. The instruction following this one should not use VR3 or VR2.

### Example

```
VCLRCAPACK ; cpack = 0 real part in high word
VMOV32 VR0, *XAR4++; ; load 1st complex input | jb + a
VMOV32 VR1, *XAR4++; ; load second complex input | jd + c
VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply|
  | VMOV32 VR0, *XAR4++; ; (jb + a)*(jd + c)=(ac+bd)+j(bc-ad)
  | ; load 1st complex input | a + jb
  | NOP  ; for next VCCMPY instr |
  | VMOV32 *XAR5++, VR3 ; store real part first
  | VSETCPACK ; cpack = 1 imag part in low word
  | VMOV32 VR1, *XAR4++; ; load second complex input | c + jd
  | VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply|
  |   | VMOV32 *XAR5++, VR2 ; (a + jb)*(c + jd)=(ac+bd)+j(bc-ad)
  |   | ; store imag part of first |
  |   | NOP  ; VCCMPY instruction |
  |   | VMOV32 *XAR5++, VR3 ; store real part first
  |   | VMOV32 *XAR5++, VR2 ; store imag part next
  | VCLRCPACK |
```
VCCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa — Complex Conjugate Multiply with Parallel Store

See also

VCLROVF
VCLROVFR
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCCMAC VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSETCPACK
VCLRCPACK
VSATON
VSATOFF
VCCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 — Complex Conjugate Multiply with Parallel Load

Operands

Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VRa</td>
<td>32-bit value pointed to by mem32. VRa can not be VR2, VR3 or VR8.</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

Opcode

| LSW: | 1110 0011 1111 0110 |
| MSW: | 0001 aaaa mem32 |

Description

Complex Conjugate 16 x 16 = 32-bit multiply operation.

If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow. The following operation is carried out:

```c
if(VSTATUS[CPACK] == 0){
    VR3 = VR0H * VR1H + VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0H * VR1L - VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
} else{
    VR3 = VR0L * VR1L + VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0L * VR1H - VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
}
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline

This is a 2p/1-cycle instruction. The multiply operation takes 2p cycles and the VMOV operation completes in a single cycle. The instruction following this one should not use VR3 or VR2.

Example

```
VCLRCPACK ; cpack = 0 real part in high word
VMOV32 VR0, *XAR4++; ; load 1st complex input | jb + a
VMOV32 VR1, *XAR4++; ; load second complex input | jd + c
VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply |
VMOV32 VR0, *XAR4++; ; (jb + a)*(jd + c)=(ac+bd)+j(bc-ad) |
               ; load 1st complex input | a + jb |
               ; for next VCCMPY instr |
NOP |
VMOV32 *XAR5++, VR3 ; store real part first
VSETCPACK ; cpack = 0 imag part in low word
VMOV32 VR1, *XAR4++; ; load second complex input | c + jd |
VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply |
VMOV32 *XAR5++, VR2 ; (a + jb)*(c + jd)=(ac+bd)+j(bc-ad) |
               ; store imag part of first |
               ; VCCMPY instruction |
NOP |
VMOV32 *XAR5++, VR3 ; store real part first
VMOV32 *XAR5++, VR2 ; store imag part next
VCLRCPACK
```
VCCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 — Complex Conjugate Multiply with Parallel Load

See also

VCLROVFI
VCLROVFR
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VSETCPACK
VCLRCPACK
VSATON
VSATOFF
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 — Complex Conjugate Multiply with Parallel Load

Operands

Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VRa</td>
<td>32-bit value pointed to by mem32. VRa can not be VR2, VR3 or VR8.</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

Opcode

LSW: 1110 0101 0000 1111

Description

Complex Conjugate 16 x 16 = 32-bit multiply operation.

If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow. The following operation is carried out:

if(VSTATUS[CPACK] == 0){
    VR3 = VR0H * VR1H + VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0H * VR1L - VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
}else{
    VR3 = VR0L * VR1L + VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y)
    VR2 = VR0L * VR1H - VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) - Im(X)*Re(Y)
}

VRa = [mem32];

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline

This is a 2p/1-cycle instruction. The multiply operation takes 2p cycles and the VMOV operation completes in a single cycle. The instruction following this one should not use VR3 or VR2.

Example

VCLRCPACK ; cpack – 0 real part in high word
VMOV32     VR0, *XAR4++ ; load 1st complex input | jb + a
VMOV32     VR1, *XAR4++ ; load second complex input | jd + c
VCCMPY     VR3, VR2, VR1, VR0 ; complex conjugate multiply|
    | VMOV32     VR0, *XAR4++ ; (jb + a)*(jd + c)=(ac+bd)+j(bc-ad)
    | ; load 1st complex input | a + jb
    | NOP ; for next VCCMPY instr |
    | VMOV32     *XAR5++, VR3 ; store real part first
    | VSETCPACK ; cpack – 1 imag part in low word
    | VMOV32     VR1, *XAR4++ ; load second complex input | c + jd
    | VCCMPY     VR3, VR2, VR1, VR0 ; complex conjugate multiply|
    |    | VMOV32     *XAR5++, VR2 ; (a + jb)*(c + jd)=(ac+bd)+j(bc-ad)
    |    | ; store imag part of first |
    |    | NOP ; VCCMPY instruction |
    | VMOV32     *XAR5++, VR3 ; store real part next
    | VMOV32     *XAR5++, VR2 ; store imag part next
VCLRCPACK

See also

VCLROVFI
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 — *Complex Conjugate Multiply with Parallel Load*

VCLROVFR
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSETCPACK
VCLRCPACK
VSATON
VSATOFF
## VCCON VRa — Complex Conjugate

### Operands

| VRa | General purpose register: VR0, VR1,...VR7. Cannot be VR8. |

### Opcode

LSW: 1110 0001 0001 aaaa

### Description

```c
if(VSTATUS[CPACK] == 0){
    if(VSTATUS[SAT] == 1){
        VRaL = sat(- VRaL)
    }else {
        VRaL = - VRaL
    }
}else {
    if(VSTATUS[SAT] == 1){
        VRaH = sat(- VRaH)
    }else {
        VRaH = - VRaH
    }
}
```

### Flags

This instruction modifies the following bits in the VSTATUS register:
- **OVFI** is set in the case an overflow or underflow of the imaginary part of the conjugate operation.

### Pipeline

This is a single-cycle instruction.

### Example

```
VCCON VR1 ; VR1 := VR1^*
```

### See also
VCDADD16 VR5, VR4, VR3, VR2  —  Complex 16 + 32 = 16 Addition

Operands
Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if(VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if(VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) + (Re(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) + (Im(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VR5L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) + (Im(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) + (Re(Y)) &gt;&gt; SHIFTR</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0000 0100

Description
Complex 16 + 32 = 16-bit operation. This operation is useful for algorithms similar to a complex FFT. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the addition is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 3.4.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

// RND is VSTATUS[RND]  
// SAT is VSTATUS[SAT]    
// SHIFTR is VSTATUS[SHIFTR] 
// SHIFTL is VSTATUS[SHIFTL] 
// VSTATUS[CPACK] = 0 
// VR4H = Re(X) 16-bit     
// VR4L = Im(X) 16-bit     
// VR3 = Re(Y) 32-bit     
// VR2 = Im(Y) 32-bit
// Calculate Z = X + Y
//

temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
temp2 = sign_extend(VR4L); // 32-bit extended Im(X)

/*
temp1 = (temp1 << SHIFTL) + VR3; // Re(Z) intermediate
temp2 = (temp2 << SHIFTL) + VR2; // Im(Z) intermediate
*/
if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
    temp2 = round(temp2 >> SHIFTR);
} else
{
    temp1 = truncate(temp1 >> SHIFTR);
    temp2 = truncate(temp2 >> SHIFTR);
}
if (SAT == 1)
{
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
} else
{
    VR5H = temp1[15:0];
    VR5L = temp2[15:0];
}

Flags
This instruction modifies the following bits in the VSTATUS register:
• OVFR is set if the real-part computation (VR5H) overflows or underflows.
• OVFI is set if the imaginary-part computation (VR5L) overflows or underflows.

Pipeline
This is a single-cycle instruction.

Example

; Example: Z = X + Y
; X = 4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 + 12j (32-bit real + 32-bit imaginary)
;
; Real:
; temp1 = 0x00000004 + 0x0000000D = 0x00000011
; VR5H = temp1[15:0] = 0x0011 = 17
; Imaginary:
; temp2 = 0x00000003 + 0x0000000C = 0x0000000F
; VR5L = temp2[15:0] = 0x000F = 15
;
VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR #0 ; VSTATUS[SHIFTR] = 0
VSETSHL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13
VMOVXI VR2, #12 ; VR2 = Im(Y) = 12
VMOVXI VR4, #3
VMOVIX VR4, #4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x0011000F = 17 + 15j

The next example illustrates the operation with a right shift value defined.

; Example: Z = X + Y with Right Shift
; X = 4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 + 12j (32-bit real + 32-bit imaginary)

; Real:
; temp1 = (0x00000004 + 0x0000000D) >> 1
; temp1 = (0x00000001) >> 1 = 0x00000008.8
; VR5H = temp1[15:0] = 0x0008 = 8
; Imaginary:
; temp2 = (0x00000003 + 0x0000000C) >> 1
; temp2 = (0x0000000F) >> 1 = 0x00000007.8
; VR5L = temp2[15:0] = 0x0007 = 7

VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETHSHR #1 ; VSTATUS[SHIFTR] = 1
VSETHSL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13
VMOVXI VR2, #12 ; VR2 = Im(Y) = 12
VMOVXI VR4, #3
VMOVX VR4, #4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x00080007 = 8 + 7j

The next example illustrates the operation with a right shift value defined as well as rounding.

; Example: Z = X + Y with Right Shift and Rounding
; X = 4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 + 12j (32-bit real + 32-bit imaginary)

; Real:
; temp1 = round((0x00000004 + 0x0000000D) >> 1)
; temp1 = round(0x00000001 >> 1)
; temp1 = round(0x00000008.8) = 0x00000009
; VR5H = temp1[15:0] = 0x0011 = 8
; Imaginary:
; temp2 = round(0x00000003 + 0x0000000C) >> 1)
; temp2 = round(0x0000000F >> 1)
; temp2 = round(0x00000007.8) = 0x00000008
; VR5L = temp2[15:0] = 0x0008 = 8

VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETHSHR #1 ; VSTATUS[SHIFTR] = 1
VSETHSL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13
VMOVXI VR2, #12 ; VR2 = Im(Y) = 12
VMOVXI VR4, #3
VMOVX VR4, #4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x00090008 = 9 + 8j

The next example illustrates the operation with both a right and left shift value defined along with rounding.

; Example: Z = X + Y with Right Shift, Left Shift and Rounding
; X = -4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 - 9j (32-bit real + 32-bit imaginary)

; Real:
; temp1 = 0xFFFFFFFC << 2 + 0x0000000D
; temp1 = 0xFFFFFFF0 + 0x0000000D = 0xFFFFFFF0
; temp1 = 0xFFFFFFF0 >> 1 = 0xFFFFFFF0
; temp1 = 0xFFFFFFF0 << 2 + 0x0000000D
; temp1 = 0xFFFFFFF0 + 0x0000000D = 0xFFFFFFF0
; temp1 = 0xFFFFFFF0 >> 1 = 0xFFFFFFF0

VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETHSHR #1 ; VSTATUS[SHIFTR] = 1
VSETHSL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13
VMOVXI VR2, #12 ; VR2 = Im(Y) = 12
VMOVXI VR4, #3
VMOVX VR4, #4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x00090008 = 9 + 8j
VCDADD16 VR5, VR4, VR3, VR2 — Complex 16 + 32 = 16 Addition

; temp1 = round(0xFFFFFFFFE.8) = 0xFFFFFFFF
; VR5H = temp1[15:0] 0xFFFF = -1;
; Imaginary:
; temp2 = 0x00000003 << 2 + 0xFFFFFFFF
; temp2 = 0x0000000C + 0xFFFFFFFF = 0x00000003
; temp2 = 0x00000003 >> 1 = -0x00000001.8
; temp1 = round(0x000000001.8) = 0x000000002
; VR5L = temp2[15:0] 0x0002 = 2
;
; VSATOFF ; VSTATUS[SAT] = 0
; VRNDON ; VSTATUS[RND] = 1
; VSETSHR #1 ; VSTATUS[SHIFTR] = 1
; VSETSHL #2 ; VSTATUS[SHIFTL] = 2
; VCLEARALL ; VR0, VR1...VR8 == 0
; VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
; VMOVXI VR2, #-9 ; VR2 = Im(Y) = -9
; VMOVIX VR2, #0xFFFF ; sign extend VR2 = 0xFFFFFFFF
; VMOVXI VR4, #3
; VMOVIX VR4, #-4 ; VR4 = X = 0xFFFFC0003 = -4 + 3j
; VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0xFFFF0002 = -1 + 2j

See also

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHL #5-bit
VSETSHR #5-bit
Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR5 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) + (Re(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) + (Im(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VR5L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) + (Im(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) + (Re(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VRa</td>
<td>Contents of the memory pointed to by [mem32]. VRa can not be VR5 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 1010
MSW: 0000 aaaa mem32

Description

Complex 16 + 32 = 16-bit operation with parallel register load. This operation is useful for algorithms similar to a complex FFT.

The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the addition is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 5.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VSTATUS[CPACK] = 0
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
temp2 = sign_extend(VR4L); // 32-bit extended Im(X)

temp1 = (temp1 << SHIFTL) + VR3; // Re(Z) intermediate
temp2 = (temp2 << SHIFTL) + VR2; // Im(Z) intermediate

if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
temp2 = round(temp2 >> SHIFTR);
}
else
{
    temp1 = truncate(temp1 >> SHIFTR);
temp2 = truncate(temp2 >> SHIFTR);
}

if (SAT == 1)
{
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
}
else
{
    VR5H = temp1[15:0];
    VR5L = temp2[15:0];
}

VRa = [mem32];

Flags
This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the real-part (VR5H) computation overflows or underflows.
- OVFI is set if the imaginary-part (VR5L) computation overflows or underflows.

Pipeline
Both operations complete in a single cycle.

Example
For more information regarding the addition operation, see the examples for the VCDADD16 VR5, VR4, VR3, VR2 instruction.

; Example: Right Shift, Left Shift and Rounding
;
; X = -4 + 3j (16-bit real + 16-bit imaginary)
; Y = 13 - 9j (32-bit real + 32-bit imaginary)
;
;
; Real:
; temp1 = 0xFFFFFFFFC << 2 + 0x0000000D
; temp1 = 0xFFFFFFFF0 + 0x0000000D = 0xFFFFFFFFD
; temp1 = 0xFFFFFFFFD >> 1 = 0xFFFFFFFFFE.8
; temp1 = round(0xFFFFFFFFFE.8) = 0xFFFFFFFF
; VR5H = temp1[15:0] 0xFFFF = -1;
; Imaginary:
; temp2 = 0x00000003 << 2 + 0xFFFFFFFF7
; temp2 = 0x0000000C + 0xFFFFFFFF7 = 0x0000000D
; temp2 = 0x0000000D >> 1 = 0x00000001.8
; temp1 = round(0x00000001.8 = 0x00000002
; VR5L = temp2[15:0] 0x0002 = 2
VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex Double Add with Parallel Load

; VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETHTR #1 ; VSTATUS[SHIFTR] = 1
VSETHL #2 ; VSTATUS[SHIFTL] = 2
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVX VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVX VR2, #-9 ; VR2 = Im(Y) = -9
VMOVX VR2, #0xFFFF ; sign extend VR2 = 0xFFFFFFF7
VMOVX VR4, #3
VMOVX VR4, #-4 ; VR4 = X = 0xFFFFFC0003 = -4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0xFFFF0002 = -1 + 2j
|| VCMOV32 VR2, *XAR7 ; VR2 = value pointed to by XAR7

See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETHL #5-bit
VSETHR #5-bit
VCDSUB16 VR6, VR4, VR3, VR2  Complex 16-32 = 16 Subtract

Operands

Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if VSTATUS[CPACK]==0</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR6 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR6H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) -(Re(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) -(Im(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VR6L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if(VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) -(Im(Y) ) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) -(Re(Y) ) &gt;&gt; SHIFTR</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0000 0101

Description

Complex 16 - 32 = 16-bit operation. This operation is useful for algorithms similar to a complex FFT.

The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the subtraction is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 5.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VSTATUS[CPACK] = 0
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

temp1 = sign_extend(VR4H);  // 32-bit extended Re(X)
temp2 = sign_extend(VR4L);  // 32-bit extended Im(X)

temp1 = (temp1 << SHIFTL) - VR3;  // Re(Z) intermediate
temp2 = (temp2 << SHIFTL) - VR2;  // Im(Z) intermediate

if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
    temp2 = round(temp2 >> SHIFTR);
} else
{
    temp1 = truncate(temp1 >> SHIFTR);
    temp2 = truncate(temp2 >> SHIFTR);
}

if (SAT == 1)
{
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
} else
{
    VR5H = temp1[15:0];
    VR5L = temp2[15:0];
}

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the real-part (VR6H) computation overflows or underflows.
- OVFI is set if the imaginary-part (VR6L) computation overflows or underflows.

Pipeline

This is a single-cycle instruction.

Example

; Example: Z = X - Y
;
; X = 4 + 6j (16-bit real + 16-bit imaginary)
; Y = 13 + 22j (32-bit real + 32-bit imaginary)
;
; Z = (4 - 13) + (6 - 22)j = -9 - 16j
;
VMOVX VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVX VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0xFFF7FFF0 = -9 + -16j

The next example illustrates the operation with a right shift value defined.

; Example: Z = X - Y with Right Shift
;
; Y = 4 + 6j (16-bit real + 16-bit imaginary)
; X = 13 + 22j (32-bit real + 32-bit imaginary)
;
; Real:
;   temp1 = (0x00000004 - 0x0000000D) >> 1
;   temp1 = (0xFFFFFFF7) >> 1
;  temp1 = 0xFFFFFFFFF8
;  VR5H = temp1[15:0] = 0xFFF8 = -8
;  Imaginary:
;  temp2 = (0x00000006 - 0x00000000) >> 1
;  temp2 = (0xFFFFFFFFF0) >> 1
;  temp2 = 0xFFFFFFFFF8
;  VR5L = temp2[15:0] = 0xFFF8 = -8

VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOXI VR4, #4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0xFFFFFFFFF8 = -5 + -8j

The next example illustrates rounding with a right shift value defined.

;  Example: Z = X-Y with Rounding and Right Shift
;  X = 4 + 6j (16-bit real + 16-bit imaginary)
;  Y = -13 + 22j (32-bit real + 32-bit imaginary)
;  Real:
;  temp1 = round((0x00000004 - 0xFFFFFFF3) >> 1)
;  temp1 = round((0x00000010 - 0xFFFFFFF3) >> 1)
;  temp1 = round(0x0000001D >> 1)
;  temp1 = round(0x0000000E.8) = 0x0000000F
;  VR5H = temp1[15:0] = 0x000F = 15
;  Imaginary:
;  temp2 = round((0x00000006 - 0x00000016) >> 1)

VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #-13 ; VR3 = Re(Y) = -13 = 0xFFFFFFF3
VMOXI VR3, #0xFFFF ; sign extend VR3 = -13 = 0xFFFFFFF3
VMOXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOXI VR4, #6
VMOXI VR4, #4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0x0009FFFFF = -5 + -8j

The next example illustrates rounding with both a left and a right shift value defined.

;  Example: Z = X-Y with Rounding and both Left and Right Shift
;  X = 4 + 6j (16-bit real + 16-bit imaginary)
;  Y = -13 + 22j (32-bit real + 32-bit imaginary)
;  Real:
;  temp1 = round((0x00000004 << 2 - 0xFFFFFFF3) >> 1)
;  temp1 = round((0x00000010 << 2 - 0xFFFFFFF3) >> 1)
;  temp1 = round(0x0000001D >> 1)
;  temp1 = round(0x0000000E.8) = 0x0000000F
;  VR5H = temp1[15:0] = 0x000F = 15
;  Imaginary:
;  temp2 = round((0x00000006 << 2 - 0x00000016) >> 1)
VCDSUB16 VR6, VR4, VR3, VR2 — Complex 16-32 = 16 Subtract

```
; temp2 = round((0x00000018 - 0x00000016) >> 1)
; temp2 = round( 0x00000002 >> 1)
; temp1 = round( 0x00000001.0) - 0x00000001
; VR5L = temp2[15:0] - 0x0001 = 1
;
VSATOFF ; VSTATUS[SAT] = 0
VRNDON  ; VSTATUS[RND] = 1
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #2 ; VSTATUS[SHIFTL] = 2
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI  VR3, #13    ; VR3 = Re(Y)
VMOVIX  VR3, #0xFFFF ; sign extend VR3 = -13 = 0xFFFFFFF3
VMOVXI  VR2, #22    ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI  VR4, #6     ; VR4 = X = 0x00000006 = 4 + 6j
VMOVIX  VR4, #4     ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0x000F0001 = 15 + 1j
```

See also

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHL #5-bit
VSETSHR #5-bit
VCDSUB16 VR6, VR4, VR3, VR2 || VMOV32 VRa, mem32  Complex 16-32 = 16 Subtract with Parallel Load

Operands

Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Im(X)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Re(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result is a complex number with a 16-bit real and a 16-bit imaginary part. The result is stored in VR6 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR6H</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) - (Re(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) - (Im(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VR6L</td>
<td>16-bit integer:</td>
</tr>
<tr>
<td></td>
<td>if (VSTATUS[CPACK]==0)</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = (Im(X) &lt;&lt; SHIFTL) - (Im(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td></td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = (Re(X) &lt;&lt; SHIFTL) - (Re(Y)) &gt;&gt; SHIFTR</td>
</tr>
<tr>
<td>VRa</td>
<td>Contents of the memory pointed to by [mem32]. VRa cannot be VR6 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 1011
MSW: 0000 aaaa mem32

Description

Complex 16 - 32 = 16-bit operation with parallel load. This operation is useful for algorithms similar to a complex FFT.

The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.

Before the addition, the first input is sign extended to 32-bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the subtraction is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 5.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.

// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VSTATUS[CPACK] = 0
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
temp2 = sign_extend(VR4L); // 32-bit extended Im(X)

if (RND == 1)
{
    temp1 = round(temp1 >> SHIFTR);
    temp2 = round(temp2 >> SHIFTR);
}
else
{
    temp1 = truncate(temp1 >> SHIFTR);
    temp2 = truncate(temp2 >> SHIFTR);
}

if (SAT == 1)
{
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
}
else
{
    VR5H = temp1[15:0];
    VR5L = temp2[15:0];
}

VRa = [mem32];

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the real-part (VR6H) computation overflows or underflows.
- OVFI is set if the imaginary-part (VR6l) computation overflows or underflows.

Pipeline

Both operations complete in a single cycle.

Example

For more information regarding the subtraction operation, please refer to VCDSUB16 VR6, VR4, VR3, VR2.

; Example: Z = X-Y with Rounding and both Left and Right Shift
;
; X = 4 + 6j (16-bit real + 16-bit imaginary)
; Y = -13 + 22j (32-bit real + 32-bit imaginary)
;
; Real:
; temp1 = round((0x00000004 << 2 - 0xFFFFFFF3) >> 1)
; temp1 = round((0x00000010 - 0xFFFFFFF3) >> 1)
; temp1 = round( 0x0000001D >> 1)
; temp1 = round( 0x0000000E.8) = 0x0000000F
; VR5H = temp1[15:0] = 0x000F = 15
; Imaginary:
; temp2 = round((0x00000006 << 2 - 0x00000016) >> 1)
; temp2 = round((0x00000018 - 0x00000016) >> 1)
; temp2 = round( 0x00000002 >> 1)
; temp1 = round( 0x00000001.0) = 0x00000001
; VR5L = temp2[15:0] = 0x0001 = 1

VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHR #1 ; VSTATUS[SHIFTR] = 1
VSETSHL #2 ; VSTATUS[SHIFTL] = 2
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, #-13 ; VR3 = Re(Y)
VMOVIX VR3, #0xFFFF ; sign extend VR3 = -13 = 0xFFFFFFFF
VMOVXI VR2, #22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, #6
VMOVIX VR4, #4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0x000F0001 = 15 + 1j
VMOVIX VR2, *XAR7 ; VR2 = contents pointed to by XAR7

See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHL #5-bit
VSETSHR #5-bit
### VCFLIP VRa — Swap Upper and Lower Half of VCU Register

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VRa</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Opcode</strong></th>
<th>LSW: 1010 0001 0000 aaaa</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Swap VRaL and VRaH</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>This instruction does not affect any flags in the VSTATUS register</td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>This is a single-cycle instruction.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>VCFLIP VR7 ; VR7H := VR7L</td>
</tr>
<tr>
<td><strong>See also</strong></td>
<td></td>
</tr>
</tbody>
</table>
VCMAC VR5, VR4, VR3, VR2, VR1, VR0  Complex Multiply and Accumulate

Operands

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>Real part of the accumulation</td>
</tr>
<tr>
<td>VR4</td>
<td>Imaginary part of the accumulation</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the product</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the product</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
</tbody>
</table>

NOTE: The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.

Opcode

LSW: 1110 0101 0000 0001

Description

Complex multiply operation.

// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
//
// VR0 = X + jX: VR0[31:16] = X, VR0[15:0] = jX
//
// Perform add
//
// if (RND == 1)
// {
// VR5 = VR5 + round(VR3 >> SHIFTR);
// VR4 = VR4 + round(VR2 >> SHIFTR);
// }
// else
// {
// VR5 = VR5 + (VR3 >> SHIFTR);
// VR4 = VR4 + (VR2 >> SHIFTR);
// }
//
// Perform multiply (X + jX) * (Y + jY)
//
// if(VSTATUS[CPACK] == 0){
// VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
// }else{
// VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0L * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
// }
// if(SAT == 1)
// {
// sat32(VR3);
// sat32(VR2);
// }

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline

This is a 2p-cycle instruction.

Example
See also

VCLROVFI
VCLROVFR
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSATON
VSATOFF
Operands

The VMAC alternates which registers are used between each cycle. For odd cycles (1, 3, 5, and so on) the following registers are used:

<table>
<thead>
<tr>
<th>Odd Cycle Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>Previous real-part total accumulation: Re(odd_sum)</td>
</tr>
<tr>
<td>VR4</td>
<td>Previous imaginary-part total accumulation: Im(odd-sum)</td>
</tr>
<tr>
<td>VR1</td>
<td>Previous real result from the multiply: Re(odd-mpy)</td>
</tr>
<tr>
<td>VR0</td>
<td>Previous imaginary result from the multiply: Im(odd-mpy)</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Pointer to a 32-bit memory location representing the first input to the multiply</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 0)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Re(X)</td>
</tr>
<tr>
<td></td>
<td>[mem32][15:0] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 1)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>[mem32][15:0] = Re(X)</td>
</tr>
<tr>
<td>XAR7</td>
<td>Pointer to a 32-bit memory location representing the second input to the multiply</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 0)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[32:16] = Re(X)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[15:0] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 1)</td>
</tr>
<tr>
<td></td>
<td>*XAR7[32:16] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>*XAR7 [15:0] = Re(X)</td>
</tr>
</tbody>
</table>

The result from odd cycle is stored as shown below:

<table>
<thead>
<tr>
<th>Odd Cycle Output</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit real part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Re(odd_sum) = Re(odd_sum) + Re(odd_mpy)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit imaginary part of the total accumulation</td>
</tr>
<tr>
<td></td>
<td>Im(odd_sum) = Im(odd_sum) + Im(odd_mpy)</td>
</tr>
<tr>
<td>VR1</td>
<td>32-bit real result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)</td>
</tr>
<tr>
<td>VR0</td>
<td>32-bit imaginary result from the multiplication:</td>
</tr>
<tr>
<td></td>
<td>Im(Z) = Re(X)*Im(Y) + Re(Y)*Im(X)</td>
</tr>
</tbody>
</table>

For even cycles (2, 4, 6, and so on) the following registers are used:

<table>
<thead>
<tr>
<th>Even Cycle Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>Previous real-part total accumulation: Re(even_sum)</td>
</tr>
<tr>
<td>VR6</td>
<td>Previous imaginary-part total accumulation: Im(even-sum)</td>
</tr>
<tr>
<td>VR3</td>
<td>Previous real result from the multiply: Re(even-mpy)</td>
</tr>
<tr>
<td>VR2</td>
<td>Previous imaginary result from the multiply Im(even-mpy)</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Pointer to a 32-bit memory location representing the first input to the multiply</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 0)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Re(X)</td>
</tr>
<tr>
<td></td>
<td>[mem32][15:0] = Im(X)</td>
</tr>
<tr>
<td></td>
<td>If(VSTATUS[CPACK] == 1)</td>
</tr>
<tr>
<td></td>
<td>[mem32][32:16] = Im(X)</td>
</tr>
</tbody>
</table>
**Even Cycle Input**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32][15:0] = Re(X)</td>
<td>Pointer to a 32-bit memory location representing the second input to the multiply</td>
</tr>
<tr>
<td>*XAR7++</td>
<td>If(VSTATUS[CPACK] == 0)</td>
</tr>
<tr>
<td>*XAR7[32:16] = Re(X)</td>
<td></td>
</tr>
<tr>
<td>*XAR7[15:0] = Im(X)</td>
<td></td>
</tr>
<tr>
<td>If(VSTATUS[CPACK] == 1)</td>
<td></td>
</tr>
<tr>
<td>*XAR7[32:16] = Im(X)</td>
<td></td>
</tr>
<tr>
<td>*XAR7[15:0] = Re(X)</td>
<td></td>
</tr>
</tbody>
</table>

The result from even cycles is stored as shown below:

**Even Cycle Output**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>32-bit real part of the total accumulation</td>
</tr>
<tr>
<td>Re(even_sum) = Re(even_sum) + Re(even_mpy)</td>
<td></td>
</tr>
<tr>
<td>VR6</td>
<td>32-bit imaginary part of the total accumulation</td>
</tr>
<tr>
<td>Im(even_sum) = Im(even_sum) + Im(even_mpy)</td>
<td></td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit real result from the multiplication:</td>
</tr>
<tr>
<td>Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)</td>
<td></td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit imaginary result from the multiplication:</td>
</tr>
<tr>
<td>Im(Z) = Re(X)*Im(Y) + Re(Y)*Im(X)</td>
<td></td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 0101 0001  
MSW: 0000 0000 mem32

**Description**

Perform a repeated multiply and accumulate operation. This instruction must be used with the repeat instruction (RPT||). The destination of the accumulate will alternate between VR7/VR6 and VR5/VR4 on each cycle.

// Cycle 1:
//
// Perform accumulate
//
if(RND == 1) {
    VR5 = VR5 + round(VR1 >> SHIFTR)
    VR4 = VR4 + round(VR0 >> SHIFTR)
} else {
    VR5 = VR5 + (VR1 >> SHIFTR)
    VR4 = VR4 + (VR0 >> SHIFTR)
}

// X and Y array element 0
//
VR1 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) + Re(Y)*Im(X)

//
// Cycle 2:
//
// Perform accumulate
//
if(RND == 1) {
    VR7 = VR7 + round(VR3 >> SHIFTR)
    VR6 = VR6 + round(VR2 >> SHIFTR)
} else {

VR7 = VR7 + (VR3 >> SHIFTR)
VR6 = VR6 + (VR2 >> SHIFTR)
}
// X and Y array element 1
//
VR3 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = Re(X)*Im(Y) + Re(Y)*Im(X)
// Cycle 3:
//
// Perform accumulate
//
if(RND == 1)
{
    VR5 = VR5 + round(VR1 >> SHIFTR)
    VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
    VR5 = VR5 + (VR1 >> SHIFTR)
    VR4 = VR4 + (VR0 >> SHIFTR)
}
// X and Y array element 2
//
VR1 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) + Re(Y)*Im(X)
etc...

Restrictions
VR0, VR1, VR2, and VR3 will be used as temporary storage by this instruction.

Flags
The VSTATUS register flags are modified as follows:

- OVFR is set in the case of an overflow or underflow of the addition or subtraction operations.
- OVFI is set in the case an overflow or underflow of the imaginary part of the addition or subtraction operations.

Pipeline
The VCCMAC takes 2p + N cycles where N is the number of times the instruction is repeated. This instruction has the following pipeline restrictions:

```
<<instruction1>> ; No restrictions
<<instruction2>> ; Cannot be a 2p instruction that writes
                 ; to VR0, VR1...VR7 registers
RPT #(N-1) ; Execute N times, where N is even
 || VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
<<instruction3>> ; No restrictions
                 ; Can read VR0, VR1...VR8
```

Example
Cascading of RPT || VCMAC is allowed as long as the first and subsequent counts are even. Cascading is useful for creating interruptible windows so that interrupts are not delayed too long by the RPT instruction. For example:

```
; Example of cascaded VMAC instructions
;
VCLEARALL ; Zero the accumulation registers
;
; Execute MACF32 N+1 (4) times
;
RPT #3
 || VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
;
; Execute MACF32 N+1 (6) times
```
RPT #5
|| VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
;
; Repeat MACF32 N+1 times where N+1 is even
;
RPT #N
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
ADDF32 VR7, VR6, VR5, VR4

See also
VCCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32

Complex Multiply and Accumulate with Parallel Load

Operands

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the product</td>
</tr>
<tr>
<td>VR3</td>
<td>Real part of the product</td>
</tr>
<tr>
<td>VR4</td>
<td>Imaginary part of the accumulation</td>
</tr>
<tr>
<td>VR5</td>
<td>Real part of the accumulation</td>
</tr>
<tr>
<td>VRa</td>
<td>Contents of the memory pointed to by mem32. VRa cannot be VR5, VR4, or VR8</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

NOTE: The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.

Opcode

LSW: 1110 0011 1111 0111
MSW: 0000 aaaa mem32

Description

Complex multiply operation.

// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
//
// VR0 - X + Xj: VR0[31:16] = Re(X), VR0[15:0] = Im(X)
// VR1 - Y + Yj: VR1[31:16] = Re(Y), VR1[15:0] = Im(Y)
//
// Perform add
//
// if (RND == 1)
// {
//    VR5 = VR5 + round(VR3 >> SHIFTR);
//    VR4 = VR4 + round(VR2 >> SHIFTR);
// }
// else
// {
//    VR5 = VR5 + (VR3 >> SHIFTR);
//    VR4 = VR4 + (VR2 >> SHIFTR);
// }
//
// Perform multiply Z = (X + Xj) * (Y + Yj)
//
// if(VSTATUS[CPACK] == 0){
//    VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
//    VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
// }else{
//    VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
//    VR2 = VR0L * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
// }
// if(SAT == 1){
//    sat32(VR3);
//    sat32(VR2);
// }
// VRa = [mem32];

Flags

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline
This is a 2p/1-cycle instruction. The multiply and accumulate is a 2p-cycle operation and the VMOV32 is a single-cycle operation.

Example

See also
VCLROVFI
VCLROVFR
VCMAC VR5, VR4, VR3, VR2, VR1, VR0
VSATON
VSATOFF
VCMAG VRb, VRa — Magnitude of a Complex Number

Operands

VRb General purpose register VR0…VR8
VRa General purpose register VR0…VR8

Opcode

LSW: 1110 0110 1111 0010
MSW: 0000 0100 bbbb aaaa

Description

Compute the magnitude of the Complex value in VRa

If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.

```
If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VRb = rnd(sat(VRaH*VRaH + VRaL*VRaL)>>VSTATUS[SHIFTR])
    }else {
        VRb = sat(VRaH*VRaH + VRaL*VRaL)>>VSTATUS[SHIFTR]
    }
}else { //VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1){
        VRb = rnd((VRaH*VRaH + VRaL*VRaL)>>VSTATUS[SHIFTR])
    }else {
        VRb = (VRaH*VRaH + VRaL*VRaL)>>VSTATUS[SHIFTR]
    }
}
```

Sign-Extension is automatically done for the shift right operations

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if overflow is detected in the complex magnitude operation of the real 32-bit result

Pipeline

This is a 2 cycle instruction

Example

```
VMOV32 VR1, VR0 ; VR1 := VR0
VCCON VR1 ; VR1 := VR1^*
VCMAG VR2, VR0 ; VR2 := magnitude(VR0)
and so forth
```

See also
VCMPY VR3, VR2, VR1, VR0 — Complex Multiply

Operands
Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0000 0000

Description
Complex 16 x 16 = 32-bit multiply operation.
If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, the result will be saturated in the event of a 32-bit overflow or underflow.

// Calculate: Z = (X + jX) * (Y + jY)
// if(VSTATUS[CPACK] == 0){
// VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
//}else{
// VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0L * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
//}
if(SAT == 1)
{
    sat32(VR3);
    sat32(VR2);
}

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline
This is a 2p-cycle instruction. The instruction following this one should not use VR3 or VR2.

Example
; Example 1
; X = 4 + 6j
; Y = 12 + 9j
;
; Z = X * Y
; Re(Z) = 4*12 - 6*9 = -6
; Im(Z) = 4*9 + 6*12 = 108
;
VSATOFF ; VSTATUS[SAT] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR0, #6
VMOVIX VR0, #4 ; VR0 = X = 0x00040006 = 4 + 6j
VMOVXI VR1, #9
VMOVIX VR1, #12 ; VR1 = Y = 0x000C0009 = 12 + 9j
VCMPY VR3, VR2, VR1, VR0 ; VR3 = Re(Z) = 0xFFFFFFFF = -6
; VR2 = Im(Z) = 0x0000006C = 108
<instruction 1> ; <- Must not use VR2, VR3
<instruction 2> ; <- VCMPY completes, VR2, VR3 valid
<instruction 2> ; Can use VR2, VR3
See also

- VCLROVFI
- VCLROVFR
- VCMAC VR5, VR4, VR3, VR2, VR1, VR0
- VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
- VSATON
- VSATOFF
VCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa — Complex Multiply with Parallel Store

Operands
Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VRa</td>
<td>Value to be stored</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0010 1100 1010
MSW: 0000 aaaa mem16

Description
Complex 16 x 16 = 32-bit multiply operation with parallel register load.

If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.

// Calculate: Z = (X + jX) * (Y + jY)
//
// if(VSTATUS[CPACK] == 0){
// VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
//}else{
// VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
// VR2 = VR0L * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
//}
if(SAT == 1)
{
    sat32(VR3);
    sat32(VR2);
}
VRa = [mem32];

Flags
This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline
This is a 2p/1-cycle instruction. The multiply operation takes 2p cycles and the VMOV operation completes in a single cycle. The instruction following this one must not use VR2 or VR3.

Example
; Example 1
; X = 4 + 6j
; Y = 12 + 9j
;
; Z = X * Y
; Re(Z) = 4*12 - 6*9 = -6
; Im(Z) = 4*9 + 6*12 = 108
;
VSATOFF ; VSTATUS[SAT] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR0, #6
VMOVIX VR0, #4 ; VR0 = X = 0x00040006 = 4 + 6j
VMOVXI VR1, #9
VMOVIX VR1, #12 ; VR1 = Y = 0x000C0009 = 12 + 9j
VR3 = Re(Z) = 0xFFFFFFF6 = -6
VCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa — Complex Multiply with Parallel Store

VCMPY VR3, VR2, VR1, VR0 ; VR2 = Im(Z) = 0x0000006C = 108
|| VMOV32 *XAR7, VR3 ; Location XAR7 points to = VR3 (before multiply)
    <instruction 1> ; <- Must not use VR2, VR3
    <instruction 2> ; <- VCMPY completes, VR2, VR3 valid

See also

VCLROVF1
VCLROVFR
VCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSATON
VSATOFF
VCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 — Complex Multiply with Parallel Load

Operands
Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3</td>
<td>Real part of the Result</td>
</tr>
<tr>
<td>VR2</td>
<td>Imaginary part of the Result</td>
</tr>
<tr>
<td>VR1</td>
<td>Second Complex Operand</td>
</tr>
<tr>
<td>VR0</td>
<td>First Complex Operand</td>
</tr>
<tr>
<td>VRa</td>
<td>32-bit value pointed to by mem32. VRa can not be VR2, VR3 or VR8.</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0011 1111 0110
MSW: 0000 aaaa mem32

Description
Complex 16 x 16 = 32-bit multiply operation with parallel register load.
If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.

// Calculate: Z = (X + jX) * (Y + jY)
if(VSTATUS[CPACK] == 0){
    VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
    VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}else{
    VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
    VR2 = VR0L * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}
if(SAT == 1)
{
    sat32(VR3);
    sat32(VR2);
}
VRa = [mem32];

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline
This is a 2p/1-cycle instruction. The multiply operation takes 2p cycles and the VMOV operation completes in a single cycle. The instruction following this one must not use VR2 or VR3.

Example
; Example 1
; X = 4 + 6j
; Y = 12 + 9j
;
; Z = X * Y
; Re(Z) = 4*12 - 6*9 = -6
; Im(Z) = 4*9 + 6*12 = 108
;
VSATOFF ; VSTATUS[SAT] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR0, #6
VMOVIX VR0, #4 ; VR0 = X = 0x00040006 = 4 + 6j
VMOVXI VR1, #9
VMOVIX VR1, #12 ; VR1 = Y = 0x000C0009 = 12 + 9j
; VR3 = Re(Z) = 0xFFFFFFFF - 6
VCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 — Complex Multiply with Parallel Load

VCMPY VR3, VR2, VR1, VR0 ; VR2 = Im(Z) = 0x0000006C = 108
|| VMOV32 VR0, *XAR7 ; VR0 = contents of location XAR7 points to
           <instruction 1> ; <- Must not use VR2, VR3
            ; <- VCMPY completes, VR2, VR3 valid
           <instruction 2> ; Can use VR2, VR3

See also
VCLROVFI
VCLROVFR
VCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSATON
VSATOFF
Complex Shift Left

**Operands**

- **VRa** General purpose register VR0…VR8
- **#4-bit** 4-bit unsigned immediate value

**Opcode**

- **LSW**: 1110 0110 1111 0010
- **MSW**: 0000 0000 IIII aaaa

**Description**

Left Shift the Real and Imaginary parts of the complex value in VRa.

```c
if(VSTATUS[CPACK] == 0){
    if(VSTATUS[SAT] == 1){
        VRaL = sat(VRaL << #4-bit Immediate) (imaginary result)
        VRaH = sat(VRaH << #4-bit Immediate) (real result)
    }else {
        VRaL = VRaL << #4-bit Immediate (imaginary result)
        VRaH = VRaH << #4-bit Immediate (real result)
    }
}else {
    if(VSTATUS[SAT] == 1){
        VRaL = sat(VRaL << #4-bit Immediate) (real result)
        VRaH = sat(VRaH << #4-bit Immediate) (imaginary result)
    }else {
        VRaL = VRaL << #4-bit Immediate (real result)
        VRaH = VRaH << #4-bit Immediate (imaginary result)
    }
}
```

**Flags**

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if overflow is detected in the shift left operation of the real signed-16-bit result.
- OVFI is set if overflow is detected in the shift left operation of the imaginary signed-16-bit result.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
VSATOFF ; turn off saturation
VCSHL16 VR5 << #8 ; VR5L := VR5L << 8 | VR5H := VR5H << 8
```

**See also**
VCSHR16 VRa >> #4-bit  Complex Shift Right

Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register VR0…VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#4-bit</td>
<td>4-bit unsigned immediate value</td>
</tr>
</tbody>
</table>

Opcode

- LSW: 1110 0110 1111 0010
- MSW: 0000 0001 IIII aaaa

Description

Right Shift the Real and Imaginary parts of the complex value in VRa.

```c
if(VSTATUS[CPACK] == 0){
    if(VSTATUS[RND] == 1){
        VRaL = rnd(VRaL >> #4-bit Immediate) (imaginary result)
        VRaH = rnd(VRaH >> #4-bit Immediate) (real result)
    }else {
        VRaL = VRaL >> #4-bit Immediate (imaginary result)
        VRaH = VRaH >> #4-bit Immediate (real result)
    }
}else {
    if(VSTATUS[RND] == 1){
        VRaL = rnd(VRaL >> #4-bit Immediate) (real result)
        VRaH = rnd(VRaH >> #4-bit Immediate) (imaginary result)
    }else {
        VRaL = VRaL >> #4-bit Immediate (real result)
        VRaH = VRaH >> #4-bit Immediate (imaginary result)
    }
}
```

Sign-Extension is automatically done for the shift right operations

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```c
VSATOFF ; turn off saturation
VCSHR16 VR6 >> #8 ; VR6L := VR6L >> 8 | VR6H := VR6H >> 8
```

See also
VCSUB VR5, VR4, VR3, VR2 — Complex 32 - 32 = 32 Subtraction

Operands

Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) - (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) - (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0000 0011

Description

Complex 32 - 32 = 32-bit subtraction operation.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFTR] bits before the subtraction. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 5.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```c
if (RND == 1)
{
    VR5 = VR5 - round(VR3 >> SHIFTR);
    VR4 = VR4 - round(VR2 >> SHIFTR);
}
else
{
    VR5 = VR5 - (VR3 >> SHIFTR);
    VR4 = VR4 - (VR2 >> SHIFTR);
}
if (SAT == 1)
{
    sat32(VR5);
    sat32(VR4);
}
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if the VR5 computation (real part) overflows or underflows.
- OVFI is set if the VR6 computation (imaginary part) overflows or underflows.

Pipeline

This is a single-cycle instruction.

Example

See also

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR #5-bit
VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 — Complex Subtraction

Operands
Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the first input: Re(X)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the first input: Im(X)</td>
</tr>
<tr>
<td>VR3</td>
<td>32-bit integer representing the real part of the 2nd input: Re(Y)</td>
</tr>
<tr>
<td>VR2</td>
<td>32-bit integer representing the imaginary part of the 2nd input: Im(Y)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location</td>
</tr>
</tbody>
</table>

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR5</td>
<td>32-bit integer representing the real part of the result: Re(Z) = Re(X) - (Re(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VR4</td>
<td>32-bit integer representing the imaginary part of the result: Im(Z) = Im(X) - (Im(Y) &gt;&gt; SHIFTR)</td>
</tr>
<tr>
<td>VRa</td>
<td>contents of the memory pointed to by [mem32]. VRa can not be VR5, VR4 or VR8.</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0011 1111 1001
MSW: 0000 aaaa mem32

Description
Complex 32 - 32 = 32-bit subtraction operation with parallel load.

The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFTR] bits before the subtraction. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 5.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.

```
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]

if (RND == 1)
{
    VR5 = VR5 - round(VR3 >> SHIFTR);
    VR4 = VR4 - round(VR2 >> SHIFTR);
}
else
{
    VR5 = VR5 - (VR3 >> SHIFTR);
    VR4 = VR4 - (VR2 >> SHIFTR);
}
if (SAT == 1)
{
    sat32(VR5);
    sat32(VR4);
}
VRa = [mem32];
```

Flags
This instruction modifies the following bits in the VSTATUS register:
• OVFR is set if the VR5 computation (real part) overflows or underflows.
• OVFI is set if the VR6 computation (imaginary part) overflows or underflows.

Pipeline
This is a single-cycle instruction.
See also

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCSUB VR5, VR4, VR3, VR2
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR #5-bit
## 5.5.5 Cyclic Redundancy Check (CRC) Instructions

The instructions are listed alphabetically, preceded by a summary.

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRC8H_1 mem16 — CRC8, High Byte</td>
<td>639</td>
</tr>
<tr>
<td>VCRC8L_1 mem16 — CRC8, Low Byte</td>
<td>640</td>
</tr>
<tr>
<td>VCRC16P1H_1 mem16 — CRC16, Polynomial 1, High Byte</td>
<td>641</td>
</tr>
<tr>
<td>VCRC16P1L_1 mem16 — CRC16, Polynomial 1, Low Byte</td>
<td>642</td>
</tr>
<tr>
<td>VCRC16P2H_1 mem16 — CRC16, Polynomial 2, High Byte</td>
<td>643</td>
</tr>
<tr>
<td>VCRC16P2L_1 mem16 — CRC16, Polynomial 2, Low Byte</td>
<td>644</td>
</tr>
<tr>
<td>VCRC24H_1 mem16 — CRC24, High Byte</td>
<td>645</td>
</tr>
<tr>
<td>VCRC24L_1 mem16 — CRC24, Low Byte</td>
<td>646</td>
</tr>
<tr>
<td>VCRC32H_1 mem16 — CRC32, High Byte</td>
<td>647</td>
</tr>
<tr>
<td>VCRC32L_1 mem16 — CRC32, Low Byte</td>
<td>648</td>
</tr>
<tr>
<td>VCRC32P2H_1 mem16 — CRC32, Polynomial 2, High Byte</td>
<td>649</td>
</tr>
<tr>
<td>VCRC32P2L_1 mem16 — CRC32, Low Byte</td>
<td>650</td>
</tr>
<tr>
<td>VCRCLR</td>
<td>651</td>
</tr>
<tr>
<td>VMOV32 mem32, VCRC — Store the CRC Result Register</td>
<td>652</td>
</tr>
<tr>
<td>VMOV32 VCRC, mem32 — Load the CRC Result Register</td>
<td>653</td>
</tr>
</tbody>
</table>
VCRC8H_1 mem16  \textit{CRC8, High Byte}

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

LSW: 1110 0010 1100 1100  
MSW: 0000 0000  mem16

**Description**

This instruction uses CRC8 polynomial $\equiv 0x07$. Calculate the CRC8 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

$$\text{if (VSTATUS[CRCMSGFLIP == 0])}$$
$$\text{temp[7:0] = [mem16][15:8];}$$
$$\text{else}$$
$$\text{temp[7:0] = [mem16][8:15];}$$
$$\text{VCRC[7:0] = CRC8 (VCRC[7:0], temp[7:0])}$$

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VCRC8L_1 mem16

**See also**

VCRC8L_1 mem16
VCRC8L_1 mem16  

**CRC8, Low Byte**

**Operands**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>16-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0010</td>
<td>1100 1011</td>
<td>Recursive 8-bit CRC instruction</td>
</tr>
</tbody>
</table>

**Description**

This instruction uses CRC8 polynomial == 0x07.

Calculate the CRC8 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```c
if (VSTATUS[CRCMSGFLIP] == 0) {
    temp[7:0] = [mem16][7:0];
} else{
    temp[7:0] = [mem16][0:7];
}
VCRC[7:0] = CRC8 (VCRC[7:0], temp[7:0])
```

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData;   // Start of data
    uint16_t CRCLen;     // Length of data in bytes
}CRC_CALC;

CRC_CALC mycrc;
...
CRC8(&mycrc);
...

; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words
.
.global _CRC8

_CRC8
    VCRCCLR ; Clear the result register
    MOV AL, *+XAR4[4] ; AL = CRCLen
    ASR AL, 2 ; AL = CRCLen/4
    SUBB AL, #1 ; AL = CRCLen/4 - 1
    MOVL XAR7, *+XAR4[2] ; XAR7 = &CRCData
    .align 2
    NOP ; Align RPTB to an odd address
    RPTB _CRC8_done, AL ; Execute block of code AL + 1 times
    VCRC8L_1 *XAR7 ; Calculate CRC for 4 bytes
    VCRC8H_1 *XAR7++ ; ...
    VCRC8L_1 *XAR7++ ; ...
    VCRC8H_1 *XAR7++ ; ...
    _CRC8_done
        MOVL XAR7, *+XAR4[0] ; XAR7 = &CRCResult
        VMOV32 *+XAR7[0], VCRC ; Store the result
    LRET ; return to caller
```

**See also**

VCRC8H_1 mem16
VCRC16P1H\_1 mem16 \textit{CRC16, Polynomial 1, High Byte}

**Operands**

<table>
<thead>
<tr>
<th>mem16</th>
<th>16-bit memory location</th>
</tr>
</thead>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW: 1110 0010 1100 1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 mem16</td>
</tr>
</tbody>
</table>

**Description**

This instruction uses CRC16 polynomial 1 \(== 0x8005\).

Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

\[
\text{if (VSTATUS[CRCMSGFLIP] == 0)}\{
    \text{temp}[7:0] = [\text{mem16}][15:8];
\text{else}\{
    \text{temp}[7:0] = [\text{mem16}][8:15];
\}
\]

\[
\text{VCRC}[15:0] = \text{CRC16(VCRC}[15:0], \text{temp}[7:0])
\]

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VCRC16P1L\_1 mem16.

**See also**

VCRC16P1L\_1 mem16  
VCRC16P2H\_1 mem16  
VCRC16P2L\_1 mem16
VCRC16P1L_1 mem16 **CRC16, Polynomial 1, Low Byte**

**Operands**

<table>
<thead>
<tr>
<th>mem16</th>
<th>16-bit memory location</th>
</tr>
</thead>
</table>

**Opcode**

<table>
<thead>
<tr>
<th>LSW: 1110 0010 1100 1110</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 mem16</td>
</tr>
</tbody>
</table>

**Description**

This instruction uses CRC16 polynomial 1 == 0x8005. Calculate the CRC16 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```c
if (VSTATUS[CRCMSGFLIP] == 0) {
    temp[7:0] = [mem16][7:0];
} else {
    temp[7:0] = [mem16][0:7];
}

VCRC[15:0] = CRC16 (VCRC[15:0], temp[7:0])
```

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData; // Start of data
    uint16_t CRCLen; // Length of data in bytes
} CRC_CALC;

CRC_CALC mycrc;
...
CRC16P1(&mycrc);
...

; -------------------
; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words
;
.global _CRC16P1
_CRC16P1

VCRCCLR ; Clear the result register
MOV AL, *+XAR4[4] ; AL = CRCLen
ASR AL, 2 ; AL = CRCLen/4
SUBB AL, #1 ; AL = CRCLen/4 - 1
MOVL XAR7, *+XAR4[2] ; XAR7 = 4CRCData
.align 2
NOP ; Align RPTB to an odd address
RPTB _CRC16P1_done, AL ; Execute block of code AL + 1 times
VCRC16P1L_1 *XAR7 ; Calculate CRC for 4 bytes
VCRC16P1H_1 *XAR7++; ...
VCRC16P1L_1 *XAR7; ...
VCRC16P1H_1 *XAR7++; ...
_CRC16P1_done
MOVL XAR7, *+XAR4[0] ; XAR7 = 4CRCResult
VMOV32 *+XAR7[0], VCRC ; Store the result
LRETR ; return to caller
```

**See also**

- VCRC16P1H_1 mem16
- VCRC16P2H_1 mem16
- VCRC16P2L_1 mem16
VCRC16P2H_1 mem16

**Operands**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>16-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 1100 1111  
MSW: 0001 0000 mem16

**Description**

This instruction uses CRC16 polynomial 2 == 0x1021.

Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```c
if (VSTATUS[CRCMSGFLIP] == 0){
    temp[7:0] = [mem16][15:8];
}else {
    temp[7:0] = [mem16][8:15];
}

VCRC[15:0] = CRC16(VCRC[15:0], temp[7:0])
```

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VCRC16P2L_1 mem16.

**See also**

VCRC16P2L_1 mem16  
VCRC16P1H_1 mem16  
VCRC16P1L_1 mem16
VCRC16P2L_1 mem16  CRC16, Polynomial 2, Low Byte

Operands

| mem16     | 16-bit memory location |

Opcode

LSW: 1110 0010 1100 1110
MSW: 0001 0000 mem16

Description

This instruction uses CRC16 polynomial 2== 0x1021.
Calculate the CRC16 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if (VSTATUS[CRCMSGFLIP] == 0){
temp[7:0] = [mem16][7:0];
} else {
temp[7:0] = [mem16][0:7];
}
VCRC[15:0] = CRC16 (VCRC[15:0], temp[7:0])
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```
typedef struct {
    uint32_t  *CRCResult;  // Address where result should be stored
    uint16_t   *CRCData;   // Start of data
    uint16_t   CRCLen;     // Length of data in bytes
}CRC_CALC;

CRC_CALC mycrc;
...
CRC16P2(&mycrc);
...
```

See also

VCRC16P2H_1 mem16
VCRC16P1H_1 mem16
VCRC16P1L_1 mem16
VCRC24H_1 mem16  

**CRC24, High Byte**

**Operands**

| mem16 | 16-bit memory location |

**Opcode**

LSW: 1110 0010 1100 1011  
MSW: 0000 0010 mem16

**Description**

This instruction uses CRC24 polynomial == 0x5D6DCB. Calculate the CRC24 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```c
if (VSTATUS[CRCMSGFLIP] == 0){
    temp[7:0] = [mem16][15:8];
}else {
    temp[7:0] = [mem16][8:15];
}
VCRC[23:0] = CRC24 (VCRC[23:0], temp[7:0])
```

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VCRC24L_1 mem16.

**See also**

VCRC24L_1 mem16
VCRC24L_1 mem16  —  CRC24, Low Byte

Operands

| mem16 | 16-bit memory location |

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 1100 1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0001 mem16</td>
</tr>
</tbody>
</table>

Description

This instruction uses CRC24 polynomial == 0x5D6DCB

Calculate the CRC24 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```c
if (VSTATUS[CRCMSGFLIP] == 0) {
    temp[7:0] = [mem16][7:0];
} else {
    temp[7:0] = [mem16][0:7];
}
VCRC[23:0] = CRC24 (VCRC[23:0], temp[7:0])
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```c
typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData; // Start of data
    uint16_t CRCLen; // Length of data in bytes
} CRC_CALC;

CRC_CALC mycrc;
...
CRC24(&mycrc);
...
;
---------------------
; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words
;
.align 2
.global _CRC24
_CRC24
    VCRCCLR ; Clear the result register
    MOV AL, */+XAR4[4] ; AL = CRCLen
    ASR AL, 2 ; AL = CRCLen/4
    SUBB AL, #1 ; AL = CRCLen/4 - 1
    MOVL XAR7, */+XAR4[2] ; XAR7 = &CRCData
    .align 2
    NOP ; Align RPTB to an odd address
    RPTB _CRC24_done, AL ; Execute block of code AL + 1 times
    VCRC24L_1 *XAR7 ; Calculate CRC for 4 bytes
    VCRC24H_1 *XAR7++ ; ...
    VCRC24L_1 *XAR7 ; ...
    VCRC24H_1 *XAR7++ ; ...
_CRC24_done
    MOVL XAR7, */+XAR4[0] ; XAR7 = &CRCResult
    VMOV32 **XAR7[0], VCRC ; Store the result
    LRETR ; return to caller
```

See also

VCRC24H_1 mem16
VCRC32H_1 mem16  CRC32, High Byte

Operands

| mem16 | 16-bit memory location |

Opcode

| LSW: 1110 0010 1100 0010 |
| MSW: 0000 0000 mem16 |

Description

This instruction uses CRC32 polynomial $1 == 0x04C11DB7$

Calculate the CRC32 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if (VSTATUS[CRCMSGFLIP] == 0){
  temp[7:0] = [mem16][15:8];
} else {
  temp[7:0] = [mem16][8:15];
}

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VCRC32L_1 mem16.

See also

VCRC32L_1 mem16
VCRC32L_1 mem16 CRC32, Low Byte

Operands

| mem16       | 16-bit memory location |

Opcode

| LSW: 1110 0010 1100 0001 |
| MSW: 0000 0000 mem16 |

Description

This instruction uses CRC32 polynomial 1 == 0x04C11DB7. Calculate the CRC32 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if (VSTATUS[CRCMSGFLIP] == 0) {
  temp[7:0] = [mem16][7:0];
} else {
  temp[7:0] = [mem16][0:7];
}

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

```c
typedef struct {
  uint32_t *CRCResult; // Address where result should be stored
  uint16_t *CRCData; // Start of data
  uint16_t CRCLen; // Length of data in bytes
}CRC_CALC;

CRC_CALC mycrc;
...
CRC32(&mycrc);
...
```

```assembly
; Clear the result register
VCRCCLR
MOV AL, *+XAR4[4] ; AL = CRCLen
ASR AL, 2 ; AL = CRCLen/4
SUBB AL, #1 ; AL = CRCLen/4 - 1
MOVL XAR7, *+XAR4[2] ; XAR7 = 4CRCDta
.align 2
NOP ; Align RPTB to an odd address
RPTB _CRC32_done, AL ; Execute block of code AL + 1 times
VCRC32L_1 *XAR7 ; Calculate CRC for 4 bytes
VCRC32H_1 *XAR7++; ; ...
VCRC32L_1 *XAR7; ; ...
VCRC32H_1 *XAR7++; ; ...
_CRC32_done
MOVL XAR7, *+XAR4[0] ; XAR7 = 4CRCResult
VMOV32 ++XAR7[0], VCRC ; Store the result
LRETR ; return to caller
```

See also

VCRC32H_1 mem16
VCRC32P2H_1 mem16  
**CRC32, Polynomial 2, High Byte**

### Operands

| mem16 | 16-bit memory location |

### Opcode

| LSW: 1110 0010 1100 1011 |
| MSW: 0000 0100 mem16 |

### Description

This instruction uses CRC32 polynomial == 0x1EDC6F41

Calculate the CRC32 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```
if (VSTATUS[CRCMSGFLIP] == 0){
  temp[7:0] = [mem16][15:8];
} else {
  temp[7:0] = [mem16][8:15];
}

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

### Flags

This instruction does not modify any flags in the VSTATUS register.

### Pipeline

This is a single-cycle instruction.

### Example

Refer to the example for VCRC32P2L_1 mem16.

### See also

VCRC32L_1 mem16
VCRC32P2L_1 mem16 — CRC32, Low Byte

### Operands

| mem16 | 16-bit memory location |

### Opcode

| LSW: 1110 0010 1100 1011 |
| MSW: 0000 0011 mem16 |

### Description

This instruction uses CRC32 polynomial == 0x04C11DB7

Calculate the CRC32 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.

```c
if (VSTATUS[CRCMSGFLIP] == 0){
    temp[7:0] = [mem16][7:0];
} else {
    temp[7:0] = [mem16][0:7];
}

VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])
```

### Flags

This instruction does not modify any flags in the VSTATUS register.

### Pipeline

This is a single-cycle instruction.

### Example

```c
typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData;  // Start of data
    uint16_t CRCLen;    // Length of data in bytes
}CRC_CALC;

CRC_CALC mycrc;
...
CRC32P2(&mycrc);
...

; -------------------
; Calculate the CRC of a block of data
; This function assumes the block is a multiple of 2 16-bit words
;
.global _CRC32P2
_CRC32P2
    VCRCCLR ; Clear the result register
    MOV AL, *+XAR4[4] ; AL = CRCLen
    ASR AL, 2 ; AL = CRCLen/4
    SUBB AL, #1 ; AL = CRCLen/4 - 1
    MOVL XAR7, *+XAR4[2] ; XAR7 = &CRCData
    .align 2
    NOP ; Align RPTB to an odd address
    RPTB _CRC32P2_done, AL ; Execute block of code AL + 1 times
    VCRC32P2L_1 *XAR7 ; Calculate CRC for 4 bytes
    VCRC32P2H_1 *XAR7++ ; ...
    VCRC32P2L_1 *XAR7 ; ...
    VCRC32P2H_1 *XAR7++ ; ...
    _CRC32P2_done
    MOVL XAR7, *+XAR4[0] ; XAR7 = &CRCResult
    VMOV32 **XAR7[0], VCRC ; Store the result
    LRET ; return to caller

See also

VCRC32P2H_1 mem16
VCRCLLR — Clear CRC Result Register

Operands

| mem16 | 16-bit memory location |

Opcode

LSW: 1110 0101 0010 0100

Description

Clear the VCRC register.

VCRC = 0x0000

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VCRC32L_1 mem16.

See also

VMOV32 mem32, VCRC
VMOV32 VCRC, mem32
### VMOV32 mem32, VCRC — Store the CRC Result Register

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>32-bit memory destination</td>
</tr>
<tr>
<td>VCRC</td>
<td>CRC result register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW: 1110 0010 0000 0110 MSW: 0000 0000 mem32</td>
<td>Store the VCRC register. ([\text{mem32}] = \text{VCRC})</td>
</tr>
</tbody>
</table>

#### Flags
This instruction does not modify any flags in the VSTATUS register.

#### Pipeline
This is a single-cycle instruction.

#### Example
```
VMOV32 VCRC, mem32
```

See also
- **VCRCCLR**
- **VMOV32 VCRC, mem32**
VMOV32 VCRC, mem32  Load the CRC Result Register

Operands

<table>
<thead>
<tr>
<th>mem32</th>
<th>32-bit memory source</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCRC</td>
<td>CRC result register</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 0110
MSW: 0000 0000 mem32

Description

Load the VCRC register.

VCRC = [mem32]

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VCRCLRR
VMOV32 mem32, VCRC
### 5.5.6 Deinterleaver Instructions

The instructions are listed alphabetically, preceded by a summary.

#### Table 5-15. Deinterleaver Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCLRDIVE — Clear DIVE bit in the VSTATUS Register</td>
<td>655</td>
</tr>
<tr>
<td>VDEC VRaL — 16-bit Decrement</td>
<td>656</td>
</tr>
<tr>
<td>VDEC VRaL</td>
<td></td>
</tr>
<tr>
<td>VINC VRaL — 16-bit Increment</td>
<td>658</td>
</tr>
<tr>
<td>VINC VRaL</td>
<td></td>
</tr>
<tr>
<td>VMOD32 VRaH, VRb, VRcH — Modulo Operation</td>
<td>660</td>
</tr>
<tr>
<td>VMOD32 VRaH, VRb, VRcH</td>
<td></td>
</tr>
<tr>
<td>VMOD32 VRaH, VRb, VRcL — Modulo Operation</td>
<td>662</td>
</tr>
<tr>
<td>VMOD32 VRaH, VRb, VRcL</td>
<td></td>
</tr>
<tr>
<td>VMOV16 VRaL, VRbH — 16-bit Register Move</td>
<td>664</td>
</tr>
<tr>
<td>VMOV16 VRaH, VRbL — 16-Bit Register Move</td>
<td>665</td>
</tr>
<tr>
<td>VMOV16 VRaH, VRbH — 16-Bit Register Move</td>
<td>666</td>
</tr>
<tr>
<td>VMPYADD VRa, VRaL, VRaH, VRbH — Multiply Add 16-Bit</td>
<td>668</td>
</tr>
<tr>
<td>VMPYADD VRa, VRaL, VRaH, VRbL — Multiply Add 16-bit</td>
<td>669</td>
</tr>
</tbody>
</table>
VCLRDIVE — Clear DIVE bit in the VSTATUS Register

VCLRDIVE

Operands: none

Opcode: LSW: 1110 0101 0010 0000

Description: Clear the DIVE (Divide by zero error) bit in the VSTATUS register.

Flags: This instruction clears the DIVE bit in the VSTATUS register

Pipeline: This is a single-cycle operation

Example:

See also:
## VDEC VRaL — 16-bit Decrement

<table>
<thead>
<tr>
<th>Operands</th>
<th>VRaL</th>
<th>Low word of a general purpose register: VR0L, VR1L,...VR7L. Cannot be VR8L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>LSW: 1110 0110 1111 0010</td>
<td>MSW: 0000 1011 0000 1aaa</td>
</tr>
<tr>
<td>Description</td>
<td>16-bit Increment</td>
<td>VRaL = VRaL - 1</td>
</tr>
<tr>
<td>Flags</td>
<td>This instruction does not affect any flags in the VSTATUS register</td>
<td></td>
</tr>
<tr>
<td>Pipeline</td>
<td>This is a single-cycle instruction</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>VDEC VR0L ; VR0L = VR0L - 1</td>
<td></td>
</tr>
<tr>
<td>See also</td>
<td>VINC VRaL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VINC VRaL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VDEC VRaL</td>
<td></td>
</tr>
</tbody>
</table>
VDEC VRaL || VMOV32 VRb, mem32  

16-bit Decrement with Parallel Load

Operands

<table>
<thead>
<tr>
<th>VRaL</th>
<th>Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRb</td>
<td>General purpose register: VR0, VR1....VR7. Cannot be VR8</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 0001  
MSW: 01bb baaa mem32

Description

16-bit Decrement with Parallel Load

VRaL = VRaL - 1  
VRb = [mem32]

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

VDEC VR0L || VMOV32 VR1, *+XAR3[4]

See also

VINC VRaL  
VDEC VRaL  
VINC VRaL || VMOV32 VRb, mem32
VINC VRaL — 16-bit Increment

Operands

VRaL Low word of a general purpose register: VR0L, VR1L,...,VR7L. Cannot be VR8L

Opcode

LSW: 1110 0110 1111 0010
MSW: 0000 1011 0000 0aaa

Description

16-bit Increment

VRaL = VRaL + 1

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

VINC VR0L ; VR0L = VR0L + 1

See also

VINC VRaL || VMOV32 VRb, mem32
VDEC VRaL
VDEC VRaL || VMOV32 VRb, mem32
## VINC VRaL || VMOV32 VRb, mem32  16-bit Increment with Parallel Load

### Operands

<table>
<thead>
<tr>
<th>VRaL</th>
<th>Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRb</td>
<td>General purpose register: VR0, VR1....VR7. Cannot be VR8</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 1000 0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 00bb baaa mem32</td>
</tr>
</tbody>
</table>

### Description

16-bit Increment with parallel load

\[ VRaL = VRaL + 1 \]
\[ VRb = \text{[mem32]} \]

### Flags

This instruction does not affect any flags in the VSTATUS register

### Pipeline

This is a single-cycle instruction

### Example

VINC VR0L || VMOV32 VR1, *+XAR3[4]

### See also

VINC VRaL
VDEC VRaL
VDEC VRaL || VMOV32 VRb, mem32
VMOD32 VRaH, VRb, VRcH — Modulo Operation

Operands

| VRaH | High word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |
| VRb  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRcH | High word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |

Opcode

LSW: 1110 0110 1000 0000
MSW: 0010 100a aabb bccc

Description

Modulo operation: 32-bit signed %16 bit unsigned

if(VRcH == 0x0){
  VSTATUS[DIVE] = 1
}else{
  VRAH = VRb % VRcH
}

Flags

This instruction modifies the following bits in the VSTATUS register:
• DIVE is set if VRcH is 0 i.e. a divide by zero error.

Pipeline

This is a 9p cycle instruction. No VMOD32 related instruction can be present in the delay slot of this instruction.

Example

VMOD32 VR5H, VR3, VR4H ; VR5H = VR3%VR4H = j
; compute j = (b * J - v * i) % n;
NOP ; D1
MOV *+XAR1[AR0], AL ; D2 Save previous Y(i+j*m)
NOP ; D3
NOP ; D4
MOV AL, *XAR4++ ; D5 AL = X(I) load X(I)
NOP ; D6
NOP ; D7
NOP ; D8
VMPYADD VR5, VR5L, VR5H, VR4H
; VR5 = VR5L + VR5H*VR4H
; = i + j*m compute i + j*m

See also

VMOD32 VRaH, VRb, VRcL
VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe
VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, Vre
VCLRDIVE
VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, VRe  

Modulo Operation with Parallel Move

Operands

| VRaH | High word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |
| VRb  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRcH | Low word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |
| VRd  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRe  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |

Opcode

LSW: 1110 0110 1111 0011
MSW: leee dddc ccbb baaa

Description

Modulo operation: 32-bit signed %16 bit unsigned
if(VRcL == 0x0){
    VSTATUS[DIVE] = 1
}else{
    VRaH = VRb % VRcH
}
VRd = VRe

Flags

This instruction modifies the following bits in the VSTATUS register:
• DIVE is set if VRcH is 0, that is, a divide by zero error.

Pipeline

This is a 9p/1 cycle instruction. The VMOD32 instruction takes 9p cycles while the VMOV32 operation completes in a single cycle. No VMOD32 related instruction can be present in the delay slot of this instruction.

Example

VMOD32 VR5H, VR3, VR4H ; VR5H = VR3%VR4H = j; VR0 = {J,I}
| VMOV32 VR0, VR6 ; compute j = (b * J - v * i) % n; load back saved J,I |
| VINC VR0L ; D1 VR1H = u, VR1L = a |
| VMOV32 VR1, **XAR3[4] ; increment I; load u, a |
| MOV +XAR1[AR0], AL ; D2 Save previous Y(i+j*m) |
| VCMPY VR3, VR2, VR1, VR0 ; D3 VR3 = a*I - u*J |
| ; compute a * I - u * J |
| VMOD32 VR1, **XAR3[2] ; D4/D1 VR1H = v, VR1L = b load v,b |
| MOV AL, *XAR4++ ; D5 AL = X(I) load X(I) |
| NOP ; D6 |
| VMOV32 VR6, VR0 ; D7 VR6 = {J,I} save current {J,I} |
| VMOD16 VR0L, *+XAR5[0] ; D8 VR0L = J load J |
| VMOD32 VR0H, VR3, VR4H ; VR0H = (VR3 % VR4H) = i |
| ; compute i = (a * I - u * J) % m; |

See also

VMOD32 VRaH, VRb, VRcH
VMOD32 VRaH, VRb, VRcL
VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe
VCLRDIVE
### VMOD32 VRaH, VRb, VRcL — Modulo Operation

#### Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaH</td>
<td>High word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H</td>
</tr>
<tr>
<td>VRb</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VRcL</td>
<td>Low word of a general purpose register: VR0L, VR1L,...VR7L. Cannot be VR8L</td>
</tr>
</tbody>
</table>

#### Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0110 1000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0010 011a aabb bccc</td>
</tr>
</tbody>
</table>

#### Description

Modulo operation: 32-bit signed %16 bit unsigned

```c
if(VRcL == 0x0){
    VSTATUS[DIVE] = 1
}else{
    VRaH = VRb % VRcL
}
```

#### Flags

This instruction modifies the following bits in the VSTATUS register:

- **DIVE** is set if VRcL is 0, that is, a divide by zero error.

#### Pipeline

This is a 9p cycle instruction. No VMOD32 related instruction can be present in the delay slot of this instruction.

#### Example

```assembly
VMOD32 VR5H, VR3, VR4L ; VR5H = VR3%VR4L = j
; compute j = (b * J - v * i) % n;
NOP ; D1
MOV *+XAR1[AR0], AL ; D2 Save previous Y(i+j*m)
NOP ; D3
NOP ; D4
MOV AL, *XAR4++ ; D5 AL = X(I) load X(I)
NOP ; D6
NOP ; D7
NOP ; D8
VMPYADD VR5, VR5L, VR5H, VR4H ; VR5 = VR5L + VR5H*VR4H
; = i + j*m compute i + j*m
```

#### See also

- VMOD32 VRaH, VRb, VRcH
- VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe
- VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, Vre
- VCLRDIVE
VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe — Modulo Operation with Parallel Move

Operands

| VRaH | High word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |
| VRb  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRcL | Low word of a general purpose register: VR0L, VR1L,...VR7L. Cannot be VR8L |
| VRd  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRe  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |

Opcode

LSW: 1110 0110 1111 0011  
MSW: 0eee dddc ccbb baaa

Description

Modulo operation: 32-bit signed %16 bit unsigned

```
if(VRcL == 0x0){
    VSTATUS[DIVE] = 1
}else{
    VRaH = VRb % VRcL
}
VRd = VRe
```

Flags

This instruction modifies the following bits in the VSTATUS register:

- DIVE is set if VRcH is 0, that is, a divide by zero error.

Pipeline

This is a 9p/1 cycle instruction. The VMOD32 instruction takes 9p cycles while the VMOV32 operation completes in a single cycle. No VMOD32 related instruction can be present in the delay slot of this instruction.

Example

```
VMOD32 VR5H, VR3, VR4L ; VR5H = VR3%VR4L = j; VR0 = {J,I}
|| VMOV32 VR0, VR6 ; compute j = (b * J - v * i) % n;
| | load back saved J,I
VINC VR0L ; D1 VR1H = u, VR1L = a
|| VMOV32 VR1, **XAR3[4] ; increment I; load u, a
| | MOV *XAR1[AR0], AL ; D2 Save previous Y(i+j*m)
VCMPY VR3, VR2, VR1, VR0 ; D3 VR3 = a*I - u*J
| | compute a * I - u * J
| | VMOD32 VR1, **XAR3[2] ; D4/D1 VR1H = v, VR1L = b load v,b
| | MOV AL, *XAR4++ ; D5 AL = X(I) load X(I)
| | NOP ; D6
| | VMOD32 VR6, VR0 ; D7 VR6 = {J,I} save current {J,I}
|| VMOD32 VR0H, VR3, VR4H ; VR0H = (VR3 % VR4H) = i
| | compute i = (a * I - u * J) % m;
```

See also

VMOD32 VRaH, VRb, VRcH  
VMOD32 VRaH, VRb, VRcL  
VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, Vre  
VCLRDIVE
VMOV16 VRaL, VRbH — 16-bit Register Move

Operands

<table>
<thead>
<tr>
<th>VRbH</th>
<th>High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaL</td>
<td>Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1111 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 1010 00bb baaa</td>
</tr>
</tbody>
</table>

Description

16-bit Register Move

VRaL = VRbH

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

VMOV16 VR5L, VR0H ; VR5L = VR0H

See also

VMOV16 VRaH, VRbL
VMOV16 VRaH, VRbH
VMOV16 VRaL, VRbL
VMOV16 VRaH, VRbL  16-Bit Register Move

Operands

| VRbL | Low word of a general purpose register: VR0L, VR1L,...VR7L. Cannot be VR8L |
| VRaH | High word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |

Opcode

LSW: 1110 0110 1111 0010
MSW: 0000 1010 01bb baaa

Description

16-bit Register Move

VRaH = VRbL

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

VMOV16 VR5H, VR0L ; VR5H = VR0L

See also

VMOV16 VRaL, VRbH
VMOV16 VRaH, VRbH
VMOV16 VRaL, VRbL
VMOV16 VRaH, VRbH  —  16-Bit Register Move

Operands

| VRbH  | High word of a general purpose register: VR0H, VR1H…VR7H. Cannot be VR8H |
| VRaH  | High word of a general purpose register: VR0H, VR1H…VR7H. Cannot be VR8H |

Opcode

LSW: 1110 0110 1111 0010
MSW: 0000 1010 10bb baaa

Description

16-bit Register Move

VRaH = VRbH

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

VMOV16 VR5H, VR0H ; VR5H = VR0H

See also

VMOV16 VRaL, VRbH
VMOV16 VRaH, VRbL
VMOV16 VRaL, VRbL
VMOV16 VRaL, VRbL 16-Bit Register Move

**Operands**

| VRbL | Low word of a general purpose register: VR0L, VR1L,...VR7L. Cannot be VR8L |
| VRaL | Low word of a general purpose register: VR0L, VR1L,...VR7L. Cannot be VR8L |

**Opcode**

LSW: 1110 0110 1111 0010  
MSW: 0000 1010 11bb baaa

**Description**

16-bit Register Move  
VRaL = VRbL

**Flags**

This instruction does not affect any flags in the VSTATUS register

**Pipeline**

This is a single-cycle instruction

**Example**

VMOV16 VR5L, VR0L ; VR5L = VR0L

**See also**

VMOV16 VRaL, VRbH  
VMOV16 VRaH, VRbL  
VMOV16 VRaH, VRbH
VMPYADD VRa, VRaL, VRaH, VRbH — Multiply Add 16-Bit

Operands

| VRbH | High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H |
| VRaH | Low word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H |
| VRaL | Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L |
| VRa  | General purpose register: VR0, VR1....VR7. Cannot be VR8 |

Opcode

| LSW: 1110 0110 1111 0010 |
| MSW: 0000 1100 00bb baaa |

Description

Performs p + q*r, where p,q, and r are 16-bit values

- If(VSTATUS[SAT] == 1) {
  - If(VSTATUS[RND] == 1) {
    VRa = rnd(sat(VRaL + VRaH * VRbH)>>VSTATUS[SHIFTR]);
  } else {
    VRa = sat(VRaL + VRaH * VRbH)>>VSTATUS[SHIFTR];
  }
} else { //VSTATUS[SAT] = 0
  - If(VSTATUS[RND] == 1) {
    VRa = rnd((VRaL + VRaH * VRbH)>>VSTATUS[SHIFTR]);
  } else {
    VRa = (VRaL + VRaH * VRbH)>>VSTATUS[SHIFTR];
  }
}

It should be noted that:

- VRaH*VRbH is represented as 32-bit temp value
- VRaL should be sign extended to 32-bit before performing add
- The add operation is a 32-bit operation

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow if 32-bit signed overflow is detected in the add operation.

Pipeline

This is a 2p cycle operation

Example

```
VMPYADD VR5, VR5L, VR5H, VR4H ; VR5 = VR5L + VR5H*VR4H
    ; = i + j*m compute i + j*m
NOP
    ; D1
```

See also

VMPYADD VRa, VRaL, VRaH, VRbL
VMPYADD VRa, VRaL, VRaH, VRbL  Multiply Add 16-bit

Operands

| VRbL | High word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |
| VRaH | Low word of a general purpose register: VR0H, VR1H,...VR7H. Cannot be VR8H |
| VRaL | Low word of a general purpose register: VR0L, VR1L,...VR7L. Cannot be VR8L |
| VRa  | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |

Opcode

| LSW: 1110 0110 1111 0010 |
| MSW: 0000 1100 01bb baaa |

Description

Performs \( p + q * r \), where \( p, q, \) and \( r \) are 16-bit values

\[
\text{If}(\text{VSTATUS}[\text{SAT}] = 1)\{
\quad \text{If}(\text{VSTATUS}[\text{RND}] = 1)\{
\quad\quad \text{VRa} = \text{rnd}((\text{VRaL} + \text{VRaH} \times \text{VRbL})\triangleright V\text{STATUS}[\text{SHIFTR}]);
\quad\quad \text{else}\{
\quad\quad\quad \text{VRa} = \text{sat}((\text{VRaL} + \text{VRaH} \times \text{VRbL})\triangleright V\text{STATUS}[\text{SHIFTR}]);
\quad\quad\}\}
\quad\text{else}\{ //\text{VSTATUS}[\text{SAT}] = 0
\quad\quad \text{If}(\text{VSTATUS}[\text{RND}] = 1)\{
\quad\quad\quad \text{VRa} = \text{rnd}((\text{VRaL} + \text{VRaH} \times \text{VRbL})\triangleright V\text{STATUS}[\text{SHIFTR}]);
\quad\quad\quad \text{else}\{
\quad\quad\quad\quad \text{VRa} = (\text{VRaL} + \text{VRaH} \times \text{VRbL})\triangleright V\text{STATUS}[\text{SHIFTR}];
\quad\quad\quad\}\}
\quad\}\}
\]

It should be noted that:
- \( \text{VRaH} \times \text{VRbL} \) is represented as 32-bit temp value
- \( \text{VRaL} \) should be sign extended to 32-bit before performing add
- The add operation is a 32-bit operation

Flags

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow if 32-bit signed overflow is detected in the add operation.

Pipeline

This is a 2p cycle operation

Example

VMPYADD VR5, VR5L, VR5H, VR4L ; VR5 = VR5L + VR5H*VR4L
\quad ; i + j*m compute \( i + j*m \)
NOP ; D1

See also

VMPYADD VRa, VRaL, VRaH, VRbH
## 5.5.7 FFT Instructions

The instructions are listed alphabetically, preceded by a summary.

### Table 5-16. FFT Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCFFT1 VR2, VR5, VR4 — Complex FFT calculation instruction</td>
<td>671</td>
</tr>
<tr>
<td>VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, #1-bit — Complex FFT calculation instruction</td>
<td>672</td>
</tr>
<tr>
<td>VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT3 VR5, VR4, VR3, VR2, VR0, #1-bit — Complex FFT calculation instruction</td>
<td>676</td>
</tr>
<tr>
<td>VCFFT3 VR5, VR4, VR3, VR2, VR0, #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT4 VR4, VR2, VR1, VR0, #1-bit — Complex FFT calculation instruction</td>
<td>680</td>
</tr>
<tr>
<td>VCFFT4 VR4, VR2, VR1, VR0, #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT5 VR5, VR4, VR3, VR2, VR1, VR0, #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT6 VR3, VR2, VR1, VR0, #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT7 VR1, VR0, #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT8 VR3, VR2, #1-bit — Complex FFT calculation instruction</td>
<td>688</td>
</tr>
<tr>
<td>VCFFT8 VR3, VR2, #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 #1-bit — Complex FFT calculation instruction</td>
<td>690</td>
</tr>
<tr>
<td>VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 #1-bit</td>
<td></td>
</tr>
<tr>
<td>VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 #1-bit — Complex FFT calculation instruction</td>
<td>693</td>
</tr>
<tr>
<td>VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 #1-bit</td>
<td></td>
</tr>
</tbody>
</table>
VCFFT1 VR2, VR5, VR4 — Complex FFT calculation instruction

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

| VR4 | First Complex Input |
| VR5 | Second Complex Input |
| VR2 | Complex Output |

Opcode
LSW: 1110 0101 0010 1011

Description
This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VR2H = rnd(sat((VR5H*VR4L - VR5L*VR4H)>>VSTATUS[SHIFTR]))
        VR2L = rnd(sat((VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR]))
    }else {
        VR2H = sat((VR5H*VR4L - VR5L*VR4H)>>VSTATUS[SHIFTR])
        VR2L = sat((VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR])
    }else {//VSTATUS[SAT] = 0
        If(VSTATUS[RND] == 1){
            VR2H = rnd((VR5H*VR4L - VR5L*VR4H)>>VSTATUS[SHIFTR])
            VR2L = rnd((VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR])
        }else {
            VR2H = (VR5H*VR4L - VR5L*VR4H)>>VSTATUS[SHIFTR])
            VR2L = (VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR])
        }
    }
}

Sign-Extension is automatically done for the shift right operations

Flags
This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

Pipeline
This is a two cycle instruction

Example
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also

VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, #1-bit  Complex FFT calculation instruction

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VR7</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR6</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR4</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR2</td>
<td>Complex Output</td>
</tr>
<tr>
<td>VR1</td>
<td>Complex Output</td>
</tr>
<tr>
<td>VR0</td>
<td>Complex Output</td>
</tr>
<tr>
<td>#1-bit</td>
<td>1-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1010 0001 0011 0001

Description
This operation is used in the butterfly operation of the FFT:

```c
If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VR0H = rnd(sat(VR7H + VR2H)>>#1-bit);
        VR0L = rnd(sat(VR7L + VR2L)>>#1-bit);
        VR1L = rnd(sat(VR7L - VR2L)>>#1-bit);
        VR1H = rnd(sat(VR7H - VR2H)>>#1-bit);
        VR2H = rnd(sat(VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
        VR2L = rnd(sat(VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
    }
    else {
        VR0H = sat(VR7H + VR2H)>>#1-bit;
        VR0L = sat(VR7L + VR2L)>>#1-bit;
        VR1L = sat(VR7L - VR2L)>>#1-bit;
        VR1H = sat(VR7H - VR2H)>>#1-bit;
        VR2H = sat(VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
        VR2L = sat(VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
    }
} else { //VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1){
        VR0H = rnd((VR7H + VR2H)>>#1-bit);
        VR0L = rnd((VR7L + VR2L)>>#1-bit);
        VR1L = rnd((VR7L - VR2L)>>#1-bit);
        VR1H = rnd((VR7H - VR2H)>>#1-bit);
        VR2H = rnd((VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
        VR2L = rnd((VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
    }
    else {
        VR0H = (VR7H + VR2H)>>#1-bit;
        VR0L = (VR7L + VR2L)>>#1-bit;
        VR1L = (VR7L - VR2L)>>#1-bit;
        VR1H = (VR7H - VR2H)>>#1-bit;
        VR2H = (VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
        VR2L = (VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
    }
}
```

Sign-Extension is automatically done for the shift right operations

Flags
This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit
VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, #1-bit — Complex FFT calculation instruction

temporary result can't fit in 16-bit destination

Pipeline
This is a two cycle instruction

Example
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, #1-bit || VMOV32 mem32, VR1

Complex FFT calculation instruction with Parallel Store

Operands

This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

| VR7 | Complex Input |
| VR6 | Complex Input |
| VR4 | Complex Input |
| VR2 | Complex Output |
| VR1 | Complex Output |
| VR0 | Complex Output |
| #1-bit | 1-bit immediate value |
| mem32 | Pointer to 32-bit memory location |

Opcode

LSW: 1110 0010 0000 0111
MSW: 0010 000I mem32

Description

This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR0H = rnd(sat(VR7H + VR2H)>>#1-bit);
    VR0L = rnd(sat(VR7L + VR2L)>>#1-bit);
    VR1L = rnd(sat(VR7L - VR2L)>>#1-bit);
    VR1H = rnd(sat(VR7H - VR2H)>>#1-bit);
    VR2H = rnd(sat(VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
    VR2L = rnd(sat(VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
  }else {
    VR0H = sat(VR7H + VR2H)>>#1-bit;
    VR0L = sat(VR7L + VR2L)>>#1-bit;
    VR1L = sat(VR7L - VR2L)>>#1-bit;
    VR1H = sat(VR7H - VR2H)>>#1-bit;
    VR2H = sat(VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
    VR2L = sat(VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
  }
}else {//VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR0H = rnd((VR7H + VR2H)>>#1-bit);
    VR0L = rnd((VR7L + VR2L)>>#1-bit);
    VR1L = rnd((VR7L - VR2L)>>#1-bit);
    VR1H = rnd((VR7H - VR2H)>>#1-bit);
    VR2H = rnd((VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
    VR2L = rnd((VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]));
  }else {
    VR0H = (VR7H + VR2H)>>#1-bit;
    VR0L = (VR7L + VR2L)>>#1-bit;
    VR1L = (VR7L - VR2L)>>#1-bit;
    VR1H = (VR7H - VR2H)>>#1-bit;
    VR2H = (VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
    VR2L = (VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
  }
}
[mem32] = VR1;

Sign-Extension is automatically done for the shift right operations
Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

Pipeline

This is a 2p/1-cycle instruction. The VCFFT operation takes 2p cycles and the VMOV operation completes in a single cycle.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT3 VR5, VR4, VR3, VR2, VR0, #1-bit—Complex FFT calculation instruction

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

| VR5 | Complex Input |
| VR4 | Complex Input |
| VR3 | Complex Output |
| VR2 | Complex Output/Complex Input from previous operation |
| VR0 | Complex Output/Complex Input from previous operation |
| #1-bit | 1-bit immediate value |

Opcode

LSW: 1010 0001 0011 0011

Description

This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR0H = rnd(sat(VR5H + VR2H)>>#1-bit);
    VR0L = rnd(sat(VR5L + VR2L)>>#1-bit);
    VR3H = rnd(sat(VR5H - VR2H)>>#1-bit);
    VR3L = rnd(sat(VR5L - VR2L)>>#1-bit);
    VR2H = rnd(sat(VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = rnd(sat(VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]);
  }else {
    VR0H = sat(VR5H + VR2H)>>#1-bit;
    VR0L = sat(VR5L + VR2L)>>#1-bit;
    VR3H = sat(VR5H - VR2H)>>#1-bit;
    VR3L = sat(VR5L - VR2L)>>#1-bit;
    VR2H = sat(VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = sat(VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]);
  }
}else {//VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR0H = rnd((VR5H + VR2H)>>#1-bit);
    VR0L = rnd((VR5L + VR2L)>>#1-bit);
    VR3H = rnd((VR5H - VR2H)>>#1-bit);
    VR3L = rnd((VR5L - VR2L)>>#1-bit);
    VR2H = rnd((VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]));
    VR2L = rnd((VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]));
  }else {
    VR0H = (VR5H + VR2H)>>#1-bit;
    VR0L = (VR5L + VR2L)>>#1-bit;
    VR3H = (VR5H - VR2H)>>#1-bit;
    VR3L = (VR5L - VR2L)>>#1-bit;
    VR2H = (VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = (VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]);
  }
}

Sign-Extension is automatically done for the shift right operations
Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

Pipeline

This is a 2p/1-cycle instruction. The VCFFT operation takes 2p cycles and the VMOV operation completes in a single cycle.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT3 VR5, VR4, VR3, VR2, VR0, #1-bit || VMOV32 VR5, mem32 — Complex FFT calculation instruction with Parallel Load

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

| VR5    | Complex Input                      |
| VR4    | Complex Input                      |
| VR3    | Complex Output                     |
| VR2    | Complex Output/Complex Input from previous operation |
| VR0    | Complex Output/Complex Input from previous operation |
| #1-bit | 1-bit immediate value               |
| mem32  | Pointer to 32-bit memory location  |

Opcode
LSW: 1110 0010 1011 0000
MSW: 0000 001I mem32

Description
This operation is used in the butterfly operation of the FFT:

```c
If(VSTATUS[SAT] == 1) {
    If(VSTATUS[RND] == 1) {
        VR0H = rnd(sat(VR5H + VR2H)>>#1-bit);
        VR0L = rnd(sat(VR5L + VR2L)>>#1-bit);
        VR3H = rnd(sat(VR5H - VR2H)>>#1-bit);
        VR3L = rnd(sat(VR5L - VR2L)>>#1-bit);
        VR2H = rnd(sat(VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]);
        VR2L = rnd(sat(VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]);
    } else {
        VR0H = sat(VR5H + VR2H)>>#1-bit;
        VR0L = sat(VR5L + VR2L)>>#1-bit;
        VR3H = sat(VR5H - VR2H)>>#1-bit;
        VR3L = sat(VR5L - VR2L)>>#1-bit;
        VR2H = sat(VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]);
        VR2L = sat(VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]);
    }
} else { //VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1) {
        VR0H = rnd((VR5H + VR2H)>>#1-bit);
        VR0L = rnd((VR5L + VR2L)>>#1-bit);
        VR3H = rnd((VR5H - VR2H)>>#1-bit);
        VR3L = rnd((VR5L - VR2L)>>#1-bit);
        VR2H = rnd((VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]));
        VR2L = rnd((VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]));
    } else {
        VR0H = (VR5H + VR2H)>>#1-bit;
        VR0L = (VR5L + VR2L)>>#1-bit;
        VR3H = (VR5H - VR2H)>>#1-bit;
        VR3L = (VR5L - VR2L)>>#1-bit;
        VR2H = (VR0H * VR4L - VR0L * VR4H)>>VSTATUS[SHIFTR]);
        VR2L = (VR0L * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]);
    }
}
VR5 = [mem32];
```

Sign-Extension is automatically done for the shift right operations.
### Flags
This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

### Pipeline
This is a 2p cycle instruction.

### Example
See the example for **VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit**

### See also
VCFFT4 VR4, VR2, VR1, VR0, #1-bit — Complex FFT calculation instruction

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR2</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>VR1</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>VR0</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>#1-bit</td>
<td>1-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1010 0001 0011 010I

Description
This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR0H = rnd(sat(VR0H + VR2H)>>#1-bit);
    VR0L = rnd(sat(VR0L + VR2L)>>#1-bit);
    VR1H = rnd(sat(VR0H - VR2H)>>#1-bit);
    VR1L = rnd(sat(VR0L - VR2L)>>#1-bit);
    VR2H = rnd(sat(VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = rnd(sat(VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]);
  }else {
    VR0H = sat(VR0H + VR2H)>>#1-bit;
    VR0L = sat(VR0L + VR2L)>>#1-bit;
    VR1H = sat(VR0H - VR2H)>>#1-bit;
    VR1L = sat(VR0L - VR2L)>>#1-bit;
    VR2H = sat(VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = sat(VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]);
  }
}else { //VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR0H = rnd((VR0H + VR2H)>>#1-bit);
    VR0L = rnd((VR0L + VR2L)>>#1-bit);
    VR1H = rnd((VR0H - VR2H)>>#1-bit);
    VR1L = rnd((VR0L - VR2L)>>#1-bit);
    VR2H = rnd((VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]));
    VR2L = rnd((VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]));
  }else {
    VR0H = (VR0H + VR2H)>>#1-bit;
    VR0L = (VR0L + VR2L)>>#1-bit;
    VR1H = (VR0H - VR2H)>>#1-bit;
    VR1L = (VR0L - VR2L)>>#1-bit;
    VR2H = (VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = (VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]);
  }
}

Sign-Extension is automatically done for the shift right operations
Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

Pipeline

This is a 2p cycle instruction.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT4 VR4, VR2, VR1, VR0, #1-bit || VMOV32 VR7, mem32 — Complex FFT calculation instruction with Parallel Load

Operands

This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR4</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR2</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>VR1</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>VR0</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>#1-bit</td>
<td>1-bit immediate value</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1011 0000
MSW: 0000 0101 mem32

Description

This operation is used in the butterfly operation of the FFT:

```c
If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VR0H = rnd(sat(VR0H + VR2H)>>#1-bit);
        VR0L = rnd(sat(VR0L + VR2L)>>#1-bit);
        VR1H = rnd(sat(VR0H - VR2H)>>#1-bit);
        VR1L = rnd(sat(VR0L - VR2L)>>#1-bit);
        VR2H = rnd(sat(VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]);
        VR2L = rnd(sat(VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]);
    }else {
        VR0H = sat((VR0H + VR2H)>>#1-bit);
        VR0L = sat((VR0L + VR2L)>>#1-bit);
        VR1H = sat((VR0H - VR2H)>>#1-bit);
        VR1L = sat((VR0L - VR2L)>>#1-bit);
        VR2H = sat((VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]);
        VR2L = sat((VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]);
    }
}else { //VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1){
        VR0H = rnd((VR0H + VR2H)>>#1-bit);
        VR0L = rnd((VR0L + VR2L)>>#1-bit);
        VR1H = rnd((VR0H - VR2H)>>#1-bit);
        VR1L = rnd((VR0L - VR2L)>>#1-bit);
        VR2H = rnd((VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]);
        VR2L = rnd((VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]);
    }else {
        VR0H = (VR0H + VR2H)>>#1-bit;
        VR0L = (VR0L + VR2L)>>#1-bit;
        VR1H = (VR0H - VR2H)>>#1-bit;
        VR1L = (VR0L - VR2L)>>#1-bit;
        VR2H = (VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR];
        VR2L = (VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR];
    }
}

VR7 = [mem32];
```

Sign-Extension is automatically done for the shift right operations.
Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

Pipeline

This is a 2p cycle instruction.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT5 VR5, VR4, VR3, VR2, VR1, VR0, #1-bit || VMOV32 mem32, VR1 — Complex FFT calculation instruction with Parallel Load

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

VR5 Complex Input
VR4 Complex Input
VR3 Complex Input
VR2 Complex Output/Complex Input from previous operation
VR1 Complex Output/Complex Input from previous operation
VR0 Complex Output/Complex Input from previous operation
#1-bit 1-bit immediate value
mem32 Pointer to 32-bit memory location

Opcode
LSW: 1110 0010 0000 0111
MSW: 0010 001I mem32

Description
This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR0H = rnd(sat(VR3H - VR2H)>>#1-bit);
    VR0L = rnd(sat(VR3L + VR2L)>>#1-bit);
    VR1H = rnd(sat(VR3H + VR2H)>>#1-bit);
    VR1L = rnd(sat(VR3L - VR2L)>>#1-bit);
    VR2H = rnd(sat(VR5H * VR4L - VR5L * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = rnd(sat(VR5L * VR4L + VR5H * VR4H)>>VSTATUS[SHIFTR]);
  }else {
    VR0H = sat(VR3H - VR2H)>>#1-bit;
    VR0L = sat(VR3L + VR2L)>>#1-bit;
    VR1H = sat(VR3H + VR2H)>>#1-bit;
    VR1L = sat(VR3L - VR2L)>>#1-bit;
    VR2H = sat(VR5H * VR4L - VR5L * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = sat(VR5L * VR4L + VR5H * VR4H)>>VSTATUS[SHIFTR]);
  }
}
else {//VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR0H = rnd((VR3H - VR2H)>>#1-bit);
    VR0L = rnd((VR3L + VR2L)>>#1-bit);
    VR1H = rnd((VR3H + VR2H)>>#1-bit);
    VR1L = rnd((VR3L - VR2L)>>#1-bit);
    VR2H = rnd((VR5H * VR4L - VR5L * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = rnd((VR5L * VR4L + VR5H * VR4H)>>VSTATUS[SHIFTR]));
  }else {
    VR0H = (VR3H - VR2H)>>#1-bit;
    VR0L = (VR3L + VR2L)>>#1-bit;
    VR1H = (VR3H + VR2H)>>#1-bit;
    VR1L = (VR3L - VR2L)>>#1-bit;
    VR2H = (VR5H * VR4L - VR5L * VR4H)>>VSTATUS[SHIFTR]);
    VR2L = (VR5L * VR4L + VR5H * VR4H)>>VSTATUS[SHIFTR]);
  }
}

[mem32] = VR1;

Sign-Extension is automatically done for the shift right operations
Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

Pipeline

This is a 2p cycle instruction.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT6 VR3, VR2, VR1, VR0, #1-bit || VMOV32 mem32, VR1

Complex FFT calculation instruction with Parallel Load

Operands

This operation assumes the following complex packing order for complex operands:

VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part

It ignores the VSTATUS[CPACK] bit.

VR3 Complex Input
VR2 Complex Output/Complex Input from previous operation
VR1 Complex Output/Complex Input from previous operation
VR0 Complex Output/Complex Input from previous operation
#1-bit 1-bit immediate value
mem32 Pointer to 32-bit memory location

Opcode

LSW: 1110 0010 0000 0111
MSW: 0010 010I mem32

Description

This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VR0H = rnd(sat(VR3H - VR2H)>>#1-bit);
        VR0L = rnd(sat(VR3L + VR2L)>>#1-bit);
        VR1H = rnd(sat(VR3H + VR2H)>>#1-bit);
        VR1L = rnd(sat(VR3L - VR2L)>>#1-bit);
    }else {
        VR0H = sat(VR3H - VR2H)>>#1-bit;
        VR0L = sat(VR3L + VR2L)>>#1-bit;
        VR1H = sat(VR3H + VR2H)>>#1-bit;
        VR1L = sat(VR3L - VR2L)>>#1-bit;
    }
}else {//VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1){
        VR0H = rnd((VR3H - VR2H)>>#1-bit);
        VR0L = rnd((VR3L + VR2L)>>#1-bit);
        VR1H = rnd((VR3H + VR2H)>>#1-bit);
        VR1L = rnd((VR3L - VR2L)>>#1-bit);
    }else {
        VR0H = (VR3H - VR2H)>>#1-bit;
        VR0L = (VR3L + VR2L)>>#1-bit;
        VR1H = (VR3H + VR2H)>>#1-bit;
        VR1L = (VR3L - VR2L)>>#1-bit;
    }
}
}[mem32] = VR1;

Sign-Extension is automatically done for the shift right operations

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline

This is a 1/1-cycle instruction. The VCFFT and VMOV operations are completed in one cycle.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT7 VR1, VR0, #1-bit || VMOV32 VR2, mem32  Complex FFT calculation instruction with Parallel Load

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part 
VRa[15:0] = Real Part 
It ignores the VSTATUS[CPACK] bit.

<table>
<thead>
<tr>
<th>VR3</th>
<th>Complex Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>VR1</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>VR0</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>#1-bit</td>
<td>1-bit immediate value</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0010 1011 0000
MSW: 0000 011I mem32

Description
This operation is used in the butterfly operation of the FFT:

```
If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VR0L = rnd(sat(VR0L + VR1L)>>#1-bit);
        VR0H = rnd(sat(VR0L - VR1L)>>#1-bit);
        VR1L = rnd(sat(VR0H + VR1H)>>#1-bit);
        VR1H = rnd(sat(VR0H - VR1H)>>#1-bit);
    }else {
        VR0L = sat(VR0L + VR1L)>>#1-bit;
        VR0H = sat(VR0L - VR1L)>>#1-bit;
        VR1L = sat(VR0H + VR1H)>>#1-bit;
        VR1H = sat(VR0H - VR1H)>>#1-bit;
    }
}else { //VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1){
        VR0L = rnd((VR0L + VR1L)>>#1-bit);
        VR0H = rnd((VR0L - VR1L)>>#1-bit);
        VR1L = rnd((VR0H + VR1H)>>#1-bit);
        VR1H = rnd((VR0H - VR1H)>>#1-bit);
    }else {
        VR0L = (VR0L + VR1L)>>#1-bit;
        VR0H = (VR0L - VR1L)>>#1-bit;
        VR1L = (VR0H + VR1H)>>#1-bit;
        VR1H = (VR0H - VR1H)>>#1-bit;
    }
}
VR2 = [mem32];
```

Sign-Extension is automatically done for the shift right operations

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline
This is a 1/1-cycle instruction. The VCFFT and VMOV operations are completed in one cycle.

Example
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT8 VR3, VR2, #1-bit  —  Complex FFT calculation instruction

Operands

This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

<table>
<thead>
<tr>
<th>VR2</th>
<th>Complex Output/Complex Input from previous operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3</td>
<td>Complex Output/Complex Input from previous operation</td>
</tr>
<tr>
<td>#1-bit</td>
<td>1-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1010 0001 0011 011I

Description

This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR2L = rnd(sat(VR2L + VR3L)>>#1-bit);
    VR2H = rnd(sat(VR2L - VR3L)>>#1-bit);
    VR3L = rnd(sat(VR2H + VR3H)>>#1-bit);
    VR3H = rnd(sat(VR2H - VR3H)>>#1-bit);
  }else {
    VR2L = sat(VR2L + VR3L)>>#1-bit;
    VR2H = sat(VR2L - VR3L)>>#1-bit;
    VR3L = sat(VR2H + VR3H)>>#1-bit;
    VR3H = sat(VR2H - VR3H)>>#1-bit;
  }
}else { //VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR2L = rnd((VR2L + VR3L)>>#1-bit);
    VR2H = rnd((VR2L - VR3L)>>#1-bit);
    VR3L = rnd((VR2H + VR3H)>>#1-bit);
    VR3H = rnd((VR2H - VR3H)>>#1-bit);
  }else {
    VR2L = (VR2L + VR3L)>>#1-bit;
    VR2H = (VR2L - VR3L)>>#1-bit;
    VR3L = (VR2H + VR3H)>>#1-bit;
    VR3H = (VR2H - VR3H)>>#1-bit;
  }
}

Sign-Extension is automatically done for the shift right operations

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline

This is a single cycle instruction.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT8 VR3, VR2, #1-bit || VOMV32 mem32, VR4  Complex FFT calculation instruction with Parallel Store

Operands

This operation assumes the following complex packing order for complex operands:

VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

| VR4               | Complex Input from previous operation |
| VR2               | Complex Output/Complex Input from previous operation |
| VR3               | Complex Output/Complex Input from previous operation |
| #1-bit            | 1-bit immediate value |
| mem32             | Pointer to 32-bit memory location |

Opcode

LSW: 1110 0010 0000 0111
MSW: 0010 011I mem32

Description

This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR2L = rnd(sat(VR2L + VR3L)>>#1-bit);
    VR2H = rnd(sat(VR2L - VR3L)>>#1-bit);
    VR3L = rnd(sat(VR2H + VR3H)>>#1-bit);
    VR3H = rnd(sat(VR2H - VR3H)>>#1-bit);
  }else {
    VR2L = sat(VR2L + VR3L)>>#1-bit;
    VR2H = sat(VR2L - VR3L)>>#1-bit;
    VR3L = sat(VR2H + VR3H)>>#1-bit;
    VR3H = sat(VR2H - VR3H)>>#1-bit;
  }
}else { //VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR2L = rnd((VR2L + VR3L)>>#1-bit);
    VR2H = rnd((VR2L - VR3L)>>#1-bit);
    VR3L = rnd((VR2H + VR3H)>>#1-bit);
    VR3H = rnd((VR2H - VR3H)>>#1-bit);
  }else {
    VR2L = (VR2L + VR3L)>>#1-bit;
    VR2H = (VR2L - VR3L)>>#1-bit;
    VR3L = (VR2H + VR3H)>>#1-bit;
    VR3H = (VR2H - VR3H)>>#1-bit;
  }
}[
[mem32] = VR4;

Sign-Extension is automatically done for the shift right operations

Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline

This is a single cycle instruction.

Example

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 #1-bit  —  Complex FFT calculation instruction

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

| VR0 | Complex Input |
| VR1 | Complex Input |
| VR2 | Complex Input |
| VR3 | Complex Input |
| VR4 | Complex Output |
| VR5 | Complex Output |
| #1-bit | 1-bit immediate value |

Opcode
LSW: 1010 0001 0011 100I

Description
This operation is used in the butterfly operation of the FFT:

```
If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VR4L = rnd(sat(VR0L + VR2L)>>#1-bit);
        VR4H = rnd(sat(VR1L + VR3L)>>#1-bit);
        VR5L = rnd(sat(VR0L - VR2L)>>#1-bit);
        VR5H = rnd(sat(VR1L - VR3L)>>#1-bit);
    }else {
        VR4L = sat(VR0L + VR2L)>>#1-bit;
        VR4H = sat(VR1L + VR3L)>>#1-bit;
        VR5L = sat(VR0L - VR2L)>>#1-bit;
        VR5H = sat(VR1L - VR3L)>>#1-bit;
    }
}else { //VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1){
        VR4L = rnd((VR0L + VR2L)>>#1-bit);
        VR4H = rnd((VR1L + VR3L)>>#1-bit);
        VR5L = rnd((VR0L - VR2L)>>#1-bit);
        VR5H = rnd((VR1L - VR3L)>>#1-bit);
    }else {
        VR4L = (VR0L + VR2L)>>#1-bit;
        VR4H = (VR1L + VR3L)>>#1-bit;
        VR5L = (VR0L - VR2L)>>#1-bit;
        VR5H = (VR1L - VR3L)>>#1-bit;
    }
}
```

Sign-Extension is automatically done for the shift right operations

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline
This is a single cycle instruction.

Example
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 #1-bit || VMOVE32 mem32, VR5 — Complex FFT calculation instruction with Parallel Store

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR1</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR2</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR3</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR4</td>
<td>Complex Output</td>
</tr>
<tr>
<td>VR5</td>
<td>Complex Output</td>
</tr>
<tr>
<td>#1-bit</td>
<td>1-bit immediate value</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0010 0000 0111
MSW: 0010 1001 mem32

Description
This operation is used in the butterfly operation of the FFT:

```
If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR4L = rnd(sat(VR0L + VR2L)>>#1-bit);
    VR4H = rnd(sat(VR1L + VR3L)>>#1-bit);
    VR5L = rnd(sat(VR0L - VR2L)>>#1-bit);
    VR5H = rnd(sat(VR1L - VR3L)>>#1-bit);
  }else {
    VR4L = sat(VR0L + VR2L)>>#1-bit;
    VR4H = sat(VR1L + VR3L)>>#1-bit;
    VR5L = sat(VR0L - VR2L)>>#1-bit;
    VR5H = sat(VR1L - VR3L)>>#1-bit;
  }
}else {//VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR4L = rnd((VR0L + VR2L)>>#1-bit);
    VR4H = rnd((VR1L + VR3L)>>#1-bit);
    VR5L = rnd((VR0L - VR2L)>>#1-bit);
    VR5H = rnd((VR1L - VR3L)>>#1-bit);
  }else {
    VR4L = (VR0L + VR2L)>>#1-bit;
    VR4H = (VR1L + VR3L)>>#1-bit;
    VR5L = (VR0L - VR2L)>>#1-bit;
    VR5H = (VR1L - VR3L)>>#1-bit;
  }
}
[mem32] = VR5;
```

Sign-Extension is automatically done for the shift right operations
Flags

This instruction modifies the following bits in the VSTATUS register:

- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline

This is a 1/1-cycle instruction. The VCFFT and VMOV operations are completed in one cycle.

Example

See the example for VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0, #1-bit

See also
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 #1-bit  Complex FFT calculation instruction

Operands

This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

<table>
<thead>
<tr>
<th>VR0</th>
<th>Complex Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR1</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR2</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR3</td>
<td>Complex Input</td>
</tr>
<tr>
<td>VR6</td>
<td>Complex Output</td>
</tr>
<tr>
<td>VR7</td>
<td>Complex Output</td>
</tr>
<tr>
<td>#1-bit</td>
<td>1-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1010 0001 0011 101I

Description

This operation is used in the butterfly operation of the FFT:

If(VSTATUS[SAT] == 1){
  If(VSTATUS[RND] == 1){
    VR6L = rnd(sat(VR0H + VR3H)>>#1-bit);
    VR6H = rnd(sat(VR1H - VR2H)>>#1-bit);
    VR7L = rnd(sat(VR0H - VR3H)>>#1-bit);
    VR7H = rnd(sat(VR1H + VR2H)>>#1-bit);
  }else {
    VR6L = sat(VR0H + VR3H)>>#1-bit;
    VR6H = sat(VR1H - VR2H)>>#1-bit;
    VR7L = sat(VR0H - VR3H)>>#1-bit;
    VR7H = sat(VR1H + VR2H)>>#1-bit;
  }
} else { //VSTATUS[SAT] = 0
  If(VSTATUS[RND] == 1){
    VR6L = rnd((VR0H + VR3H)>>#1-bit);
    VR6H = rnd((VR1H - VR2H)>>#1-bit);
    VR7L = rnd((VR0H - VR3H)>>#1-bit);
    VR7H = rnd((VR1H + VR2H)>>#1-bit);
  }else {
    VR6L = (VR0H + VR3H)>>#1-bit;
    VR6H = (VR1H - VR2H)>>#1-bit;
    VR7L = (VR0H - VR3H)>>#1-bit;
    VR7H = (VR1H + VR2H)>>#1-bit;
  }
}

Sign-Extension is automatically done for the shift right operations

Flags

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline

This is a single cycle instruction.

Example

_CFFT_run1024Pt:
  ...
  etc ...
  ...
  MOVL       *-SP[ARG_OFFSET], XAR4
  VSATON
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 #1-bit — Complex FFT calculation instruction

```assembly
.CFFT_run1024Pt_stages1and2Combined:
  MOVZ AR0, ++XAR4[NSAMPLES_OFFSET]
  MOVL XAR2, ++XAR4[INBUFFER_OFFSET]
  MOVL XAR1, ++XAR4[OUTBUFFER_OFFSET]

.lp_amode
SETC AMODE

NOP *,ARP2
VMOV32 VR0, *BR0++
VMOV32 VR1, *BR0++
VCFFT7 VR1, VR0, #1
|| VMOV32 VR2, *BR0++
VMOV32 VR3, *BR0++
VCFFT8 VR3, VR2, #1
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0, #1
.align 2
RPTB _CFFT_run1024Pt_stages1and2CombinedLoop, #S12_LOOP_COUNT

VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1
|| VMOV32 VR0, *BR0++
VMOV32 VR1, *BR0++
VCFFT7 VR1, VR0, #1
|| VMOV32 VR2, *BR0++
VMOV32 VR3, *BR0++
VCFFT8 VR3, VR2, #1
|| VMOV32 *XAR1++, VR4
VMOV32 *XAR1++, VR6
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0, #1
|| VMOV32 *XAR1++, VR5
VMOV32 *++, VR7, ARP2

._CFFT_run1024Pt_stages1and2CombinedLoop:
  VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1
  VMOV32 *XAR1++, VR4
  VMOV32 *XAR1++, VR6
  VMOV32 *XAR1++, VR5
  VMOV32 *XAR1++, VR7

._CFFT_run1024Pt_stages1and2CombinedEnd:
  .c28_amode
CLRC AMODE

._CFFT_run1024Pt_stages3and4Combined:
... etc ...
... VSETSHR #15
VRNDON
  MOVL XAR2, ++XAR4[S34_INPUT_OFFSET]
  MOVL XAR1, #S34_INSEP
  MOVL XAR0, #S34_OUTSEP
  MOVL XAR6, ++XAR4[S34_OUTPUT_OFFSET]
  MOVL XAR7, XAR6
  ADDB XAR7, #S34_GROUPSEP
  MOVL XAR3, #vcu2_twiddleFactors
```
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 #1-bit — Complex FFT calculation instruction

MOVL *-SP[TFPTR_OFFSET], XAR3
MOVL XAR4, XAR2
ADDB XAR4, #S34_GROUPSEP
MOVL XAR5, #S34_OUTER_LOOP_COUNT

_CFFT_run1024Pt_stages3and4OuterLoop:

MOVL XAR3, *-SP[TFPTR_OFFSET]

; Inner Butterfly Loop
VMOV32 VR5, *+XAR4[AR1]
VMOV32 VR6, *+XAR2[AR1]
VMOV32 VR7, *XAR4++
VCFFT1 VR2, VR5, VR4

VMOV32 VR5, *XAR2++
VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, #1

.align 2
RPTB _CFFT_run1024Pt_stages3and4InnerLoop, #S34_INNER_LOOP_COUNT
VMOV32 VR4, *XAR4++
VMOV32 VR5, VR4, VR3, VR2, VR0, #1
|| VMOV32 VR5, *+XAR4[AR1]

VMOV32 VR6, *+XAR2[AR1]
VCFFT4 VR4, VR2, VR1, VR0, #1
|| VMOV32 VR7, *XAR4++

VMOV32 VR4, *XAR3++
VMOV32 *XAR6++, VR0

VCFFT5 VR5, VR4, VR3, VR2, VR1, VR0, #1
|| VMOV32 *XAR7++, VR1

VMOV32 VR5, *XAR2++
VMOV32 *+XAR6[AR0], VR0

VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, #1
|| VMOV32 *+XAR7[AR0], VR1

_CFFT_run1024Pt_stages3and4InnerLoop:

VMOV32 VR4, *XAR3++
VCFFT3 VR5, VR4, VR3, VR2, VR0, #1

NOP
VCFFT4 VR4, VR2, VR1, VR0, #1

NOP
VMOV32 *XAR6++, VR0
VCFFT6 VR3, VR2, VR1, VR0, #1
|| VMOV32 *XAR7++, VR1

NOP
VMOV32 *+XAR6[AR0], VR0
VMOV32 *+XAR7[AR0], VR1

ADDB XAR2, #S34_POST_INCREMENT
ADDB XAR4, #S34_POST_INCREMENT
ADDB XAR6, #S34_POST_INCREMENT
ADDB XAR7, #S34_POST_INCREMENT

BANZ _CFFT_run1024Pt_stages3and4OuterLoop, AR5--

_CFFT_run1024Pt_stages3and4CombinedEnd:
See also

The entire FFT implementation, with accompanying code comments, can be found in the VCU Library in controlSUITE.
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 #1-bit || VMOV32 VR0, mem32 — Complex FFT calculation instruction with Parallel Load

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.

| VR0 | Complex Input |
| VR1 | Complex Input |
| VR2 | Complex Input |
| VR3 | Complex Input |
| VR6 | Complex Output |
| VR7 | Complex Output |
| #1-bit | 1-bit immediate value |
| mem32 | pointer to 32-bit memory location |

Opcode
LSW: 1110 0010 1011 0000
MSW: 0000 100I mem32

Description
This operation is used in the butterfly operation of the FFT:

```c
If(VSTATUS[SAT] == 1){
    If(VSTATUS[RND] == 1){
        VR6L = rnd(sat(VR0H + VR3H)>>#1-bit);
        VR6H = rnd(sat(VR1H - VR2H)>>#1-bit);
        VR7L = rnd(sat(VR0H - VR3H)>>#1-bit);
        VR7H = rnd(sat(VR1H + VR2H)>>#1-bit);
    }else {
        VR6L = sat(VR0H + VR3H)>>#1-bit;
        VR6H = sat(VR1H - VR2H)>>#1-bit;
        VR7L = sat(VR0H - VR3H)>>#1-bit;
        VR7H = sat(VR1H + VR2H)>>#1-bit;
    }
}else { //VSTATUS[SAT] = 0
    If(VSTATUS[RND] == 1){
        VR6L = rnd((VR0H + VR3H)>>#1-bit);
        VR6H = rnd((VR1H - VR2H)>>#1-bit);
        VR7L = rnd((VR0H - VR3H)>>#1-bit);
        VR7H = rnd((VR1H + VR2H)>>#1-bit);
    }else {
        VR6L = (VR0H + VR3H)>>#1-bit;
        VR6H = (VR1H - VR2H)>>#1-bit;
        VR7L = (VR0H - VR3H)>>#1-bit;
        VR7H = (VR1H + VR2H)>>#1-bit;
    }
}
VR0 = [mem32];
```

Sign-Extension is automatically done for the shift right operations

Flags
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline
This is a 1/1-cycle instruction. The VCFFT and VMOV operations are completed in one cycle.

Example
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, #1-bit
### 5.5.8 Galois Instructions

The instructions are listed alphabetically, preceded by a summary.

#### Table 5-17. Galois Field Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGFACC VRa, VRb, #4-bit — Galois Field Instruction</td>
<td>699</td>
</tr>
<tr>
<td>VGFACC VRa, VRb, VR7 — Galois Field Instruction</td>
<td>700</td>
</tr>
<tr>
<td>VGFACC VRa, VRb, VR7</td>
<td></td>
</tr>
<tr>
<td>VGFADD4 VRa, VRb, VRC, #4-bit — Galois Field Four Parallel Byte X Byte Add</td>
<td>702</td>
</tr>
<tr>
<td>VGFINIT mem16 — Initialize Galois Field Polynomial and Order</td>
<td>703</td>
</tr>
<tr>
<td>VGFMAC4 VRa, VRb, VRc — Galois Field Four Parallel Byte X Byte Multiply and Accumulate</td>
<td>704</td>
</tr>
<tr>
<td>VGMAC4 VRa, VRb, VRc</td>
<td></td>
</tr>
<tr>
<td>VGMAC4 VRa, VRb, VRc</td>
<td></td>
</tr>
<tr>
<td>VPACK4 VRa, mem32, #2-bit — Byte Packing</td>
<td>707</td>
</tr>
<tr>
<td>VREVB VRa — Byte Reversal</td>
<td>708</td>
</tr>
<tr>
<td>VSHLMB VRa, VRb — Shift Left and Merge Right Bytes</td>
<td>709</td>
</tr>
</tbody>
</table>
**VGFACC VRa, VRb, #4-bit  Galois Field Instruction**

### Operands

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRb</td>
<td>General purpose register: VR0, VR1....VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VRa</td>
<td>General purpose register: VR0, VR1....VR7. Cannot be VR8</td>
</tr>
<tr>
<td>#4-bit</td>
<td>4-bit Immediate Value</td>
</tr>
</tbody>
</table>

### Opcode

- LSW: 1110 0110 1000 0001
- MSW: 0000 00aa abbb IIII

### Description

Performs the following sequence of operations

- If (I[0:0] == 1 )
  
  VRa[7:0] = VRa[7:0] ^ VRb[7:0]

- If (I[1:1] == 1 )
  
  VRa[7:0] = VRa[7:0] ^ VRb[15:8]

- If (I[2:2] == 1 )
  
  VRa[7:0] = VRa[7:0] ^ VRb[23:16]

- If (I[3:3] == 1 )
  
  VRa[7:0] = VRa[7:0] ^ VRb[31:24]

### Flags

This instruction does not affect any flags in the VSTATUS register

### Pipeline

This is a single-cycle instruction

### Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

### See also

- VGFACC VRa, VRb, VR7
- VGFACC VRa, VRb, VR7 || VMOV32 VRc, mem32
VGFACC VRa, VRb, VR7 — Galois Field Instruction

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRb</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VRa</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VR7</td>
<td>General purpose register: VR7</td>
</tr>
</tbody>
</table>

Opcode

| LSW: 1110 0110 1000 0001 |
| MSW: 0000 0100 00aa abbb |

Description

Performs the following sequence of operations

If (VR7[0:0] == 1)

VRa[7:0] = VRa[7:0] ^ VRb[7:0]

If (VR7[1:1] == 1)

VRa[7:0] = VRa[7:0] ^ VRb[15:8]

If (VR7[2:2] == 1)

VRa[7:0] = VRa[7:0] ^ VRb[23:16]

If (VR7[3:3] == 1)

VRa[7:0] = VRa[7:0] ^ VRb[31:24]

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also

VGFACC VRa, VRb, #4-bit
VGFACC VRa, VRb, VR7 || VMOV32 VRc, mem32
VGFACC VRa, VRb, VR7 || VMOV32 VRc, mem32 — Galois Field Instruction with Parallel Load

Operands

- **VRb**: General purpose register: VR0, VR1,...,VR7. Cannot be VR8
- **VRa**: General purpose register: VR0, VR1,...,VR7. Cannot be VR8
- **VRc**: General purpose register: VR0, VR1,...,VR7. Cannot be VR8
- **VR7**: General purpose register: VR7
- **mem32**: Pointer to a 32-bit memory location

Opcode

LSW: 1110 0010 1011 011a
MSW: aabb bccc mem32

Description

Performs the following sequence of operations

- If (VR7[0:0] == 1)
  
  VRa[7:0] = VRa[7:0] ^ VRb[7:0]

- If (VR7[1:1] == 1)
  
  VRa[7:0] = VRa[7:0] ^ VRb[15:8]

- If (VR7[2:2] == 1)
  
  VRa[7:0] = VRa[7:0] ^ VRb[23:16]

- If (VR7[3:3] == 1)
  
  VRa[7:0] = VRa[7:0] ^ VRb[31:24]

  VRc = [mem32]

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a 1/1-cycle instruction. Both the VGFACC and VMOV32 operation complete in a single cycle.

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also

- VGFACC VRa, VRb, #4-bit
- VGFACC VRa, VRb, VR7
VGFADD4 VRa, VRb, VRc, #4-bit — Galois Field Four Parallel Byte X Byte Add

Operands

| VRb       | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRa       | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRc       | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| #4-bit    | 4-bit Immediate Value |

Opcode

| LSW: 1110 0110 1000 0000 |
| MSW: 000a aabb bccc IIII |

Description

Performs the following sequence of operations

If (I[0:0] == 1 )
  VRa[7:0] = VRb[7:0] ^ VRc[7:0]
else
  VRa[7:0] = VRb[7:0]

If (I[1:1] == 1 )
else
  VRa[15:8] = VRb[15:8]

If (I[2:2] == 1 )
else

If (I[3:3] == 1 )
else
  VRa[31:24] = VRb[31:24]

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single cycle instruction

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also
VGFINIT mem16

- Initialize Galois Field Polynomial and Order

Operands

| mem16 | Pointer to 16-bit memory location |

Opcode

LSW: 1110 0010 1100 0101
MSW: 0000 0000 mem16

Description

Initialize GF Polynomial and Order

VSTATUS[GF POLY] = [mem16][7:0]
VSTATUS[GF ORDER] = [mem16][10:8]

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also
VGFMAC4 VRa, VRb, VRc — Galois Field Four Parallel Byte X Byte Multiply and Accumulate

Operands

<table>
<thead>
<tr>
<th>VRb</th>
<th>General purpose register: VR0, VR1,...VR7. Cannot be VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRa</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VRc</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0010 001a aabb bccc</td>
</tr>
</tbody>
</table>

Description

Performs the follow sequence of operations:

VRa[7:0] = (VRa[7:0] * VRb[7:0]) ^ VRc[7:0]

The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also

VGFMPC4 VRa, VRb, VRc || VMOV32 VR0, mem32
VGFMPY4 VRa, VRb, VRc  

Galois Field Four Parallel Byte X Byte Multiply

Operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRb</td>
<td>General purpose register: VR0, VR1....VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VRa</td>
<td>General purpose register: VR0, VR1....VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VRc</td>
<td>General purpose register: VR0, VR1....VR7. Cannot be VR8</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1000 0000  
MSW: 0010 000a aabb bccc

Description

Performs the following sequence of operations:

\[
\begin{align*}
VRa[7:0] &= VRb[7:0] \times VRc[7:0] \\
\end{align*}
\]

The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single cycle instruction.

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE.

See also

VGFMPY4 VRa, VRb, VRc || VMOV32 VR0, mem32
VGFMPY4 VRa, VRb, VRc || VMOV32 VR0, mem32 — Galois Field Four Parallel Byte X Byte Multiply with Parallel Load

Operands

| VRb | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRa | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRc | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VR0 | General purpose register: VR0 |
| mem32 | Pointer to a 32-bit memory location |

Opcode

| LSW: 1110 0010 1011 010a |
| MSW: aabb bccc mem32 |

Description

Performs the following sequence of operations

- VRa[7:0] = VRb[7:0] * VRc[7:0]
- VR0 = [mem32]

The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a 1/1-cycle instruction. Both the VGFMPY4 and VMOV32 operation complete in a single cycle.

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also

VGFMPY4 VRa, VRb, VRc
VGFMAC4 VRa, VRb, VRc || PACK4 VR0, mem32, #2-bit — Galois Field Four Parallel Byte X Byte Multiply and Accumulate with Parallel Byte Packing

Operands

<table>
<thead>
<tr>
<th>VRb</th>
<th>General purpose register: VR0, VR1,...VR7. Cannot be VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRa</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VRc</td>
<td>General purpose register: VR0, VR1,...VR7. Cannot be VR8</td>
</tr>
<tr>
<td>VR0</td>
<td>General purpose register: VR0</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
<tr>
<td>#2-bit</td>
<td>2-bit Immediate Value</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 1011 1IIa</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: aabb bccc mem32</td>
</tr>
</tbody>
</table>

Description

Performs the follow sequence of operations:

VRa[7:0] = (VRa[7:0] * VRb[7:0]) ^ VRc[7:0]

If (I == 0)

VR0[7:0] = [mem32][7:0]
VR0[15:8] = [mem32][7:0]
VR0[23:16] = [mem32][7:0]
VR0[31:24] = [mem32][7:0]

Else If (I == 1)

VR0[7:0] = [mem32][15:8]
VR0[15:8] = [mem32][15:8]
VR0[23:16] = [mem32][15:8]
VR0[31:24] = [mem32][15:8]

Else If (I == 2)

VR0[7:0] = [mem32][23:16]
VR0[15:8] = [mem32][23:16]
VR0[23:16] = [mem32][23:16]
VR0[31:24] = [mem32][23:16]

Else If (I == 3)

VR0[7:0] = [mem32][31:24]
VR0[15:8] = [mem32][31:24]
VR0[23:16] = [mem32][31:24]
VR0[31:24] = [mem32][31:24]

The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a 1/1-cycle instruction. Both the VGFMAC4 and PACK4 operations complete in a single cycle.

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also
VPACK4 VRa, mem32, #2-bit  —  Byte Packing

Operands

<table>
<thead>
<tr>
<th>VRa</th>
<th>General purpose register: VR0, VR1,...,VR7. Cannot be VR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32</td>
<td>Pointer to a 32-bit memory location</td>
</tr>
<tr>
<td>#2-bit</td>
<td>2-bit Immediate Value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1011 0001
MSW: 000a aaII mem32

Description

Pack Ith byte from a memory location 4 times in VRa

If (I == 0)
    VRa[7:0] = [mem32][7:0]
    VRa[15:8] = [mem32][7:0]
    VRa[23:16] = [mem32][7:0]
    VRa[31:24] = [mem32][7:0]

Else If (I == 1)
    VRa[7:0] = [mem32][15:8]
    VRa[15:8] = [mem32][15:8]
    VRa[23:16] = [mem32][15:8]
    VRa[31:24] = [mem32][15:8]

Else If (I == 2)
    VRa[7:0] = [mem32][23:16]
    VRa[15:8] = [mem32][23:16]
    VRa[23:16] = [mem32][23:16]
    VRa[31:24] = [mem32][23:16]

Else If (I == 3)
    VRa[7:0] = [mem32][31:24]
    VRa[15:8] = [mem32][31:24]
    VRa[23:16] = [mem32][31:24]
    VRa[31:24] = [mem32][31:24]

The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags

This instruction does not affect any flags in the VSTATUS register

Pipeline

This is a single-cycle instruction

Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

See also
# VREVB VRa — Byte Reversal

## Operands

| VRa | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |

## Opcode

| LSW: 1110 0110 1000 0000 |
| MSW: 0010 0100 0000 0aaa |

## Description

Reverse Bytes

Input: VRa = \{B3,B2,B1,B0\}

Output: VRa = \{B0,B1,B2,B3\}

## Flags

This instruction does not affect any flags in the VSTATUS register

## Pipeline

This is a single-cycle instruction

## Example

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

## See also
### VSHLMB VRa, VRb — Shift Left and Merge Right Bytes

| VRa | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |
| VRb | General purpose register: VR0, VR1,...VR7. Cannot be VR8 |

**Opcode**

| LSW | 1110 0110 1000 0000 |
| MSW | 0010 0100 01aa abbb |

**Description**

Shift Left and Merge Bytes

- **Input**: VRa = \{B7,B6,B5,B4\}
- **Input**: VRb = \{B3,B2,B1,B0\}
- **Output**: VRa = \{B6,B5,B4,B3\}
- **Output**: VRb = \{B2,B1,B0,8'b0\}

**Restrictions**

VRa ≠ VRb. The source and destination registers must be different

**Flags**

This instruction does not affect any flags in the VSTATUS register

**Pipeline**

This is a single-cycle instruction

**Example**

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

**See also**

See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE
### 5.5.9 Viterbi Instructions

The instructions are listed alphabetically, preceded by a summary.

#### Table 5-18. Viterbi Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITBM2 VR0 — Code Rate 1:2 Branch Metric Calculation</td>
<td>712</td>
</tr>
<tr>
<td>VITBM2 VR0, mem32 — Branch Metric Calculation CR=1/2</td>
<td>713</td>
</tr>
<tr>
<td>VITBM2 VR0</td>
<td></td>
</tr>
<tr>
<td>VITBM3 VR0, VR1, VR2 — Code Rate 1:3 Branch Metric Calculation</td>
<td>715</td>
</tr>
<tr>
<td>VITBM3 VR0, VR1, VR2</td>
<td></td>
</tr>
<tr>
<td>VITBM3 VR0L, VR1L, mem16 — Branch Metric Calculation CR=1/3</td>
<td>717</td>
</tr>
<tr>
<td>VITDHSADDSUB VR4, VR3, VR2, VRa — Viterbi Double Add and Subtract, High</td>
<td>718</td>
</tr>
<tr>
<td>VITDHSADDSUB VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITDHSUBADD VR4, VR3, VR2, VRa — Viterbi Add and Subtract Low</td>
<td>721</td>
</tr>
<tr>
<td>VITDHSUBADD VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITDLADDSUB VR4, VR3, VR2, VRa — Viterbi Add and Subtract Low</td>
<td>723</td>
</tr>
<tr>
<td>VITDLADDSUB VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITDLSUBADD VR4, VR3, VR2, VRa — Viterbi Subtract and Add Low</td>
<td>725</td>
</tr>
<tr>
<td>VITDLSUBADD VR4, VR3, VR2, VRa</td>
<td></td>
</tr>
<tr>
<td>VITHSEL VRa, VRb, VR4, VR3 — Viterbi Select High</td>
<td>727</td>
</tr>
<tr>
<td>VITHSEL VRa, VRb, VR4, VR3</td>
<td></td>
</tr>
<tr>
<td>VITLSEL VRa, VRb, VR4, VR3 — Viterbi Select, Low Word</td>
<td>729</td>
</tr>
<tr>
<td>VITLSEL VRa, VRb, VR4, VR3</td>
<td></td>
</tr>
<tr>
<td>VITSTAGE — Parallel Butterfly Computation</td>
<td>732</td>
</tr>
<tr>
<td>VITSTAGE</td>
<td></td>
</tr>
<tr>
<td>VITSTAGE</td>
<td></td>
</tr>
<tr>
<td>VMOV32 VSM (k+1):VSM(k), mem32 — Load Consecutive State Metrics</td>
<td>736</td>
</tr>
<tr>
<td>VMOV32 mem32, VSM (k+1):VSM(k) — Store Consecutive State Metrics</td>
<td>737</td>
</tr>
<tr>
<td>VSETK #3-bit — Set Constraint Length for Viterbi Operation</td>
<td>738</td>
</tr>
<tr>
<td>VSMINIT mem16 — State Metrics Register initialization</td>
<td>739</td>
</tr>
<tr>
<td>VTCLEAR — Clear Transition Bit Registers</td>
<td>740</td>
</tr>
<tr>
<td>VTRACE mem32, VR0, VT0, VT1 — Viterbi Traceback, Store to Memory</td>
<td>741</td>
</tr>
<tr>
<td>VTRACE VR1, VR0, VT0, VT1 — Viterbi Traceback, Store to Register</td>
<td>743</td>
</tr>
<tr>
<td>VTRACE VR1, VR0, VT0, VT1</td>
<td></td>
</tr>
</tbody>
</table>
### VITBM2 VR0 — Code Rate 1:2 Branch Metric Calculation

#### Operands
Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit decoder input 1</td>
</tr>
</tbody>
</table>

The result of the operation is also stored in VR0 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR0H</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L - VR0L</td>
</tr>
</tbody>
</table>

#### Opcode
LSW: 1110 0101 0000 1100

#### Description
Branch metric calculation for code rate = 1/2.

```
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR0H is decoder input 1

// Calculate the branch metrics by performing 16-bit signed addition and subtraction
VR0L = VR0L + VR0H; // VR0L = branch metric 0
VR0H = VR0L - VR0L; // VR0H = branch metric 1
if (SAT == 1)
{
    sat16(VR0L);
    sat16(VR0H);
}
```

#### Flags
This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

#### Pipeline
This is a single-cycle instruction.

#### Example
See also

```c
VITBM2 VR0 || VMOV32 VR2, mem32
VITBM3 VR0, VR1, VR2
```
VITBM2 VR0, mem32  Branch Metric Calculation CR=1/2

Operands  
Before the operation, the inputs are loaded into the registers as shown below.

Opcode  
LSW: 1110 0010 1000 0000  
MSW: 0000 0001 mem16

Description  
Calculates two Branch-Metrics (BMs) for CR = ½

If(VSTATUS[SAT] == 1){  
  VR0L = sat([mem32][15:0] + [mem32][31:16]);  
  VR0H = sat([mem32][15:0] - [mem32][31:16]);
}else {  
  VR0L = [mem32][15:0] + [mem32][31:16];  
  VR0H = [mem32][15:0] - [mem32][31:16];
}

Flags  
This instruction modifies the following bits in the VSTATUS register:

• OVFR is set if overflow is detected in the computation of 16-bit signed result

Pipeline  
This is a single-cycle instruction.

Example  

; Viterbi K=4 CR = 1/2
;
; etc ...
;
VSETK #CONSTRAINT_LENGTH ; Set constraint length
MOV AR1, #SMETRICINIT_OFFSET
VSMINIT *[XAR4][AR1] ; Initialize the state metrics
MOV AR1, #NBITS_OFFSET
MOV AL, *[XAR4][AR1]
LSR AL, 2
SUBB AL, #2
MOV AR3, AL ; Initialize the BMSEL register
  ; for butterfly 0 to K-1
MOVL XAR6, *[XAR4][BMSELINIT_OFFSET]
VMOV32 VR2, *XAR6 ; Initialize BMSel for
  ; butterfly 0 to 7
VITBM2 VR0, *XAR0++ ; Calculate and store BMs in
  ; VR0L and VR0H
;
; etc ...

See also  
VITBM2 VR0
VITBM2 VR0 || VMOV32 VR2, mem32
VITSTAGE_VITBM2_VR0_mem32
VITBM2 VR0 || VMOV32 VR2, mem32 — Code Rate 1:2 Branch Metric Calculation with Parallel Load

**Operands**

Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit decoder input 1</td>
</tr>
<tr>
<td>[mem32]</td>
<td>pointer to 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR0 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR0H</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L - VR0L</td>
</tr>
<tr>
<td>VR2</td>
<td>contents of memory pointed to by [mem32]</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0011 1111 1100
MSW: 0000 0000 mem32

**Description**

Branch metric calculation for a code rate of 1/2 with parallel register load.

```c
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR0H is decoder input 1

// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction

VR0L = VR0L + VR0H; // VR0L = branch metric 0
VR0H = VR0L - VR0L; // VR0H = branch metric 1
if (SAT == 1) {
  sat16(VR0L);
  sat16(VR0H);
}
VR2 = [mem32] // Load VR2L and VR2H with the next state metrics
```

**Flags**

This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

**Pipeline**

Both operations complete in a single cycle.

**Example**

See also

VITBM2 VR0
VITBM3 VR0, VR1, VR2
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32
VITBM3 VR0, VR1, VR2  Code Rate 1:3 Branch Metric Calculation

**Operands**
Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit decoder input 1</td>
</tr>
<tr>
<td>VR2L</td>
<td>16-bit decoder input 2</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR0 and VR1 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR1L + VR2L</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L + VR1L - VR2L</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit branch metric 2 = VR0L - VR1L + VR2L</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit branch metric 3 = VR0L - VR1L - VR2L</td>
</tr>
</tbody>
</table>

**Opcode**
LSW: 1110 0101 0000 1101

**Description**
Calculate the four branch metrics for a code rate of 1/3.

```c
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR1L is decoder input 1
// VR2L is decoder input 2

// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction

VR0L = VR0L + VR1L + VR2L;  // VR0L = branch Metric 0
VR0H = VR0L + VR1L - VR2L;  // VR0H = branch Metric 1
VR1L = VR0L - VR1L + VR2L;  // VR1L = branch Metric 2
VR1H = VR0L - VR1L - VR2L;  // VR1H = branch Metric 3
if(SAT == 1)
{
    sat16(VR0L);
    sat16(VR0H);
    sat16(VR1L);
    sat16(VR1H);
}
```

**Flags**
This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

**Pipeline**
This is a 2p-cycle instruction. The instruction following VITBM3 must not use VR0 or VR1.

**Example**
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

**See also**
VITBM2 VR0  
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32  
VITBM2 VR0 || VMOV32 VR2, mem32
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32  —  Code Rate 1:3 Branch Metric Calculation with Parallel Load

Operands
Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit decoder input 0</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit decoder input 1</td>
</tr>
<tr>
<td>[mem32]</td>
<td>pointer to a 32-bit memory location</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR0 and VR1 and VR2 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>16-bit branch metric 0 = VR0L + VR1L + VR2L</td>
</tr>
<tr>
<td>VR0H</td>
<td>16-bit branch metric 1 = VR0L + VR1L - VR2L</td>
</tr>
<tr>
<td>VR1L</td>
<td>16-bit branch metric 2 = VR0L - VR1L + VR2</td>
</tr>
<tr>
<td>VR1H</td>
<td>16-bit branch metric 3 = VR0L - VR1L - VR2L</td>
</tr>
<tr>
<td>VR2</td>
<td>Contents of the memory pointed to by [mem32]</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0011 1111 1101
MSW: 0000 0000 mem32

Description
Calculate the four branch metrics for a code rate of 1/3 with parallel register load.

```c
// SAT is VSTATUS[SAT]
// VR0L is decoder input 0
// VR1L is decoder input 1
// VR2L is decoder input 2
//
// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction
//
// VR0L = VR0L + VR1L + VR2L; // VR0L = branch Metric 0
// VR0H = VR0L + VR1L - VR2L; // VR0H = branch Metric 1
// VR1L = VR0L - VR1L + VR2L; // VR1L = branch Metric 2
// VR1H = VR0L - VR1L - VR2L; // VR1H = branch Metric 3
if(SAT == 1)
{
    sat16(VR0L);
    sat16(VR0H);
    sat16(VR1L);
    sat16(VR1H);
}
VR2 = [mem32];
```

Flags
This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

Pipeline
This is a 2p/1-cycle instruction. The VBITM3 operation takes 2p cycles and the VMOV32 completes in a single cycle. The next instruction must not use VR0 or VR1.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also
VITBM2 VR0
VITBM2 VR0 || VMOV32 VR2, mem32
VITBM3 VR0L, VR1L, mem16  Branch Metric Calculation CR=1/3

Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>Low word of the general purpose register VR0</td>
</tr>
<tr>
<td>VR1L</td>
<td>Low word of the general purpose register VR1</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer to 16-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1100 0101
MSW: 0000 0010 mem16

Description

Calculates four Branch-Metrics (BM) for CR = 1/3

```c
If(VSTATUS[SAT] == 1){
    VR0L = sat(VR0L + VR1L + [mem16]);
    VR0H = sat(VR0L + VR1L - [mem16]);
    VR1L = sat(VR0L - VR1L + [mem16]);
    VR1H = sat(VR0L - VR1L - [mem16]);
}else {
    VR0L = VR0L + VR1L + [mem16];
    VR0H = VR0L + VR1L - [mem16];
    VR1L = VR0L - VR1L + [mem16];
    VR1H = VR0L - VR1L - [mem16];
}
```

Flags

This instruction modifies the following bits in the VSTATUS register.

- OVFR is set if overflow is detected in the computation of a 16-bit signed result

Pipeline

This is a single-cycle instruction.

Example

See the example for VITSTAGE || VMOV16 VROL, mem16

See also

VITBM3

VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32
VITDHADDSUB VR4, VR3, VR2, VRa — Viterbi Double Add and Subtract, High

Operands
Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaH.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaH</td>
<td>Branch metric 1. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaH</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0111 aaaa

Description
Viterbi high add and subtract. This instruction is used to calculate four path metrics.

// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaH with the branch metric.
//
// VR3L = VR2L + VRaH // Path metric 0
// VR3H = VR2H - VRaH // Path metric 1
// VR4L = VR2L - VRaH // Path metric 2
// VR4H = VR2H + VRaH // Path metric 3

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
;
; Example Viterbi decoder code fragment
; Viterbi butterfly calculations
; Loop once for each decoder input pair
;
; Branch metrics = BM0 and BM1
; XAR5 points to the input stream to the decoder
...
...
_loop:
  VMOV32 VR0, *XAR5++ ; Load two inputs into VR0L, VR0H
  VITBM2 VR0          ; VR0L = BM0    VR0H = BM1
  || VMOV32 VR2, *XAR1++ ; Load previous state metrics

  ...

; 2 cycle Viterbi butterfly
;
  VITDLADDSUB VR4,VR3,VR2,VR0 ; Perform add/sub
  VITLSEL VR6,VR5,VR4,VR3    ; Perform compare/select
  || VMOV32 VR2, *XAR1++ ; Load previous state metrics

; 2 cycle Viterbi butterfly, next stage
;
  VITDHADDSUB VR4,VR3,VR2,VR0
VITHSEL VR6, VR5, VR4, VR3
|| VMOV32 VR2, *XAR1++

; 2 cycle Viterbi butterfly, next stage
;
VITDLADDSUB VR4, VR3, VR2, VR0
|| VMOV32 *XAR2++, VR5
   ...
   ...

See also
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDHADDSUB VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Add and Subtract High with Parallel Store

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaH.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaH</td>
<td>Branch metric 1. VRa must be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Value to be stored. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaH</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 1001
MSW: bbbb aaaa mem32

Description

Viterbi high add and subtract. This instruction is used to calculate four path metrics.

```c
// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaH with the branch metric.
//
// VR3L = VR2L + VRaH // Path metric 0
// VR3H = VR2H - VRaH // Path metric 1
// VR4L = VR2L - VRaH // Path metric 2
// VR4H = VR2H + VRaH // Path metric 3
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VITDHSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa  

**Viterbi Add and Subtract Low**

**Operands**

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H + VRaL</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L + VRaL</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H - VRaL</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0101 1111 aaaa

**Description**

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

```
// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaL with the branch metric.
//
// VR3L = VR2L - VRaL  // Path metric 0
// VR3H = VR2H + VRaL  // Path metric 1
// VR4L = VR2L + VRaL  // Path metric 2
// VR4H = VR2H - VRaL  // Path metric 3
```

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

**See also**

- VITDHADDSUB VR4, VR3, VR2, VRa
- VITDHSUBADD VR4, VR3, VR2, VRa
- VITDLSUBADD VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Subtract and Add, High with Parallel Store

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaH.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaH</td>
<td>Branch metric 1. VRa must be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Contents to be stored. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

The result of the operation is stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1  = VR2H + VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2  = VR2L + VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3  = VR2H - VRaH</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 1011
MSW: bbbb aaaa mem32

Description

Viterbi high subtract and add. This instruction is used to calculate four path metrics.

```cpp
//
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaH with the branch metric.
//
// [mem32] = VRb // Store VRb to memory
VR3L = VR2L - VRaH // Path metric 0
VR3H = VR2H + VRaH // Path metric 1
VR4L = VR2L + VRaH // Path metric 2
VR4H = VR2H - VRaH // Path metric 3
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

See also

VITDHDSSUB VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
VITDSLADDSUB VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa  Viterbi Add and Subtract Low

Operands
Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaL</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0101 0011 aaaa

Description
This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

//
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaL with the branch metric.
//
VR3L = VR2L + VRaL  // Path metric 0
VR3H = VR2H - VRaL  // Path metric 1
VR4L = VR2L - VRaL  // Path metric 2
VR4H = VR2H + VRaL  // Path metric 3

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also
VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Add and Subtract Low with Parallel Load

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa can be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Contents to be stored to memory</td>
</tr>
</tbody>
</table>

The result of the operation is four path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H - VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H + VRaL</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 0000 1000
MSW: bbbb aaaa mem32

Description

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

```c
// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaL with the branch metric.
// [mem32] = VRb // Store VRb
VR3L = VR2L + VRaL // Path metric 0
VR3H = VR2H - VRaL // Path metric 1
VR4L = VR2L - VRaL // Path metric 2
VR4H = VR2H + VRaL // Path metric 3
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa  Viterbi Subtract and Add Low

Operands

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
</tbody>
</table>

The result of the operation is four path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H + VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H - VRaL</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1110 aaaa

Description

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

```
// Calculate the four path metrics by performing 16-bit signed addition and subtraction
// Before this operation VR2L and VR2H are loaded with the state metrics and VRaH with the branch metric.
//
VR3L = VR2L - VRaL // Path metric 0
VR3H = VR2H + VRaL // Path metric 1
VR4L = VR2L + VRaL // Path metric 2
VR4H = VR2H - VRaL // Path metric 3
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Subtract and Add, Low with Parallel Store

Operands
Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR2L</td>
<td>16-bit state metric 0</td>
</tr>
<tr>
<td>VR2H</td>
<td>16-bit state metric 1</td>
</tr>
<tr>
<td>VRaL</td>
<td>Branch metric 0. VRa must be VR0 or VR1.</td>
</tr>
<tr>
<td>VRb</td>
<td>Value to be stored. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0 = VR2L - VRaH</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1 = VR2H + VRaH</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2 = VR2L + VRaH</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3 = VR2H - VRaL</td>
</tr>
<tr>
<td>[mem32]</td>
<td>Contents of VRb. VRb can be VR5, VR6, VR7 or VR8.</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0010 0000 1010
MSW: bbbb aaaa mem32

Description
This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.

```
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaH with the branch metric.
//
// [mem32] = VRb     // Store VRb into mem32
// VR3L = VR2L - VRaL // Path metric 0
// VR3H = VR2H + VRaL // Path metric 1
// VR4L = VR2L + VRaL // Path metric 2
// VR4H = VR2H - VRaL // Path metric 3
```

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also
VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLADDSSUB VR4, VR3, VR2, VRa
VITHSEL VRa, VRb, VR4, VR3  Viterbi Select High

Operands

Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaH</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbH</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1111 0111
MSW: 0000 0000 bbbb aaaa

Description

This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16 bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITLSEL instruction.

```
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbH = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
}
else
{
    VRbH = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}
T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaH = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
}
else
{
    VRaH = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
```

Flags

This instruction does not modify any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

Example

Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also

VITLSEL VRa, VRb, VR4, VR3
VITHSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32

Viterbi Select High with Parallel Load

Operands
Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
<tr>
<td>[mem32]</td>
<td>pointer to 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaH</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbH</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VR2</td>
<td>Contents of the memory pointed to by [mem32].</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0011 1111 1111
MSW: bbbb aaaa mem32

Description
This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16-bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITLSEL instruction.

```c
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbH = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
}
else
{
    VRbH = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}

T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaH = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
}
else
{
    VRaH = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
VR2 = [mem32]; // Load VR2
```

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also
VITLSEL VRa, VRb, VR4, VR3
VITLSEL VRa, VRb, VR4, VR3  Viterbi Select, Low Word

Operands
Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaL</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbL</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0110 1111 0110
MSW: 0000 0000 bbbb aaaa

Description
This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16-bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITHSEL instruction.

```c
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbL = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
} else
{
    VRbL = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}

T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaL = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
} else
{
    VRaL = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
```

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

See also
VITHSEL VRa, VRb, VR4, VR3
VITLSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 — Viterbi Select Low with Parallel Load

Operands
Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR3L</td>
<td>16-bit path metric 0</td>
</tr>
<tr>
<td>VR3H</td>
<td>16-bit path metric 1</td>
</tr>
<tr>
<td>VR4L</td>
<td>16-bit path metric 2</td>
</tr>
<tr>
<td>VR4H</td>
<td>16-bit path metric 3</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location.</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRaL</td>
<td>16-bit state metric 0. VRa can be VR6 or VR8.</td>
</tr>
<tr>
<td>VRbL</td>
<td>16-bit state metric 1. VRb can be VR5 or VR7.</td>
</tr>
<tr>
<td>VT0</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VT1</td>
<td>The transition bit is appended to the end of the register.</td>
</tr>
<tr>
<td>VR2</td>
<td>Contents of 32-bit memory pointed to by mem32.</td>
</tr>
</tbody>
</table>

Opcode
LSW: 1110 0011 1111 1110
MSW: bbbb aaaa mem32

Description
This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16-bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITHSEL instruction. In parallel the VR2 register is loaded with the contents of memory pointed to by [mem32].

```
T0 = T0 <<< 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
    VRbL = VR3L; // New state metric 0
    T0[0:0] = 0; // Store the transition bit
}
else
{
    VRbL = VR3H; // New state metric 0
    T0[0:0] = 1; // Store the transition bit
}
T1 = T1 <<< 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
    VRaL = VR4L; // New state metric 1
    T1[0:0] = 0; // Store the transition bit
}
else
{
    VRaL = VR4H; // New state metric 1
    T1[0:0] = 1; // Store the transition bit
}
VR2 = [mem32]
```

Flags
This instruction does not modify any flags in the VSTATUS register.

Pipeline
This is a single-cycle instruction.

Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
VITLSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 — Viterbi Select Low with Parallel Load

See also

VITHSEL VRa, VRb, VR4, VR3
VITSTAGE — Parallel Butterfly Computation

VITSTAGE

Operands: None
Opcode: LSW: 1110 0101 0010 0110
Description: VITSTAGE instruction performs 32 viterbi butterflies in a single cycle. This instruction does the following:

• Depends on the Initial 64 State Metrics of the current stage stored in registers VSM0 to VSM63
• Depends on the Branch Metrics Select configuration stored in registers VR2 to VR5
• Depends on the Computed Branch Metrics of the current stage stored in registers VR0 and VR1
• Computes the State Metrics for the next stage and updates registers VSM0 to VSM63. The 16-bit signed result of the computation is saturated if VSTATUS[SAT] == 1
• Computes transition bits for all 64 states and updates registers VT0 and VT1

Flags: This instruction modifies the following bits in the VSTATUS register.

• OVFR is set if overflow is detected in the computation of a 16-bit signed result

Pipeline: This is a single-cycle instruction.

Example:

; Viterbi K=4 CR = 1/2
; etc ...
; VSETK #CONSTRAINT_LENGTH ; Set constraint length
MOV AR1, #SMETRICINIT_OFFSET
VSMINIT +*XAR4[AR1] ; Initialize the state metrics
MOV AR1, #NBITS_OFFSET
MOV VR0, *+XAR4[AR1]
LSR AL, 2
SUBB AL, #2
MOV AR3, AL ; Initialize the BMSEL register
; for butterfly 0 to K-1
MOV XAR6, *+XAR4[BMSELINIT_OFFSET]
VMOV32 VR2, *+XAR6 ; Initialize BMSEL for
; butterfly 0 to 7
VITBM2 VR0, *+XAR0++ ; Calculate and store BMs in
; VR0L and VR0H
.align 2
RPTB _VITERBI_runK4CR12_stageAandB, AR3
_VITERBI_runK4CR12_stageA:
VITSTAGE ; Compute NSTATES/2 butterflies
; in parallel,
VITBM2 VR0, *XAR0++ ; compute branch metrics for
; next butterfly
VMOV32 *XAR2++, VT1 ; Store VT1
VMOV32 *XAR2++, VT0 ; Store VT0
; etc ...

See also:

VITSTAGE || VITBM2 VR0, mem32
VITSTAGE || VMOV16 VROL, mem16
VITSTAGE || VITBM2 VR0, mem32  Parallel Butterfly Computation with Parallel Branch Metric Calculation  

CR=1/2

Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>Destination register</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1000 0000
MSW: 0000 0010 mem32

Description

VITSTAGE instruction performs 32 viterbi butterflies in a single cycle. This instruction does the following:

- Depends on the Initial 64 State Metrics of the current stage stored in registers VSM0 to VSM63
- Depends on the Branch Metrics Select configuration stored in registers VR2 to VR5
- Depends on the Computed Branch Metrics of the current stage stored in registers VR0 and VR1
- Computes the State Metrics for the next stage and updates registers VSM0 to VSM63. The 16-bit signed result of the computation is saturated if VSTATUS[SAT] == 1
- Computes transition bits for all 64 states and updates registers VT0 and VT1

\[
VR0L = [\text{mem32}[15:0]] + [\text{mem32}[31:16]] \\
VR0H = [\text{mem32}[15:0]] - [\text{mem32}[31:16]]
\]

Flags

This instruction modifies the following bits in the VSTATUS register.

- OVFR is set if overflow is detected in the computation of a 16-bit signed result

Pipeline

This is a single-cycle instruction.

Example

```c
; Viterbi K=4 CR = 1/2
;
; etc ...

VSETK  #CONSTRAINT_LENGTH ; Set constraint length
MOV    AR1,  #SMETRICINIT_OFFSET
VSMINIT  *+XAR4[AR1]  ; Initialize the state metrics
MOV    AR1,  #NBITS_OFFSET
MOV    AL,  **XAR4[AR1]
LSR    AL,  2
SUBB   AL,  #2
MOV    AR3,  AL  ; Initialize the BMSEL register
                ; for butterfly 0 to K-1
MOV    XAR6,  **XAR4[BMSELINIT_OFFSET]
VMOV32  VR2,  *XAR6  ; Initialize BMSEL for
                    ; butterfly 0 to 7
VITBM2  VR0,  *XAR0++  ; Calculate and store BMs in
                   ; VR0L and VR0H

.align 2
RPTB   _VITERBI_runK4CR12_stageAandB, AR3
_VITERBI_runK4CR12_stageA:  
    VITSTAGE  ; Compute NSTATES/2 butterflies
               ; in parallel,
    VITBM2  VR0,  *XAR0++  ; compute branch metrics for
               ; next butterfly
    VMOV32  *XAR2++,  VT1  ; Store VT1
    VMOV32  *XAR2++,  VT0  ; Store VT0
```

Copyright © 2014–2019, Texas Instruments Incorporated
See also

VITSTAGE

VITSTAGE || VMOV16 VROL, mem16
VITSTAGE || VMOV16 VR0L, mem1  Parallel Butterfly Computation with Parallel Load

Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0L</td>
<td>Low word of the destination register</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer to 16-bit memory location</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0010 1100 0101
MSW: 0000 0011 mem16

Description

VITSTAGE instruction performs 32 viterbi butterflies in a single cycle. This instruction does the following:

- Depends on the Initial 64 State Metrics of the current stage stored in registers VSM0 to VSM63
- Depends on the Branch Metrics Select configuration stored in registers VR2 to VR5
- Depends on the Computed Branch Metrics of the current stage stored in registers VR0 and VR1
- Computes the State Metrics for the next stage and updates registers VSM0 to VSM63. The 16-bit signed result of the computation is saturated if VSTATUS[SAT] == 1
- Computes transition bits for all 64 states and updates registers VT0 and VT1

VR0L = [mem16]

Flags

This instruction modifies the following bits in the VSTATUS register.
- OVFR is set if overflow is detected in the computation of a 16-bit signed result

Pipeline

This is a single-cycle instruction.

Example

; Viterbi K=7 CR = 1/3
; etc ...
_VITERBI_runK7CR13_stageA:
    VITSTAGE ; Compute NSTATES/2 butterflies in parallel,
    | | VMOV16 VR0L, *XAR0++ ; Load LLR(A) for next butterfly
    | VMOV16 VR1L, *XAR0++ ; Load LLR(B) for next butterfly
    | VITBM3 VR0L, VR1L, *XAR0++ ; Load LLR(C) and compute branch metric for next butterfly
    | VMOV32 *XAR2++, VT1 ; Store VT1
    | VMOV32 *XAR2++, VT0 ; Store VT0
    ; etc ...

See also

VITSTAGE
VITSTAGE || VITBM2 VR0, mem32
## VMOV32 VSM (k+1):VSM(k), mem32 — Load Consecutive State Metrics

### Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSM(k+1):VSM(k)</td>
<td>Consecutive State Metric Registers (VSM1:VSM0 ... VSM63:VSM62)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

### Opcode

LSW: 1110 0010 1000 0000  
MSW: 001n nnnn mem32

### Description

Load a pair of Consecutive State Metrics from memory:

\[
\begin{align*}
VSM(k+1) &= [\text{mem32}]_{31:16}; \\
VSM(k)  &= [\text{mem32}]_{15:0};
\end{align*}
\]

**Note:**
- \( n-k/2 \), used in opcode assignment
- \( k \) is always even

### Flags

This instruction does not affect any flags in the VSTATUS register.

### Pipeline

This is a single-cycle instruction.

### Example

\`\text{VMOV32 VSM63: VSM62, \*XAR7++} ` 

### See also

\`\text{VMOV32 mem32, VSM (k+1):VSM(k)} `
VMOV32 mem32, VSM (k+1):VSM(k)  

**Store Consecutive State Metrics**

### Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSM(k+1):VSM(k)</td>
<td>Consecutive State Metric Registers (VSM1:VSM0 .... VSM63:VSM62)</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

### Opcode

LSW: 1110 0010 0000 1110  
MSW: 000n nnnn mem32  

### Description

Store a pair of Consecutive State Metrics from memory:  

\[
\begin{align*}
&[\text{mem32}][31:16] = \text{VSM}(k+1); \\
&[\text{mem32}][15:0] = \text{VSM}(k);
\end{align*}
\]

### NOTE:

- n-k/2, used in opcode assignment  
- k is always even

### Flags

This instruction does not affect any flags in the VSTATUS register.

### Pipeline

This is a single-cycle instruction.

### Example

VMOV32 *XAR7++ VSM63: VSM62

### See also

VMOV32 VSM (k+1):VSM(k), mem32
VSETK #3-bit — Set Constraint Length for Viterbi Operation

Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>#3-bit</td>
<td>3-bit immediate value</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0110 1111 0010  
MSW: 0000 1001 0000 0III

Description

VSTATUS[K] = #3-bit Immediate

Flags

This instruction does not affect any flags in the VSTATUS register.

Pipeline

This is a single-cycle instruction.

See also
VSMINIT mem16  —  State Metrics Register initialization

**Operands**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem16</td>
<td>Pointer to 16-bit memory location</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1111 0010 1100 0101  
MSW: 0000 0001 mem16

**Description**

Initializes the state metric registers.

VSM0 = 0  
VSM1 to VSM63 = [mem16]

**Flags**

This instruction does not affect any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

VSMINIT *+[XAR4][AR1] ; Initialize the state metrics

**See also**
# VTCLEAR

**Clear Transition Bit Registers**

<table>
<thead>
<tr>
<th><strong>Operands</strong></th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opcode</strong></td>
<td>LSW: 1110 0101 0010 1001</td>
</tr>
</tbody>
</table>
| **Description** | Clear the VT0 and VT1 registers.  
VT0 = 0;  
VT1 = 0; |
| **Flags**     | This instruction does not modify any flags in the VSTATUS register. |
| **Pipeline**  | This is a single-cycle instruction. |
| **Example**   | |
| **See also**  | VCLEARALL  
VCLEAR VRa |
VTRACE mem32, VR0, VT0, VT1  *Viterbi Traceback, Store to Memory*

**Operands**

Before the operation, the path metrics are loaded into the registers as shown below using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT0</td>
<td>transition bit register 0</td>
</tr>
<tr>
<td>VT1</td>
<td>transition bit register 1</td>
</tr>
<tr>
<td>VR0</td>
<td>Initial value is zero. After the first VTRACE, this contains information from the previous trace-back.</td>
</tr>
</tbody>
</table>

The result of the operation is the new state metrics stored in VRa and VRb as shown below:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[mem32]</td>
<td>Traceback result from the transition bits.</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0010 0000 1100  
MSW: 0000 0000 mem32

**Description**

Trace-back from the transition bits stored in VT0 and VT1 registers. Write the result to memory. The transition bits in the VT0 and VT1 registers are stored in the following format by the VITLSEL and VITHSEL instructions:

- VT0[31]: Transition bit [State 0]
- VT0[30]: Transition bit [State 1]
- VT0[29]: Transition bit [State 2]
- ...
- VT0[0]: Transition bit [State 31]
- VT1[31]: Transition bit [State 32]
- VT1[30]: Transition bit [State 33]
- VT1[29]: Transition bit [State 34]
- ...
- VT1[0]: Transition bit [State 63]

```c
// Calculate the decoder output bit by performing a traceback from the transition bits stored in the VT0 and VT1 registers
//
K = VSTATUS[K];
S = VR0[K-2:0];
VR0[31:K-1] = 0;
if (S < (1<<(K-2))){
    temp[0] = VT0[(1 << (K-2))- 1 -S];
}else{
    temp[0] = VT1[(1 << (K-1))- 1 -S];
}
[mem32][0] = temp;
[mem32][31:1] = 0;
VR0[K-2:0] = 2*VR0[K-2:0] + temp[0];
```

**Flags**

This instruction does not modify any flags in the VSTATUS register.

**Pipeline**

This is a single-cycle instruction.

**Example**

```c
// Example traceback code fragment
```
//
// XAR5 points to the beginning of Decoder Output array
//
VCLEAR VR0
MOVL XAR5,*+XAR4[0]

//
// To retrieve each original message:
// Load VT0/VT1 with the stored transition values
// and use VTRACE instruction
//
VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE *XAR5++, VR0, VT0, VT1

VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE *XAR5++, VR0, VT0, VT1
...
...etc for each VT0/VT1 pair

See also
VTRACE VR1, VR0, VT0, VT1
VTRACE VR1, VR0, VT0, VT1  

**Viterbi Traceback, Store to Register**

## Operands

Before the operation, the path metrics are loaded into the registers as shown below using a Viterbi AddSub or SubAdd instruction.

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT0</td>
<td>transition bit register 0</td>
</tr>
<tr>
<td>VT1</td>
<td>transition bit register 1</td>
</tr>
<tr>
<td>VR0</td>
<td>Initial value is zero. After the first VTRACE, this contains information from the previous trace-back.</td>
</tr>
</tbody>
</table>

The result of the operation is the output of the decoder stored in VR1:

<table>
<thead>
<tr>
<th>Output Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR1</td>
<td>Traceback result from the transition bits.</td>
</tr>
</tbody>
</table>

## Opcode

LSW: 1110 0101 0010 1000

## Description

Trace-back from the transition bits stored in VT0 and VT1 registers. Write the result to VR1. The transition bits in the VT0 and VT1 registers are stored in the following format by the VITLSEL and VITHSEL instructions:

<table>
<thead>
<tr>
<th>VT0[31]</th>
<th>Transition bit [State 0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT0[30]</td>
<td>Transition bit [State 1]</td>
</tr>
<tr>
<td>VT0[29]</td>
<td>Transition bit [State 2]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>VT0[0]</td>
<td>Transition bit [State 31]</td>
</tr>
<tr>
<td>VT1[31]</td>
<td>Transition bit [State 32]</td>
</tr>
<tr>
<td>VT1[30]</td>
<td>Transition bit [State 33]</td>
</tr>
<tr>
<td>VT1[29]</td>
<td>Transition bit [State 34]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>VT1[0]</td>
<td>Transition bit [State 63]</td>
</tr>
</tbody>
</table>

```c
// Calculate the decoder output bit by performing a traceback from the transition bits stored in the VT0 and VT1 registers

K = VSTATUS[K];
S = VR0[K-2:0];
VR0[31:K-1] = 0;
if (S < (1<<(K-2))) {
    temp[0] = VT0[(1<<(K-2))- 1 -S];
} else {
    temp[0] = VT1[(1<<(K-1))- 1 -S];
}
if(VSTATUS[OPACK]==0){
    VR1 = VR1<<1;
    VR1[0:0] = temp[0] ;
    VR0[K-2:0] = 2*VR0[K-2:0] + temp[0];
} else{
    VR1 = VR1>>1
    VR1[31:31] = temp[0] ;
    VR0[K-2:0] = 2*VR0[K-2:0] + temp[0];
}
```

## Flags

This instruction does not modify any flags in the VSTATUS register.

## Pipeline

This is a single-cycle instruction.
VTRACE VR1, VR0, VT0, VT1 — Viterbi Traceback, Store to Register

Example

See also

VTRACE mem32, VR0, VT0, VT1
VTRACE VR1, VR0, VT0, VT1 || VMOV32 VT0, mem32  Trace-back with Parallel Load

### Operands

<table>
<thead>
<tr>
<th>Input Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT0</td>
<td>Traceback register</td>
</tr>
<tr>
<td>VT1</td>
<td>Traceback register</td>
</tr>
<tr>
<td>VR0</td>
<td>Decoded output bits register</td>
</tr>
<tr>
<td>VR1</td>
<td>Decoded output bits register</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer to 32-bit memory location</td>
</tr>
</tbody>
</table>

### Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0010 1011 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0001 mem32</td>
</tr>
</tbody>
</table>

### Description

Trace-back with Parallel Load

\[ K = \text{VSTATUS}[K]; \]
\[ S = \text{VR0}[(K-2):0]; \text{VR0}[31:K-1] = 0; \]
\[
\text{if (} S < (1 << (K-2)))
\]
\[ \quad \text{temp}[0] = \text{VT0}[(1<<(K-2))- 1 - S]; \]
\[
\text{else}
\]
\[ \quad \text{temp}[0] = \text{VT1}[(1<<(K-1))- 1 - S]; \]
\[
\text{if(VSTATUS[OPACK]==0)}
\]
\[ \quad \text{VR1} = \text{VR1}<<1; \]
\[ \quad \text{VR1}[0:0] = \text{temp}[0]; \]
\[ \quad \text{VR0}[K-2:0] = 2*\text{VR0}[K-2:0] + \text{temp}[0]; \]
\[ \text{else}\]
\[ \quad \text{VR1} = \text{VR1}>>1; \]
\[ \quad \text{VR1}[31:31] = \text{temp}[0]; \]
\[ \quad \text{VR0}[K-2:0] = 2*\text{VR0}[K-2:0] + \text{temp}[0]; \]
\[ \]
\[ \text{VT0} = \{\text{mem32}\} \]

### Flags

This instruction does not affect any flags in the VSTATUS register.

### Pipeline

This is a 1/1 cycle instruction. The VTRACE and VMOV32 instruction complete in a single cycle.

### Example

```assembly
; etc ...
.align 2
RPTB   _tb_loop_ovlp2, #12
VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE VR1,VR0,VT0,VT1
| VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE VR1,VR0,VT0,VT1
_tb_loop_ovlp2
; etc ...
```

### See also

VTRACE mem32, VR0, VT0, VT1
5.6 Rounding Mode

This section details the rounding operation as applied to a right shift. When the rounding mode is enabled in the VSTATUS register, .5 will be added to the right shifted intermediate value before truncation. If rounding is disabled the right shifted value is only truncated. Table 5-19 shows the bit representation of two values, 11.0 and 13.0. The columns marked Bit-1, Bit-2 and Bit-3 hold temporary bits resulting from the right shift operation.

Table 5-19. Example: Values Before Shift Right

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit -3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>13.000</td>
</tr>
</tbody>
</table>

Table 5-19 shows the intermediate values after the right shift has been applied to Val B. The columns marked Bit-1, Bit-2 and Bit-3 hold temporary bits resulting from the right shift operation.

Table 5-20. Example: Values after Shift Right

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit -3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.625</td>
</tr>
</tbody>
</table>

When the rounding mode is enabled, .5 will be added to the intermediate result before truncation. shows the bit representation of Val A + Val (B >> 3) operation with rounding. Notice .5 is added to the intermediate shifted right value. After the addition, Table 5-21 the bits in Bit-1, Bit-2 and Bit-3 are removed. In this case the result of the operation will be 13 which is the truncated value after rounding.

Table 5-21. Example: Addition with Right Shift and Rounding

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit -3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.625</td>
</tr>
<tr>
<td>.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.500</td>
</tr>
<tr>
<td>Val A + Val B &gt;&gt; 3 + .5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>13.125</td>
</tr>
</tbody>
</table>

When the rounding mode is disabled, the value is simply truncated. Table 5-22 shows the bit representation of the operation Val A + (Val B >> 3) without rounding. After the addition, the bits in Bit-1, Bit-2 and Bit-3 are removed. In this case the result of the operation will be 12 which is the truncated value without rounding.

Table 5-22. Example: Addition with Rounding After Shift Right

<table>
<thead>
<tr>
<th></th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit-1</th>
<th>Bit-2</th>
<th>Bit -3</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11.000</td>
</tr>
<tr>
<td>Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.625</td>
</tr>
<tr>
<td>Val A + Val B &gt;&gt; 3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>12.625</td>
</tr>
</tbody>
</table>
Table 5-23 shows more examples of the intermediate shifted value along with the result if rounding is enabled or disabled. In each case, the truncated value is without .5 added and the rounded value is with .5 added.

**Table 5-23. Shift Right Operation With and Without Rounding**

<table>
<thead>
<tr>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
<th>Bit -1</th>
<th>Bit -2</th>
<th>Value</th>
<th>Result with RND = 0</th>
<th>Result with RND = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.00</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.75</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1.50</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.25</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.75</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.50</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-0.50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-0.75</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1.00</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-1.25</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1.50</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1.75</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-2.00</td>
<td>-2</td>
<td>-2</td>
</tr>
</tbody>
</table>
Chapter 6
SPRUHS1C—October 2014—Revised November 2019

Fast Integer Division Unit (FINTDIV)

The TMS320C2000™ DSP family consists of fixed-point and floating-point digital signal processors. TMS320C2000™ Digital Signal Processors combine control peripheral integration and ease of use of a microcontroller (MCU) with the processing power and C efficiency of TI’s leading DSP technology. This chapter provides an overview of the fast integer division and instructions supported by the C28x fast integer division unit (FINTDIV).

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Overview</td>
<td>749</td>
</tr>
<tr>
<td>6.2 Components of the C28x plus FINTDIV (C28x+FINTDIV)</td>
<td>750</td>
</tr>
<tr>
<td>6.3 CPU Register Set</td>
<td>750</td>
</tr>
<tr>
<td>6.4 Pipeline</td>
<td>750</td>
</tr>
<tr>
<td>6.5 Types of Divisions supported by C28x+FINTDIV</td>
<td>750</td>
</tr>
<tr>
<td>6.6 C28x+Fast Integer Division – Fast Integer Division Instruction Set</td>
<td>752</td>
</tr>
</tbody>
</table>
6.1 Overview

The C28x processor plus fast division unit (C28x+FINTDIV) extends the capabilities of the C28x floating point CPU by adding instructions to support division operations in an optimal manner.

Throughout this document the following notations are used:

• C28x refers to the C28x fixed and floating point CPU.
• C28x plus FINTDIV refer to the C28x CPU with enhancements to support fast integer division operations.

6.1.1 Compatibility With the C28x Fixed-Point CPU and C28x Floating Point CPU

No changes have been made to the C28x base set of instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x CPU are completely compatible with the C28x CPU + FINTDIV and all of the features of the C28x documented in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430 - www.ti.com/lit/spru430) apply to the C28x CPU + FINTDIV.

6.1.2 Fast Integer Division Code development

The TI C28 C/C++ Compiler 18.12.2.LTS supports the generation of FINTDIV instructions in one of three ways:

• Intrinsics, declared in stdlib.h, which take a numerator and denominator and return a structure containing both the remainder and quotient. The intrinsics supported are mentioned in the TMS320C28x Optimizing C/C++ Compiler (www.ti.com/lit/spru514).
• Operators for division/modulus, which will automatically be optimized.
• Standard library functions ldiv and lldiv, both found in stdlib.h.

Only the intrinsics support the alternative Euclidean/Modulo division types. Operators and the standard library functions will perform division according to the C standard.

Compiler option, --idiv_support, controls support for these division sequences. A value of 'none' implies no hardware support for the new instructions, and a value of 'idiv0' implies support for the current specification for the new instructions. The option is only valid when FPU32 or FPU64 is available (--float_support=fpu32 or fpu64) and when using the C2000 EABI (--abi=eabi).

The --opt_for_speed (-mf) option controls whether the sequences themselves are generated inline in the assembly, or are calls to pre-generated sequences in the runtime support library. This is to assist in lowering code size, as these sequences can be anywhere from 8 to 32 instructions long. At the default level (-mf2) or higher, the sequences will be inlined. At lower levels (-mf0 and -mf1), the sequences will be calls to the runtime support library. This affects all three forms of support: intrinsics, operators, and the standard library.


Examples for using the Fast integer division intrinsics are provided in the library section (libraries\math\FASTINTDIV) of C2000WARE.

Examples

The following 3 example functions are equivalent, and will perform a signed 32 by signed 32-bit division and return the quotient:

```
#include <stdlib.h>

long divide_op(long numerator, long denominator)
{
    return numerator / denominator;
}

long divide_intrinsic(long numerator, long denominator)
```
Components of the C28x plus FINTDIV (C28x+FINTDIV)

The C28x+FINTDIV contains

- A central processing unit for generating data and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory.
- A fast Integer division unit (FINTDIV) executing single cycle instructions.

Some features of the C28x+FINTDIV central processing unit are:

- For each of the different division functions, the instructions support operands of different types and size of operands (e.g: 16-bit signed (i16), 16-bit unsigned (ui16), 32-bit signed (i32), 32-bit unsigned (ui32), 64-bit signed (i64) and 64-bit unsigned (ui64)) and different permutations (e.g: ui32/ui32, i32/ui32, i64/i32, ui64/ui32, i64/ui64, i64/i64, etc.) for each of the different division functions.
- Set of instructions which extracts the sign of numerator and denominator based on the operand data type and size, save the flag corresponding to sign of quotient and remainder, and convert the numerator and denominator into unsigned numbers.
- Conditional subtract instruction which can execute multiple conditional subtract operations in a single cycle. This will help perform unsigned division.
- Sign assignment operation to assign sign of quotient and remainder based on the division type (truncated, modulo or euclidean) and flag saved. The unsigned division results obtained from condition subtract are modified based on the type of division.
- Each of the operations used for implementing the division is single cycle and interruptible and hence offer low Interrupt Service Routine (ISR) latency.

6.3 CPU Register Set

The C28x+FINTDIV architecture is the same as the C28x CPU with an extended register and instruction set to support fast division operations. Devices with the C28x+FINTDIV include the standard C28x register set plus an additional set of 6 FINTDIV registers - six source and destination division registers.

6.4 Pipeline

The pipeline flow for FINTDIV instructions is identical to that of the C28x CPU described in TMS320C28x DSP CPU and Instruction Set Reference Guide (www.ti.com/lit/spru430). All the fast division instructions take 1 cycle and do not require delay to allow the operation to complete. This also simplifies the development of software as the need to avoid register conflicts is not necessary while developing software using the FINTDIV.

6.5 Types of Divisions supported by C28x+FINTDIV

In this section, a brief overview of the type of divisions supported by the C28x+FINTDIV is explained. Division is one of the complex operations supported by the different real time processors. In addition to traditional division function, other types of division functions are used in real-time control system applications which are quite unique compared to other embedded processing applications. C28x+FINTDIV processor supports modulo division (floored division) and euclidean division functions in addition to traditional division (truncated division) approach. Different division functions are obtained as given below. The transfer function for the different types of division is shown in Figure 6-1.
The C28x+FINTDIV provide an open and scalable approach to facilitate different types of division while accelerating the division operation and making it completely interruptible. For each of the different division functions, the instructions support operands of different types and size of operands (e.g: 16-bit signed (i16), 16-bit unsigned (ui16), 32-bit signed (i32), 32-bit unsigned (ui32), 64-bit signed (i64) and 64-bit unsigned (ui64)) and different permutations (e.g: ui32/ui32, i32/ui32, i64/i32, ui64/ui32, ui64/ui64, i64/i64, etc.) of the operands for each of the different division functions. The FINTDIV consists of (i) a set of instructions which extract the sign of numerator and denominator based on the operand data type and size, save the flag corresponding to sign of quotient and remainder, and convert the numerator and denominator into unsigned numbers (ii) Conditional subtract instruction which can execute multiple conditional subtract operations in a single cycle. This will help perform unsigned division. (iii) Sign assignment operation to assign sign of quotient and remainder based on the division type (truncated, modulo or euclidean) and flag saved in the first step. The results thus obtained are modified based on the type of division. Each of the operations used for implementing the division is single cycle and interruptible and hence offer low Interrupt Service Routine (ISR) latency.
6.6 C28x+Fast Integer Division – Fast Integer Division Instruction Set

This chapter describes the assembly language instructions of the TMS320C28x plus FINTDIV division unit (C28x+FINTDIV). The instructions listed here are an extension to the standard C28x instruction set. For information on standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430).

6.6.1 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Operands
- Opcode
- Description
- Exceptions
- Pipeline
- Examples
- See also

The example INSTRUCTION is shown to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information. On the C28x+FINTDIV instructions, follow the same format as the C28x. The explanations for the syntax of the operands used in the instruction descriptions for the TMS320C28x plus floating-point processor are given in Table 6-1. For information on the operands of standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RbH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RcH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RdH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>ReH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RfH</td>
<td>R0H to R7H registers</td>
</tr>
</tbody>
</table>
INSTRUCTION dest1, source1, source2  

**Short Description**

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest1</td>
<td>description for the 1st operand for the instruction</td>
</tr>
<tr>
<td>source1</td>
<td>description for the 2nd operand for the instruction</td>
</tr>
<tr>
<td>source2</td>
<td>description for the 3rd operand for the instruction</td>
</tr>
</tbody>
</table>

Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).

**Opcode**

This section shows the opcode for the instruction.

**Description**

Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.

**Restrictions**

Any constraints on the operands or use of the instruction imposed by the processor are discussed.

**Pipeline**

This section describes the instruction in terms of pipeline cycles as described in Section 6.4.

**Example**

Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution. All examples assume the device is running with the OBJMODE set to 1. Normally the boot ROM or the c-code initialization will set this bit.

**See Also**

Lists related instructions.
### 6.6.2 Instructions

The instructions are listed alphabetically, preceded by a summary.

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSI32DIV32 R2H, R1H, R3H</td>
<td>755</td>
</tr>
<tr>
<td>ABSI32DIV32U R2H, R1H</td>
<td>756</td>
</tr>
<tr>
<td>ABSI64DIV32 R2H, R1H:R0H, R3H</td>
<td>757</td>
</tr>
<tr>
<td>ABSI64DIV32U R2H, R1H:R0H</td>
<td>758</td>
</tr>
<tr>
<td>ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H</td>
<td>759</td>
</tr>
<tr>
<td>ABSI64DIV64U R2H:R4H, R1H:R0H</td>
<td>760</td>
</tr>
<tr>
<td>SUBC4UI32 R2H, R1H, R3H</td>
<td>761</td>
</tr>
<tr>
<td>SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H</td>
<td>762</td>
</tr>
<tr>
<td>NEGI32DIV32 R1H, R2H</td>
<td>763</td>
</tr>
<tr>
<td>ENEGI32DIV32 R1H, R2H, R3H</td>
<td>764</td>
</tr>
<tr>
<td>MNEGI32DIV32 R1H, R2H, R3H</td>
<td>765</td>
</tr>
<tr>
<td>NEGI64DIV32 R1H:R0H, R2H</td>
<td>766</td>
</tr>
<tr>
<td>ENEGI64DIV32 R1H:R0H, R2H, R3H</td>
<td>767</td>
</tr>
<tr>
<td>MNEGI64DIV32 R1H:R0H, R2H, R3H</td>
<td>768</td>
</tr>
<tr>
<td>NEGI64DIV64 R1H:R0H, R2H:R4H</td>
<td>769</td>
</tr>
<tr>
<td>ENEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H</td>
<td>770</td>
</tr>
<tr>
<td>MNEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H</td>
<td>771</td>
</tr>
</tbody>
</table>
ABSI32DIV32 R2H, R1H, R3H

Operands

<table>
<thead>
<tr>
<th>R3H</th>
<th>Denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1H</td>
<td>Numerator</td>
</tr>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0110 1000

Description

\[ NI = R1H(31) \]
\[ TF = (R1H(31))^\sim(R3H(31)) \]
if \(((R1H = 0x8000_0000) \mid (R3H = 0x8000_0000)) \} \{ LVF = 1\}
R2H = 0
if \((R1H(31) = 1) \} \{R1H = -R1H\}
if \((R3H(31) = 1) \} \{R3H = -R3H\}


Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
ABSI32DIV32U R2H, R1H

Operands

<table>
<thead>
<tr>
<th>R1H</th>
<th>Numerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0110 1001

Description

NI = R1H(31)
TF = R1H(31)
if (R1H = 0x8000_0000) { LVF = 1}
R2H = 0
if (R1H(31) = 1) (R1H = −R1H)


Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
ABSI64DIV32 R2H, R1H:R0H, R3H

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H</td>
<td>Denominator</td>
</tr>
<tr>
<td>R1H:R0H</td>
<td>Numerator</td>
</tr>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1010 1000

Description

NI = R1H(31)
TF = R1H(31) ^ R3H(31)
if ((R1H:R0H = 0x8000_0000_0000_0000) | (R3H = 0x8000_0000)) { LVF = 1}
R2H = 0
if (R1H(31) = 1) {R1H:R0H = -(R1H:R0H)}
if (R3H(31) = 1) {R3H = -(R3H)}


Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
**ABS164DIV32U R2H, R1H:R0H**

**Operands**

<table>
<thead>
<tr>
<th>R1H:R0H</th>
<th>Numerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

**Opcode**

LSW: 1110 0101 1010 1001

**Description**

\[ NI = R1H(31) \]
\[ TF = R1H(31) \]
\[ \text{if} \ (R1H:R0H = 0x8000_0000_0000_0000) \ { \text{LVF} = 1} \]
\[ R2H = 0 \]
\[ \text{if} \ (R1H(31) = 1) \ (R1H:R0H = -(R1H:R0H)) \]


**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.
ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H:R5H</td>
<td>Denominator</td>
</tr>
<tr>
<td>R1H:R0H</td>
<td>Numerator</td>
</tr>
<tr>
<td>R2H:R4H</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1011 1000

Description

NI = R1H(31)
TF = R1H(31) ^ R3H(31)
if ((R1H:R0H = 0x8000_0000_0000_0000) | (R3H:R5H = 0x8000_0000_0000_0000)) { LVF = 1}
R2H:R4H = 0
if (R1H(31) = 1) {R1H:R0H = -(R1H:R0H)}
if (R3H(31) = 1) {R3H:R5H = -(R3H:R5H)}


Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
**ABSI64DIV64U R2H:R4H, R1H:R0H**

**Operands**

| R1H:R0H | Numerator |
| R2H:R4H | Denominator |

**Opcode**

LSW: 1110 0101 1011 1001

**Description**

NI = R1H(31)

TF = R1H(31)

if (R1H:R0H = 0x8000_0000_0000_0000) { LVF = 1}

R2H:R4H = 0

if (R1H(31) = 1) {R1H:R0H = -(R1H:R0H)}


**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.
SUBC4UI32 R2H, R1H, R3H

Operands

<table>
<thead>
<tr>
<th>R3H</th>
<th>Denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1H</td>
<td>Numerator/Quotient</td>
</tr>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0110 0100

Description

ZI = 0
If (R3H = 0x0) {LVF = 1}
for(i=1;i<=4;i++) {
    temp(32:0) = (R2H << 1) + R1H(31) - R3H
    if(temp(32:0) >= 0)
        R2H = temp(31:0);
        R1H = (R1H << 1) + 1
    else
        R2H:R1H = (R2H:R1H) << 1
}
If (R2H = 0x0) {ZI = 1}


Flags

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H

**Operands**

| R3H:R5H   | Denominator         |
| R1H:R0H   | Numerator/Quotient  |
| R2H:R4H   | Remainder           |

**Opcode**

LSW: 1110 0101 0110 0101

**Description**

ZI = 0

If ((R3H:R5H) = 0x0) {LVF = 1}

for (i=1;i<=2;i++) {
    temp(64:0) = ((R2H:R4H) << 1) + R1H(31) - (R3H:R5H)
    if(temp(64:0) >= 0)
        (R2H:R4H) = temp(63:0);
    else
        (R2H:R4H:R1H:R0H)=(R2H:R4H:R1H:R0H)<<1
}

If (R2H:R4H = 0x0) {ZI = 1}


**Flags**

This instruction modifies the following flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.
NEGI32DIV32 R1H, R2H

Operands

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0110 1010

Description

if(TF = TRUE)
R1H = -R1H
if(NI = TRUE)
(R2H) = -(R2H)


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
ENEGI32DIV32 R1H, R2H, R3H

Operands

<table>
<thead>
<tr>
<th>R3H</th>
<th>Denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 0110 1011

Description

IF (NI = 1 && ZI = 0) {
  R1H = R1H + 1
  R2H = R3H - R2H
}
if(TF = TRUE)
  R1H = -R1H


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
MNEGI32DIV32 R1H, R2H, R3H

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H</td>
<td>Denominator</td>
</tr>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW: 1110 0101 0110 1100</td>
<td></td>
</tr>
</tbody>
</table>

Description

if (TF = 1 & ZI = 0) {
    R1H = R1H + 1
    R2H = R3H - R2H
}
if (TF = TRUE)
    R1H = -R1H
if (NI XOR TF = TRUE)
    (R2H) = -(R2H)


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
NEGI64DIV32 R1H:R0H, R2H

Operands

<table>
<thead>
<tr>
<th>R2H</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1H:R0H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1010 1010

Description

if(TF = TRUE)
(R1H:R0H) = -(R1H:R0H)
if(NI = TRUE)
(R2H) = -(R2H)


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
ENEGI64DIV32 R1H:R0H, R2H, R3H

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H</td>
<td>Denominator</td>
</tr>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H:R0H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1010 1011

Description

if (NI = 1 && ZI = 0) {
  R1H:R0H = R1H:R0H + 1
  R2H = R3H-R2H
}
if(TF = TRUE)
  R1H:R0H = -R1H:R0H


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
MNEGI64DIV32 R1H:R0H , R2H, R3H

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H</td>
<td>Denominator</td>
</tr>
<tr>
<td>R2H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H:R0H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1010 1100

Description

If (TF = 1 & ZI = 0) {
    R1H:R0H = R1H:R0H + 1
    R2H = R3H - R2H
}
If (TF = TRUE)
    R1H:R0H = -R1H:R0H
If (NI XOR TF = TRUE)
    (R2H) = -(R2H)


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
NEGI64DIV64 R1H:R0H, R2H:R4H

Operands

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2H:R4H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H:R0H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW: 1110 0101 1011 1010</td>
<td></td>
</tr>
</tbody>
</table>

Description

if(TF = TRUE) (R1H:R0H) = -(R1H:R0H)
if(NI = TRUE) (R2H:R4H) = -(R2H:R4H)


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
**Fast Integer Division Unit (FINTDIV)**

**ENEGI64DIV64 R1H:R0H , R2H:R4H, R3H:R5H**

**Operands**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H:R5H</td>
<td>Denominator</td>
</tr>
<tr>
<td>R2H:R4H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H:R0H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

**Opcode**

```
LSW: 1110 0101 1011 1011
```

**Description**

```
if (NI = 1 && ZI = 0) {
    R1H:R0H = R1H:R0H + 1
    R2H:R4H = R3H:R5H-R2H:R4H
}
if (TF = TRUE)
    R1H:R0H = -R1H:R0H
```


**Flags**

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

This is a single-cycle instruction.
MNEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H

Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H:R5H</td>
<td>Denominator</td>
</tr>
<tr>
<td>R2H:R4H</td>
<td>Remainder</td>
</tr>
<tr>
<td>R1H:R0H</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

Opcode

LSW: 1110 0101 1011 1100

Description

if (TF = 1 & ZI = 0) {
    R1H:R0H = R1H:R0H + 1
    R2H:R4H = R3H:R5H - R2H:R4H
}
if (TF = TRUE)
    R1H:R0H = -R1H:R0H
if (NI XOR TF = TRUE)
    (R2H:R4H) = -(R2H:R4H)


Flags

This instruction does not modify any flags in the STF register:

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Pipeline

This is a single-cycle instruction.
The Trigonometric Math Unit (TMU) is a fully programmable block that enhances the instruction set of the C28-FPU to more efficiently execute common trigonometric and arithmetic operations.

This document describes the architecture and instruction set of the C28x+FPU+TMU. For a list of all devices with the TMU, see the TMS320x28xx, 28xxx DSP Peripheral Reference Guide (SPRU566).

### Topic | Page
--- | ---
7.1 Overview | 773
7.2 Components of the C28x+FPU Plus TMU | 773
7.3 Data Format | 774
7.4 Pipeline | 775
7.5 TMU Instruction Set | 780
7.1 Overview

The TMU extends the capabilities of a C28x+FPU enabled processor by adding instructions to speed up the execution of common trigonometric and arithmetic operations listed in Table 7-1.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>C Equivalent Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPY2PIF32 RaH,RbH</td>
<td>a = b * 2pi</td>
</tr>
<tr>
<td>DIV2PIF32 RaH,RbH</td>
<td>a = b / 2pi</td>
</tr>
<tr>
<td>DIVF32 RaH,RbH,RcH</td>
<td>a = b/c</td>
</tr>
<tr>
<td>SQRTF32 RaH,RbH</td>
<td>a = sqrt(b)</td>
</tr>
<tr>
<td>SINPUF32 RaH,RbH</td>
<td>a = sin(b*2pi)</td>
</tr>
<tr>
<td>COSPUF32 RaH,RbH</td>
<td>a = cos(b*2pi)</td>
</tr>
<tr>
<td>ATANPUF32 RaH,RbH</td>
<td>a = atan(b)/2pi</td>
</tr>
<tr>
<td>QUADF32 RaH,RbH,RcH,RdH</td>
<td>Operation to assist in calculating ATANPU2</td>
</tr>
</tbody>
</table>

Table 7-2. TMU Type 1 Additional Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>C Equivalent Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEXP2F32 RaH,RbH</td>
<td>RaH = 2^RbH</td>
</tr>
<tr>
<td>LOG2F32 RaH,RbH</td>
<td>RaH = LOG2(RbH)</td>
</tr>
</tbody>
</table>

7.2 Components of the C28x+FPU Plus TMU

The TMU extends the capabilities of the C28x+FPU processors by adding new instructions and, in some cases, leveraging existing FPU instructions to carry out common arithmetic operations used in control applications. No changes have been made to existing instructions, pipeline or memory bus architecture. All TMU instructions use the existing FPU register set (R0H to R7H) to carry out their operations.

7.2.1 Interrupt Context Save and Restore

Since the TMU uses the same register set and flags as the FPU, there are no special considerations with regards to interrupt context save and restore.

If a TMU operation is executing when an interrupt occurs, the C28 can initiate an interrupt context switch without affecting the TMU operation. The TMU will continue to process the operation to completion. Even though most TMU operations are multi-cycle, the TMU operation will have completed by the time register context save operations for the FPU are commenced. When restoring FPU registers, you must make sure that all TMU operations are completed before restoring any register used by another TMU operation.
7.3 Data Format

The treatment of the various IEEE floating-point numerical formats for this TMU is the same as the FPU implementation.

7.3.1 Floating Point Encoding

The encoding of the floating-point formats is given in Table 7-3.

<table>
<thead>
<tr>
<th>S32</th>
<th>E32 (7:0)</th>
<th>M32 (22:0)</th>
<th>Value (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Zero (V = 0)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Negative Zero (V = -0)</td>
</tr>
<tr>
<td>0 +ve</td>
<td>0</td>
<td>non zero</td>
<td>De-normalized (V=(-1)^S * 2^{(E-126)} * (0.M))</td>
</tr>
<tr>
<td>1 -ve</td>
<td>0</td>
<td>non zero</td>
<td>Normal Range (V=(-1)^S * 2^{(E-127)} * (1.M))</td>
</tr>
<tr>
<td>0 +ve</td>
<td>1 to 254</td>
<td>0 to 0x7FFFFF</td>
<td>Positive Max (V = +Max)</td>
</tr>
<tr>
<td>1 -ve</td>
<td>254</td>
<td>0x7FFFFF</td>
<td>Negative Max (V = -Max)</td>
</tr>
<tr>
<td>0</td>
<td>max=255</td>
<td>0</td>
<td>Positive Infinity (V = +Infinity)</td>
</tr>
<tr>
<td>1</td>
<td>max=255</td>
<td>0</td>
<td>Negative Infinity (V = -Infinity)</td>
</tr>
<tr>
<td>x</td>
<td>max=255</td>
<td>non zero</td>
<td>Not A Number (V = NaN)</td>
</tr>
</tbody>
</table>

7.3.2 Negative Zero:

All TMU operations generate a positive (S==0, E==0, M==0) zero, never a negative zero if the result of the operation is zero. All TMU operations treat negative zero operations as zero.

7.3.3 De-Normalized Numbers:

A de-normalized operand (E==0, M!=0) input is treated as zero (E==0, M==0) by all TMU operations. TMU operations never generate a de-normalized value.

7.3.4 Underflow:

Underflow occurs when an operation generates a value that is too small to represent in the given floating-point format. Under such cases, a zero value is returned. If a TMU operation generates an underflow condition, then the latched underflow flag (LUF) is set to 1. The LUF flag will remain latched until cleared by the user executing an instruction that clears the flag.

7.3.5 Overflow:

Overflow occurs when an operation generates a value that is too large to represent in the given floating-point format. Under such cases, a ± Infinity value is returned. If a TMU operation generates an overflow condition, then the latched overflow flag (LVF) is set to 1. The LVF flag will remain latched until cleared by the user executing an instruction that clears the flag.

7.3.6 Rounding:

There are various rounding formats supported by the IEEE standard. Rounding has no meaning for TMU operations (rounding is inherent in the implementation). Hence rounding mode is ignored by TMU operations.

7.3.7 Infinity and Not a Number (NaN):

An NaN operand (E==max, M!=0) input is treated as Infinity (E==max, M==0) for all operations. TMU operations will never generate a NaN value but Infinity instead.
7.4 Pipeline

The TMU enhances the instruction set of the C28-FPU and, therefore, operates the C28x pipeline in the same fashion as the FPU. For a detailed explanation on the working of the pipeline, see the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUEO2).

7.4.1 Pipeline and Register Conflicts

In addition to the restrictions mentioned in the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUEO2), the TMU places the following restrictions on its instructions:

Example 7-1. SINPUF32 Operation (4p cycles)

```
SINPUF32 RaH,RbH ; Value in registers RbH read in this cycle.
Instruction1 ; Instructions 1-3 cannot operate on register RaH.
Instruction2 ; Instructions 1-3 can operate on register RbH.
Instruction3 ; Instructions 1-3 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result in RaH usable by Instruction 4.
```

Example 7-2. COSPUF32 Operation (4p cycles)

```
COSPUF32 RaH,RbH ; Value in registers RbH read in this cycle.
Instruction1 ; Instructions 1-3 cannot operate on register RaH.
Instruction2 ; Instructions 1-3 can operate on register RbH.
Instruction3 ; Instructions 1-3 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result in RaH usable by Instruction 4.
```

Example 7-3. ATANPUF32 Operation (4p cycles)

```
ATANPUF32 RaH,RbH ; Value in registers RbH read in this cycle.
Instruction1 ; Instructions 1-3 cannot operate on register RaH.
Instruction2 ; Instructions 1-3 can operate on register RbH.
Instruction3 ; Instructions 1-3 can be any TMU/FPU/VCU/CPU operation.
; Result, LVF flag updated on Instruction3 (4th cycle).
Instruction4 ; Result in RaH usable by Instruction 4.
```

Example 7-4. DIVF32 Operation (5p cycles)

```
DIVF32 RaH,RbH,RcH ; Value in registers RbH & RcH read in this cycle.
Instruction1 ; Instructions 1-4 cannot operate on register RaH.
Instruction2 ; Instructions 1-4 can operate on register RbH & RcH.
Instruction3 ; Instructions 1-4 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result, LVF and LUF flags updated on Instruction4 (5th cycle).
Instruction5 ; Result in RaH usable by Instruction 5.
```

Example 7-5. SQRTF32 Operation (5p cycles)

```
SQRTF32 RaH,RbH ; Value in register RbH read in this cycle.
Instruction1 ; Instructions 1-4 cannot operate on register RaH.
Instruction2 ; Instructions 1-4 can operate on register RbH.
Instruction3 ; Instructions 1-4 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result, LVF flag updated on Instruction4 (5th cycle).
Instruction5 ; Result in register RaH usable by Instruction 5.
```
Example 7-6. QUADF32 Operations (5p cycles)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUADF32 RaH, RbH, RcH, RdH</td>
<td>Value in registers RcH &amp; RdH read in this cycle.</td>
</tr>
<tr>
<td>Instruction1</td>
<td>Instructions 1-4 cannot operate on registers RaH &amp; RbH.</td>
</tr>
<tr>
<td>Instruction2</td>
<td>Instructions 1-4 can operate on register RbH.</td>
</tr>
<tr>
<td>Instruction3</td>
<td>Instructions 1-4 can be any TMU/FPU/VCU/CPU operation.</td>
</tr>
<tr>
<td>Instruction4</td>
<td>Result, LVF and LUF flags updated on Instruction4 (5th cycle).</td>
</tr>
<tr>
<td>Instruction5</td>
<td>Result in registers RaH &amp; RbH usable by Instruction5.</td>
</tr>
</tbody>
</table>
### 7.4.2 Delay Slot Requirements

The Delay slot requirements for the TMU instructions are presented in Table 7-4.

**Table 7-4. Delay Slot Requirements for TMU Instructions**

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Any Single Cycle FPU operation (including any memory load/store operation)</td>
</tr>
<tr>
<td></td>
<td>SINPUF32/COSPUF32/ATANPUF32/QUADF32/MPY2PIF32/DIV2PIF32/DIVF32/SQRTF32</td>
</tr>
<tr>
<td>2</td>
<td>All FPU 2p-cycle operations MPY/ADD/SUB/….</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>SINPUF32/COSPUF32/ATANPUF32/QUADF32/MPY2PIF32/DIV2PIF32/DIVF32/SQRTF32</td>
</tr>
<tr>
<td>3</td>
<td>SINPUF32/COSPUF32/ATANPUF32</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>All TMU or FPU operations</td>
</tr>
<tr>
<td>4</td>
<td>QUADF32/DIVF32/SQRTF32</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>All TMU or FPU operations</td>
</tr>
</tbody>
</table>

**Special Cases Involving MPY2PIF32/DIV2PIF32**

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>MPY2PIF32/DIV2PIF32</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>SINPUF32/COSPUF32</td>
</tr>
<tr>
<td>6</td>
<td>MPY2PIF32/DIV2PIF32</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>ATANPUF32/QUADF32/DIVF32/SQRTF32</td>
</tr>
<tr>
<td>7</td>
<td>MPY2PIF32/DIV2PIF32</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>All FPU operations</td>
</tr>
<tr>
<td>8</td>
<td>MPY2PIF32/DIV2PIF32</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>MOV32 mem,RxH; Special case: Store result of MPY2PIF32/DIV2PIF32 to memory (but does not include MOV32 operation between CPU and FPU registers).</td>
</tr>
</tbody>
</table>

The “NOPs” can be any other FPU, TMU, VCU or CPU operation that does not conflict with the current active TMU operation (does not use same destination register). For example,

**Example 7-7. Use of Non-Conflicting Instructions in Delay Slots**

```
SINPUF32 R0H,R1H
COSPUF32 R2H,R1H
MOV32 R4H,@VarA
MOV32 R5H,@VarB
ADDF32 R3H,R4H,R0H ; SINPUF32 value (R0H) used here
ADDF32 R7H,R5H,R2H ; COSPUF32 value (R2H) used here
```
The delay FPU slot requirements apply to the operation whereby the destination register value is subsequently used by the TMU operation. For example, in the following case, a parallel MPY and MOV operation precedes the TMU operation and the result from MPY operation is used, then two delay slots are required (Case 2 of Table 7-4):

Example 7-8. Delay Slot Requirement for TMU Instructions That Use Results of Prior FPU Instructions

<table>
<thead>
<tr>
<th>MPY32</th>
<th>R3H, R2H, R1H</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV32</td>
<td>R4H, @varA</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>SINPUF32</td>
<td>R6H, R3H</td>
</tr>
</tbody>
</table>

If however the result of the parallel MOV operation is used, then no delay slots are required since the MOV will complete in a single cycle. (Case 1 of Table 7-4):

Example 7-9. FPU Instruction Followed by a Non-Dependent TMU Instruction

<table>
<thead>
<tr>
<th>MPY32</th>
<th>R3H, R2H, R1H</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV32</td>
<td>R4H, @varA</td>
</tr>
<tr>
<td>SINPUF32</td>
<td>R6H, R4H</td>
</tr>
</tbody>
</table>

7.4.3 Effect of Delay Slot Operations on the Flags

The LVF and LUF flags can only be set. If multiple operations (from FPU or TMU) try to set the flags, the operations on the flags are ORed together. Operations that set the LVF or LUF flags (either FPU or TMU) are allowed in delay slots. For example, the following sequence of operations is valid:

Example 7-10. Valid Back-to-Back Instructions That may Set the LVF, LUF Flag

| MPY2PIF32 | R0H, R0H   |
| MPY2PIF32 | R1H, R1H   |

If the SETFLG, SAVE, RESTORE, MOVST0, or loading and storing of the STF register, operations try to modify the state of the LVF, LUF flags while a TMU or any other FPU operation is trying to set the flags, the LUV, LVF flags are undefined. This can only occur if the SAVE, SETFLG, RESTORE, MOVST0, or loading and storing of the STF register, operations are placed in the delay slots of the pipeline operations; this should be avoided. This also applies to ZF and NF flags, which are not affected by TMU operations.

7.4.4 Multi-Cycle Operations in Delay Slots

A multi-cycle operation like RET, BRANCH, CALL is equivalent to a minimum four NOPs. For example, the code shown in Example 7-11 returns the correct value because LRETR takes a minimum of four cycles to execute (equivalent to four NOPs):

Example 7-11. Multi-Cycle Operation in the Delay Slot of a TMU Instruction

| DIVF32   | R0H, R2H, R1H |
|          | LRETR         |
7.4.5 Moves From FPU Registers to C28x Registers

When transferring from floating-point unit registers (result of an FPU or TMU operation) to the C28x CPU register, additional pipeline alignment is required as shown in Example 7-12.

Example 7-12. Floating-Point to C28x Register Software Pipeline Alignment

```Assembly
; SINPUF32: Per unit sine: 4 pipeline cycle operation
; An alignment cycle is required before copying R0H to ACC
SINPUF32 R0H,R1H
NOP ; Delay Slot 1
NOP ; Delay Slot 2
NOP ; Delay Slot 3
NOP ; Alignment cycle
MOV32 @ACC,R0H
```
7.5 **TMU Instruction Set**

This section describes the assembly language instructions of the TMU.

7.5.1 **Instruction Descriptions**

The explanations for the syntax of the operands are given in Table 7-5. For information on the operands of standard C28x instructions, see the *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#16FHi</td>
<td>16-bit immediate (hex or float) value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FHiHex</td>
<td>16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.</td>
</tr>
<tr>
<td>#16FLoHex</td>
<td>A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32Fhex</td>
<td>32-bit immediate value that represents an IEEE 32-bit floating-point value</td>
</tr>
<tr>
<td>#32F</td>
<td>Immediate float value represented in floating-point representation</td>
</tr>
<tr>
<td>#0.0</td>
<td>Immediate zero</td>
</tr>
<tr>
<td>#RC</td>
<td>16-bit immediate value for the repeat count</td>
</tr>
<tr>
<td><em>(0:16bitAddr)</em></td>
<td>16-bit immediate address, zero extended</td>
</tr>
<tr>
<td>CNDF</td>
<td>Condition to test the flags in the STF register</td>
</tr>
<tr>
<td>FLAG</td>
<td>Selected flags from STF register (OR) 11 bit mask indicating which floating-point status flags to change</td>
</tr>
<tr>
<td>label</td>
<td>Label representing the end of the repeat block</td>
</tr>
<tr>
<td>mem16</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location</td>
</tr>
<tr>
<td>mem32</td>
<td>Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location</td>
</tr>
<tr>
<td>RaH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RbH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RcH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RdH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>ReH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RH</td>
<td>R0H to R7H registers</td>
</tr>
<tr>
<td>RB</td>
<td>Repeat Block Register</td>
</tr>
<tr>
<td>STF</td>
<td>FPU Status Register</td>
</tr>
<tr>
<td>VALUE</td>
<td>Flag value of 0 or 1 for selected flag (OR) 11 bit mask indicating the flag value; 0 or 1</td>
</tr>
</tbody>
</table>
### INSTRUCTION dest1, source1, source2 Short Description

**Operands**

<table>
<thead>
<tr>
<th>dest1</th>
<th>Description for the 1st operand for the instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>source1</td>
<td>Description for the 2nd operand for the instruction</td>
</tr>
<tr>
<td>source2</td>
<td>Description for the 3rd operand for the instruction</td>
</tr>
</tbody>
</table>

Each instruction has a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).

**Opcode**

This section shows the opcode for the instruction.

**Description**

Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.

**Restrictions**

Any constraints on the operands or use of the instruction are discussed.

**Pipeline**

This section describes the instruction in terms of pipeline cycles.

**Example**

If applicable, register and memory values are given before and after instruction execution. All examples assume the device is running with the OBJMODE set to 1. Normally the boot ROM or the c-code initialization will set this bit.

**See Also**

Lists related instructions.
7.5.2 Common Restrictions

For all the TMU instructions, the inputs are conditioned as follows (LVF, LUF are not affected):

- Negative zero is treated as positive zero
- Positive or negative denormalized numbers are treated as positive zero
- Positive and negative NaN are treated as positive and negative infinity respectively

7.5.3 TMU Type 0 Instructions

The TMU Type 0 instructions are listed below.

Table 7-6. Summary of Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPY2PIF32 RaH, RbH — 32-Bit Floating-Point Multiply by Two Pi</td>
<td>783</td>
</tr>
<tr>
<td>DIV2PIF32 RaH, RbH — 32-Bit Floating-Point Divide by Two Pi</td>
<td>784</td>
</tr>
<tr>
<td>DIVF32 RaH, RbH, RcH — 32-Bit Floating-Point Division</td>
<td>785</td>
</tr>
<tr>
<td>SQRTF32 RaH, RbH — 32-Bit Floating-Point Square Root</td>
<td>787</td>
</tr>
<tr>
<td>SINPUF32 RaH, RbH — 32-Bit Floating-Point Sine (per unit)</td>
<td>788</td>
</tr>
<tr>
<td>COSPUF32 RaH, RbH — 32-Bit Floating-Point Cosine (per unit)</td>
<td>790</td>
</tr>
<tr>
<td>ATANPUF32 RaH, RbH — 32-Bit Floating-Point ArcTangent (per unit)</td>
<td>792</td>
</tr>
<tr>
<td>QUADF32 RaH, RbH, RcH — Quadrant Determination Used in Conjunction With ATANPUF32()</td>
<td>793</td>
</tr>
</tbody>
</table>
**MPY2PIF32 RaH, RbH**  *32-Bit Floating-Point Multiply by Two Pi*

**Operands**

- **RaH**: Floating-point destination register (R0H to R7H)
- **RbH**: Floating-point source register (R0H to R7H)

**Opcode**

- **LSW**: 1110 0010 0111 0000
- **MSW**: 0000 0000 00bb baaa

**Description**

This operation is similar to the MPYF32 operation except that the second operand is the constant value 2π:

\[ RaH = RbH \times 2\pi \]

This operation is used in converting Per Unit values to Radians. Per Unit values are used in control applications to represent normalized radians:

<table>
<thead>
<tr>
<th>Per Unit</th>
<th>Radians</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2π</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1.0</td>
<td>2π</td>
</tr>
</tbody>
</table>

2π = 6.28318530718 \(\approx 1.570796326795 \times 2^{2}\)

In IEEE 32-bit Floating point format:

\[ S = 0 << 31 = 0x00000000 \]
\[ E = (2 + 127) << 23 = 129 << 23 = 0x40800000 \]
\[ M = (1.570796326795 \times 2^{23}) \& 0x007FFFFF = 0x00490FDB \]

\[ 2π = S+E+M = 0x40C90FDB \]

**Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Restrictions**

If( RaH result is too big for floating-point number, Ea > 255 ){  
  **RaH** = ±Infinity  
  **LVF** = 1;  
}

**Pipeline**

Instruction takes 2 pipeline cycles to execute if followed by either SINPUF32, COSPUF32 or MOV32 mem, Rx operations and 3 pipeline cycles for all other operations (FPU or TMU).

**Example**

```plaintext
; Convert Per Unit value to Radians:
MOV32 ROH, @PerUnit ; ROH = Per Unit value
MPY2PIF32 ROH, ROH ; ROH = ROH * 2π
NOP ; pipeline delay
MOV32 @Radians, ROH ; store Radian result
; 4 cycles
```
DIV2PIF32 RaH, RbH — 32-Bit Floating-Point Divide by Two Pi

DIV2PIF32 RaH, RbH  
32-Bit Floating-Point Divide by Two Pi

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0010 0111 0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

Description

This operation is similar to the MPYF32 operation except that the second operand is the constant value 1/2pi:

\[ RaH = RbH \times \frac{1}{2\pi} \]

This operation is used in converting Radians to Per unit values. Per unit values are used in control representing normalized Radians:

<table>
<thead>
<tr>
<th>Per Unit</th>
<th>Radians</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2pi</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-2pi</td>
</tr>
</tbody>
</table>

In IEEE 32-bit Floating point format:

\[ \frac{1}{2\pi} = 0.1591549430919 = 1.273239544735 \times 2^{-3} \]

\[ S = 0 << 31 = 0x00000000 \]

\[ E = (-3+127) << 23 = 124 << 23 = 0x3E000000 \]

\[ M = (1.273239544735 \times 2^{23}) & 0x007FFFFF = 0x0022F983 \]

\[ \frac{1}{2\pi} = S+E+M = 0x3E22F983 \]

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Restrictions

If( RaH result is too small for floating-point number, Ea < 0) {

\[ RaH = 0.0 \]

\[ LUF = 1; \]

}

Pipeline

Instruction takes 2 pipeline cycles to execute if followed by either SINPUF32, COSPUF32 or MOV32 mem, Rx operations and 3 pipeline cycles for all other operations (FPU or TMU).

Example

;;; Convert Per Unit value to Radians:
MOV32 R0H, @Radians ; R0H = Radian value
DIV2PIF32 R0H, R0H ; R0H = R0H * 1/2pi
NOP ; pipeline delay
MOV32 @Per Unit ; store Per Unit result
; 4 cycles
DIVF32 RaH, RbH, RcH  

32-Bit Floating-Point Division

Operands

RaH  Floating-point destination register (R0H to R7H)
RbH  Floating-point source register (R0H to R7H)
RcH  Floating-point source register (R0H to R7H)

Opcode

LSW  1110 0010 0111 0100
MSW  0000 000c cccb baaa

Description

RaH = RbH/RcH

The sequence of operations are as follows:

\[
\begin{align*}
Sa &= Sb \ XOR \ Sc; \\
Ea &= (Eb - Ec) + 127; \\
Ma &= Mb / Mc; & \quad 0.5 < Ma < 2.0 \\
\text{if}(Ma < 1.0)\{ \\
\quad Ea &= Ea - 1; \\
\quad Ma &= Ma * 2.0; \\
\}\text{Re-normalize mantissa range} \\
\text{if}(Ea >= 255)\{ \\
\quad Ea &= 255; & \quad \text{Return Inf} \\
\quad Ma &= 0; \\
\quad LVF &= 1; & \quad \text{Set overflow flag} \\
\}\text{Check if result too big:} \\
\text{if}((Ea == 0) \& (Ma != 0))\{ \\
\quad Sa &= 0; \\
\quad Ea &= 0; & \quad \text{Return zero} \\
\quad Ma &= 0; \\
\quad LUF &= 1; & \quad \text{Set underflow flag} \\
\}\text{Check if result Denorm value:} \\
\text{if}(Ea < 0)\{ \\
\quad Sa &= 0; \\
\quad Ea &= 0; & \quad \text{Return zero} \\
\quad Ma &= 0; \\
\quad LUF &= 1; & \quad \text{Set underflow flag} \\
\}
\]

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
DIVF32 RaH, RbH, RcH — 32-Bit Floating-Point Division

Restrictions

The following boundary conditions apply:

<table>
<thead>
<tr>
<th>Division</th>
<th>Result</th>
<th>LVF</th>
<th>LUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/0</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>0/Inf</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Inf/Normal</td>
<td>Inf</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Inf/0</td>
<td>Inf</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Inf/Inf</td>
<td>Inf</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Normal/0</td>
<td>Inf</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Normal/Inf</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Pipeline

Instruction takes 5 pipeline cycles to execute.

Example

;;;; Calculate Z = Y/X
MOV32 R0H,@X ; R0H = X
MOV32 R1H,@Y ; R1H = Y
DIVF32 R2H,R1H,R0H ; R2H = R1H/R0H = Y/X = Z
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
MOV32 @Z,R2H ; Z = Y/X
; 8 cycles
**SQRTF32 RaH, RbH**  
**32-Bit Floating-Point Square Root**

**Operands**

- **RaH**: Floating-point destination register (R0H to R7H)
- **RbH**: Floating-point source register (R0H to R7H)

**Opcode**

- **LSW**: 1110 0010 0111 0111
- **MSW**: 0000 0000 00bb baaa

**Description**

\[ RaH = \sqrt{RbH} \]

**Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Restrictions**

\[
\text{If}( RbH < 0.0 \text{ or } -\text{Inf} ) \{ \text{// Check if input is negative:} \\
\quad Sa = 0; \quad \text{// Return zero} \\
\quad Ea = 0; \\
\quad Ma = 0; \\
\quad LVF = 1; \quad \text{// Set overflow flag} \\
\}
\]

\[
\text{If}( RbH == +\text{Inf} ) \{ \\
\quad Sa = 0; \quad \text{// Return Inf} \\
\quad Ea = 255; \\
\quad Ma = 0; \\
\quad LVF = 1; \quad \text{// Set overflow flag} \\
\}
\]

**Pipeline**

Instruction takes 5 pipeline cycles to execute.

**Example**

```c
;; Calculate Y = sqrt(X)
MOV32 R0H, @X ; R0H = X
SQRTF32 R1H, R0H ; R1H = sqrt(X)
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
MOV32 @Y, R1H ; Y = sqrt(X)
7 cycles
```

SINPUF32 RaH, RbH — 32-Bit Floating-Point Sine (per unit)

Operands

- **RaH**: Floating-point destination register (R0H to R7H)
- **RbH**: Floating-point source register (R0H to R7H)

Opcode

- **LSW**: 1110 0010 0111 1000
- **MSW**: 0000 0000 00bb baaa

Description

This instruction performs the following equivalent operation:

\[ \text{PerUnit} = \text{fraction}(RbH) \]
\[ RaH = \sin(\text{PerUnit} \times 2\pi) \]

In control applications radians are usually normalized to the range of -1.0 to 1.0.

<table>
<thead>
<tr>
<th>Per Unit</th>
<th>Radians</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2\pi</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-2\pi</td>
</tr>
</tbody>
</table>

The operation takes the fraction of the input value RbH. This equates to the cosine waveform repeating itself every 2\pi radians.

<table>
<thead>
<tr>
<th>RbH</th>
<th>Per Unit</th>
<th>Radians</th>
<th>Sine Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>0.0</td>
<td>0\pi/2</td>
<td>0.0</td>
</tr>
<tr>
<td>1.75</td>
<td>0.75</td>
<td>3\pi/2</td>
<td>-1.0</td>
</tr>
<tr>
<td>1.5</td>
<td>0.5</td>
<td>\pi</td>
<td>0.0</td>
</tr>
<tr>
<td>1.25</td>
<td>0.25</td>
<td>\pi/2</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>0\pi/2</td>
<td>0.0</td>
</tr>
<tr>
<td>0.75</td>
<td>0.75</td>
<td>3\pi/2</td>
<td>-1.0</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>\pi</td>
<td>0.0</td>
</tr>
<tr>
<td>0.25</td>
<td>0.25</td>
<td>\pi/2</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.25</td>
<td>-0.25</td>
<td>-\pi/2</td>
<td>-1.0</td>
</tr>
<tr>
<td>-0.5</td>
<td>-0.5</td>
<td>-\pi</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.75</td>
<td>-0.75</td>
<td>-3\pi/2</td>
<td>1.0</td>
</tr>
<tr>
<td>-1.0</td>
<td>0.0</td>
<td>0\pi/2</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.25</td>
<td>-0.25</td>
<td>-\pi/2</td>
<td>-1.0</td>
</tr>
<tr>
<td>-1.5</td>
<td>-0.5</td>
<td>-\pi</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.75</td>
<td>-0.75</td>
<td>-3\pi/2</td>
<td>1.0</td>
</tr>
<tr>
<td>-2.0</td>
<td>0.0</td>
<td>0\pi/2</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
SINPUF32 RaH, RbH — 32-Bit Floating-Point Sine (per unit)

Restrictions
If the input value is too small (\( \leq 2^{-33} \)) or too big (\( \geq 2^{22} \)), then the output will be returned as 0.0 (no flags affected).

Pipeline
Instruction takes 4 pipeline cycles to execute.

Example

```assembly
;; Convert Radian value to PerUnit value and calculate Sin value:
MOV32 ROH,@RadianValue ; ROH = Radian value
DIV2PIF32 R1H,ROH ; R1H=ROH/2pi= Per Unit Value
NOP ; pipeline delay
SINPUF32 R2H,R1H ; R2H = SINPU(fraction(R1H))
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
MOV32 @SinValue,R2H ; Sin Value=sin(Radian Value)
```

; 8 cycles
COSPUF32 RaH, RbH — 32-Bit Floating-Point Cosine (per unit)

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0010 0111 1001</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

Description

This instruction performs the following equivalent operation:

\[
\text{PerUnit} = \text{fraction}(RbH) \\
\text{RaH} = \cos(\text{PerUnit} \times 2\pi)
\]

In control applications radians are usually normalized to the range of -1.0 to 1.0.

<table>
<thead>
<tr>
<th>Per Unit</th>
<th>Radians</th>
<th>Cosine Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2\pi</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-2\pi</td>
<td>-1.0</td>
</tr>
</tbody>
</table>

The operation takes the fraction of the input value RbH. This equates to the cosine waveform repeating itself every 2\pi radians.

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Restrictions
If the input value is too small (\(\leq 2^{-33}\)) or too big (\(\geq 2^{22}\)), then the output will be returned as 1.0 (no flags affected).

Pipeline
Instruction takes 4 pipeline cycles to execute.

Example

```assembly
;; Convert Radian value to PerUnit value and
;; calculate Sin value:
MOV32 R0H,@RadianValue ; R0H = Radian value
DIV2PIF32 R1H,R0H ; R1H=R0H/2pi= Per Unit Value
NOP ; pipeline delay
COSPUF32 R2H,R1H ; R2H = COSPU(fraction(R1H))
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
MOV32 @CosValue,R2H ; Cos Value=cos(Radian Value)
```

; 8 cycles
ATANPUF32 RaH, RbH — 32-Bit Floating-Point ArcTangent (per unit)

ATANPUF32 RaH, RbH 32-Bit Floating-Point ArcTangent (per unit)

Operands

RaH  Floating-point destination register (R0H to R7H)
RbH  Floating-point source register (R0H to R7H)

Opcode

LSW  1110 0010 0111 1010
MSW  0000 0000 00bb baaa

Description

This instruction computes the arc tangent of a given value and returns the result as a per-unit value:

\[ \text{PerUnit} = \frac{\text{atan}(RbH)}{2\pi} \]

The operation limits the input range of the input value RbH to:

\[-1.0 \leq RbH \leq 1.0\]

Values outside this range return 0.125 as follows:

<table>
<thead>
<tr>
<th>RbH</th>
<th>Per Unit</th>
<th>Radians</th>
<th>ATANPU Value</th>
<th>LVF Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;1.0</td>
<td>0.125</td>
<td>(\pi/4)</td>
<td>0.125</td>
<td>1</td>
</tr>
<tr>
<td>1.0</td>
<td>0.125</td>
<td>(\pi/4)</td>
<td>0.125</td>
<td>1</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-0.125</td>
<td>-(\pi/4)</td>
<td>-0.125</td>
<td>1</td>
</tr>
<tr>
<td>&lt;-1.0</td>
<td>-0.125</td>
<td>-(\pi/4)</td>
<td>-0.125</td>
<td>1</td>
</tr>
</tbody>
</table>

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Pipeline

Instruction takes 4 pipeline cycles to execute.

Example

```plaintext
;; Calculate ATAN and generate Per Unit value and
;; convert to Radians:
MOV32    R0H, @AtanValue  ; R0H = Atan Value
ATANPUF32 R1H, R0H       ; R1H = ATANPU(R0H)
NOP      ; pipeline delay
NOP      ; pipeline delay
NOP      ; pipeline delay
MPY2PIF32 R2H, R1H       ; R2H = R1H * 2\pi
                      ; = Radian value
NOP      ; pipeline delay
MOV      @RadianValue, R2H ; Store result
                      ; 8 cycles
```
QUADF32 RaH, RbH, RcH  Quadrant Determination Used in Conjunction With ATANPUF32()

Operands

<table>
<thead>
<tr>
<th>RaH</th>
<th>Floating-point destination register (R0H to R7H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RbH</td>
<td>Floating-point destination register (R0H to R7H)</td>
</tr>
<tr>
<td>RcH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
<tr>
<td>RdH</td>
<td>Floating-point source register (R0H to R7H)</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0010 0111 1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 dddc ccbb baaa</td>
</tr>
</tbody>
</table>

Description

This operation, in conjunction with atanpu(), is used in calculating atanpu2() for a full circle:

- RdH = X value
- RbH = Y value
- RbH = Ratio of X & Y
- RaH = Quadrant value (0.0, ±0.25, ±0.5)

Figure 7-1 shows how the values RaH and RbH are generated based on the contents of RbH and RcH.

Figure 7-1. Calculation of RaH (Quadrant) and RbH (Ratio) Based on RcH (Y) and RdH (X) Values
QUADF32 RaH, RbH, RcH — Quadrant Determination Used in Conjunction With ATANPUF32()

The algorithm for this instruction is as follows:

```c
if( (fabs(RcH(Y)) == 0.0) & (fabs(RdH(X)) == 0.0) ) {
    RaH(Quadrant) = 0.0;
    RbH(Ratio)   = 0.0;
} else if( fabs(RcH(Y)) <= fabs(RdH(X)) ) {
    RbH(Ratio)   = RcH(Y) / RdH(X);
    if( RdH(X) >= 0.0 )
        RaH(Quadrant) = 0.0;
    else {
        if( RcH(Y) >= 0.0 )
            RaH(Quadrant) = 0.5;
        else
            RaH(Quadrant) = -0.5;
    }
} else {
    if( RcH(Y) >= 0.0 )
        RaH(Quadrant) = 0.25;
    else
        RaH(Quadrant) = -0.25;
    RbH(Ratio)   = -RdH(X) / RcH(Y);
}
```

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Restrictions

<table>
<thead>
<tr>
<th>Division</th>
<th>Result</th>
<th>LVF</th>
<th>LUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/0</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>0/Inf</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Inf/Normal</td>
<td>Inf</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Inf/0</td>
<td>Inf</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Inf/Inf</td>
<td>Inf</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Normal/0</td>
<td>Inf</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Normal/Inf</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Pipeline

Instruction takes 5 pipeline cycles to execute.
Example

;; Calculate Z = atan2(Y,X), where Z is in radians:

MOV32 R0H,0X ; R0H = X
MOV32 R1H,0Y ; R1H = Y

;; if(Y <= X) R2H= R1H/R0H
;; else R2H= -R0H/R1H

;; R3H= 0.0, +/-0.25, +/-0.5
QUADF32 R3H,R2H,R1H,R0H

NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay

;; R4H = ATANPU(R2H)(Per Unit result)
ATANPUF32 R4H,R2H

NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay

;; R5H = R3H + ATANPU(R4H) = ATANPU2 value
ADDF32 R5H,R3H,R4H

NOP ; pipeline delay

;; R6H = ATANPU2 * 2pi = atan2 value(radians)
MPY2PIF32 R6H,R5H

NOP ; pipeline delay

MOV32 0Z,R6H ; store result

; 16 cycles
7.5.4 TMU Type 1 Instructions

TMU Type 1 has all of the Type 0 instructions and adds the IEXP2F32 and LOG2F32 instructions.

Table 7-7. Summary of Instructions

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEXP2F32 RaH, RbH — 32-Bit Floating-Point Inverse Exponent</td>
<td>797</td>
</tr>
<tr>
<td>LOG2F32 RaH, RbH — 32-Bit Floating-Point Base-2 Logarithm</td>
<td>798</td>
</tr>
</tbody>
</table>
**IEXP2F32 RaH, RbH — 32-Bit Floating-Point Inverse Exponent**

**Operands**

- **RaH**: Floating-point destination register (R0H to R7H)
- **RbH**: Floating-point source register (R0H to R7H)

**Opcode**

<table>
<thead>
<tr>
<th>LSW</th>
<th>1110 0010 0111 0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

**Description**

This instruction computes \(2.0f\) raised to the inverse power of a floating point number. The equivalent operation is:

\[
RaH = 2^{-|RbH|}
\]

<table>
<thead>
<tr>
<th><strong>RbH</strong></th>
<th><strong>EXP Value (RaH)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>-Inf / Nan</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.0</td>
<td>0.25</td>
</tr>
<tr>
<td>-1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Denorm</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>2.0</td>
<td>0.25</td>
</tr>
<tr>
<td>Inf / Nan</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Pipeline**

Instruction takes 4 pipeline cycles to execute.

**Example**

```plaintext
;; Calculate inverse exponent
IEXP2F32 R2H,R1H           ; R2H = 2^-|R1H|
NOP                         ; pipeline delay
NOP                         ; pipeline delay
NOP                         ; pipeline delay
MOV32 @ExpValue,R2H         ; ExpValue = 2^-|R1H|
                        ; 5 Cycles
```
**LOG2F32 RaH, RbH — 32-Bit Floating-Point Base-2 Logarithm**

**Operands**
- **RaH**: Floating-point destination register (R0H to R7H)
- **RbH**: Floating-point source register (R0H to R7H)

**Opcode**
- **LSW**: 1110 0010 0111 0010
- **MSW**: 0000 0000 00bb baaa

**Description**
This instruction computes the base-2 logarithm of a floating point number. The equivalent operation is:

\[ RaH = \log_2(RbH) \]

Domain \( (RbH) = [-\infty, \infty] \)
Range \( (RaH) = [0, 128) \cup \{\infty\} \)

<table>
<thead>
<tr>
<th>RbH</th>
<th>LOG2 Value (RaH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Inf / Nan 0.0</td>
<td>-Inf</td>
</tr>
<tr>
<td>-2.0</td>
<td>-Inf</td>
</tr>
<tr>
<td>-1.0</td>
<td>-Inf</td>
</tr>
<tr>
<td>0.0</td>
<td>-Inf</td>
</tr>
<tr>
<td>Denorm</td>
<td>-Inf</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Inf / Nan</td>
<td>Inf</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>TF</th>
<th>ZI</th>
<th>NI</th>
<th>ZF</th>
<th>NF</th>
<th>LUF</th>
<th>LVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Pipeline**
Instruction takes 4 pipeline cycles to execute.

**Example**
```plaintext
;; Calculate base-2 logarithm
LOG2F32 R2H,R1H ; R2H = \log_2(R1H)
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
MOV32 @LogValue,R2H ; LogValue = \log_2(RbH)
;; 5 Cycles
```
## Revision History

### Changes from May 23, 2018 to November 15, 2018

<table>
<thead>
<tr>
<th>Related Documentation: Extensive changes have been made to this document since the last publication.</th>
<th>9</th>
</tr>
</thead>
</table>
IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated