# Contents

Preface ........................................................................................................................................ 4

1 Introduction ........................................................................................................................ 8
   1.1 Introduction to the Central Processing Unit (CPU) ......................................................... 9
   1.2 Compatibility with the C28x Fixed-Point CPU ............................................................. 9
      1.2.1 Floating-Point Code Development ..................................................................... 10
   1.3 Components of the C28x plus Floating-Point CPU ...................................................... 10
      1.3.1 Emulation Logic ................................................................................................. 11
      1.3.2 Memory Map .................................................................................................... 11
      1.3.3 On-Chip Program and Data ............................................................................... 11
      1.3.4 CPU Interrupt Vectors ..................................................................................... 11
   1.4 Memory Interface ........................................................................................................ 11
      1.4.1 Address and Data Buses ................................................................................... 11
      1.4.2 Alignment of 32-Bit Accesses to Even Addresses ............................................... 12

2 CPU Register Set ................................................................................................................. 14
   2.1 CPU Registers .............................................................................................................. 15
      2.1.1 Floating-Point Status Register (STF) ................................................................. 17
      2.1.2 Repeat Block Register (RB) ............................................................................... 19

3 Pipeline ................................................................................................................................. 22
   3.1 Pipeline Overview ........................................................................................................ 23
   3.2 General Guidelines for Floating-Point Pipeline Alignment ........................................... 23
   3.3 Moves from FPU Registers to C28x Registers ............................................................. 24
   3.4 Moves from C28x Registers to FPU Registers ............................................................. 24
   3.5 Parallel Instructions ................................................................................................... 25
   3.6 Invalid Delay Instructions ......................................................................................... 25
   3.7 Optimizing the Pipeline .............................................................................................. 28

4 Instruction Set ....................................................................................................................... 31
   4.1 Instruction Descriptions ............................................................................................. 32
   4.2 Instructions ................................................................................................................ 34

A Revision History ................................................................................................................ 149
   A.1 Changes ..................................................................................................................... 149
List of Figures

1-1. FPU Functional Block Diagram ................................................................. 9
2-1. C28x With Floating-Point Registers............................................................. 15
2-2. Floating-point Unit Status Register (STF) .................................................... 17
2-3. Repeat Block Register (RB) ....................................................................... 19
3-1. FPU Pipeline ............................................................................................... 23

List of Tables

2-1. 28x Plus Floating-Point CPU Register Summary ........................................ 16
2-2. Floating-point Unit Status (STF) Register Field Descriptions ..................... 17
2-3. Repeat Block (RB) Register Field Descriptions .......................................... 19
4-1. Operand Nomenclature .............................................................................. 32
4-2. Summary of Instructions ............................................................................ 34
A-1. Technical Changes Made in This Revision ................................................ 149
Preface


Read This First

This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C28x floating-point DSP.

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 in 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— TMS320F28335, TMS320F28334, TMS320F28332, TMS320F28235, TMS320F28234, TMS320F28232 Digital Signal Controllers (DSCs) Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications for the F2833x/2823x devices.
SPRZ272— TMS320F28335, F28334, F28332, TMS320F28235, F28234, F28232 Digital Signal Controllers (DSCs) Silicon Errata describes the advisories and usage notes for different versions of silicon.

CPU User’s Guides—
SPRU430 — TMS320C28x CPU and Instruction Set Reference Guide describes the central processing unit (CPU) and the assembly language instructions of the TMS320C28x fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.
SPRUEO2 — TMS320C28x Floating Point Unit and Instruction Set Reference Guide describes the floating-point unit and includes the instructions for the FPU.

Peripheral Guides—
SPRU566 — TMS320x28xx, 28xxx DSP Peripheral Reference Guide describes the peripheral reference guides of the 28x digital signal processors (DSPs).
SPRUF80 — TMS320x2833x, 2823x System Control and Interrupts Reference Guide describes the various interrupts and system control features of the 2833x and 2823x digital signal controllers (DSCs).
SPRUE812 — TMS320x2833x, 2823x Analog-to-Digital Converter (ADC) Reference Guide describes how to configure and use the on-chip ADC module, which is a 12-bit pipelined ADC.
SPRU949 — TMS320x2833x, 2823x DSC External Interface (XINTF) Reference Guide describes the XINTF, which is a non-multiplexed asynchronous bus, as it is used on the 2833x and 2823x devices.

SPRU963 — TMS320x2833x, 2823x Boot ROM Reference Guide describes the purpose and features of the bootloader (factory-programmed boot-loading software) and provides examples of code. It also describes other contents of the device on-chip boot ROM and identifies where all of the information is located within that memory.

SPRUF87 — TMS320x2833x, 2823x Multichannel Buffered Serial Port (McBSP) Reference Guide describes the McBSP available on the 2833x and 2823x devices. The McBSPs allow direct interface between a DSP and other devices in a system.

SPRUF88 — TMS320x2833x, 2823x Direct Memory Access (DMA) Module Reference Guide describes the DMA on the 2833x and 2823x devices.

SPRUG04 — TMS320x2833x, 2823x Enhanced Pulse Width Modulator (ePWM) Module Reference Guide describes the main areas of the enhanced pulse width modulator that include digital motor control, switch mode power supply control, UPS (uninterruptible power supplies), and other forms of power conversion.

SPRUG02 — TMS320x2833x, 2823x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide describes the operation of the high-resolution extension to the pulse width modulator (HRPWM).

SPRUG44 — TMS320x2833x, 2823x Enhanced Capture (eCAP) Module Reference Guide describes the enhanced capture module. It includes the module description and registers.

SPRUG05 — TMS320x2833x, 2823x Enhanced Quadrature Encoder Pulse (eQEP) Module Reference Guide describes the eQEP module, which is used for interfacing with a linear or rotary incremental encoder to get position, direction, and speed information from a rotating machine in high-performance motion and position control systems. It includes the module description and registers.

SPRUEU1 — TMS320x2833x, 2823x Enhanced Controller Area Network (eCAN) Reference Guide describes the eCAN that uses established protocol to communicate serially with other controllers in electrically noisy environments.

SPRUFZ5 — TMS320x2833x, 2823x Serial Communications Interface (SCI) Reference Guide describes the SCI, which is a two-wire asynchronous serial port, commonly known as a UART. The SCI modules support digital communications between the CPU and other asynchronous peripherals that use the standard non-return-to-zero (NRZ) format.

SPRUEU3 — TMS320x2833x, 2823x DSC Serial Peripheral Interface (SPI) Reference Guide describes the SPI - a high-speed synchronous serial input/output (I/O) port - that allows a serial bit stream of programmed length (one to sixteen bits) to be shifted into and out of the device at a programmed bit-transfer rate.

SPRUG03 — TMS320x2833x, 2823x Inter-Integrated Circuit (I2C) Module Reference Guide describes the features and operation of the inter-integrated circuit (I2C) module.

Tools Guides—

SPRUEK1 — TMS320C28x Assembly Language Tools v5.0.0 User’s Guide describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C28x device.

SPRUEK2 — TMS320C28x Optimizing C/C++ Compiler v5.0.0 User’s Guide describes the TMS320C28x™ C/C++ compiler. This compiler accepts ANSI standard C/C++ source code and produces TMS320 DSP assembly language source code for the TMS320C28x device.

SPRUEO2 — TMS320C28x Instruction Set Simulator Technical Overview describes the simulator, available within the Code Composer Studio for TMS320C2000 IDE, that simulates the instruction set of the C28x™ core.
SPRU625 — TMS320C28x DSP/BIOS 5.32 Application Programming Interface (API) Reference Guide describes development using DSP/BIOS.
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.

### Topic

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction to the Central Processing Unit (CPU)</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Compatibility with the C28x Fixed-Point CPU</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Components of the C28x plus Floating-Point CPU</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Memory Interface</td>
<td>11</td>
</tr>
</tbody>
</table>
1.1 Introduction to the Central Processing Unit (CPU)

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.

1.2 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.
1.2.1 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.

1.3 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 3-1
- 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 x 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.
1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.4 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.

1.4.1 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.

### 1.4.2 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.
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 CPU Registers</td>
<td>15</td>
</tr>
</tbody>
</table>
## 2.1 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 2-1 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 2-1. 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></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>

FPU registers R0H - R7H and STF are shadowed for fast context save and restore.
<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>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>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.
2.1.1 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 2-2. 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 forced to 0 by the RESTORE instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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 zero, the MPYF32, ADDF32 and SUBF32 instructions will round to zero (truncate).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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>
<tr>
<td></td>
<td></td>
<td></td>
<td>The condition tested with the TESTTF instruction is false.</td>
</tr>
</tbody>
</table>
```

Table 2-2. Floating-point Unit Status (STF) Register Field Descriptions
### Table 2-2. 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>
</table>
| 5    | ZI    | Zero Integer Flag  
|      |       | The following instructions modify this flag based on the integer value stored in the destination register: MOV32, MOVD32, MOVD32  
|      |       | The SETFLG and SAVE instructions can also be used to modify this flag.  
|      |       | 0 | The integer value is not zero.  
|      |       | 1 | The integer value is zero.  
| 4    | NI    | Negative Integer Flag  
|      |       | The following instructions modify this flag based on the integer value stored in the destination register: MOV32, MOVD32, MOVD32  
|      |       | The SETFLG and SAVE instructions can also be used to modify this flag.  
|      |       | 0 | The integer value is not negative.  
|      |       | 1 | The integer value is negative.  
| 3    | ZF    | Zero Floating-Point Flag  
|      |       | (1) (2)  
|      |       | The following instructions modify this flag based on the floating-point value stored in the destination register: MOV32, MOVD32, MOVD32, ABSF32, NEGF32  
|      |       | The CMPF32, MAXF32, and MINF32 instructions modify this flag based on the result of the operation.  
|      |       | The SETFLG and SAVE instructions can also be used to modify this flag.  
|      |       | 0 | The floating-point value is not zero.  
|      |       | 1 | The floating-point value is zero.  
| 2    | NF    | Negative Floating-Point Flag  
|      |       | (1) (2)  
|      |       | The following instructions modify this flag based on the floating-point value stored in the destination register: MOV32, MOVD32, MOVD32, ABSF32, NEGF32  
|      |       | The CMPF32, MAXF32, and MINF32 instructions modify this flag based on the result of the operation.  
|      |       | The SETFLG and SAVE instructions can also be used to modify this flag.  
|      |       | 0 | The floating-point value is not negative.  
|      |       | 1 | The floating-point value is negative.  
| 1    | LUF   | Latched Underflow Floating-Point Flag  
|      |       | The following instructions will set this flag to 1 if an underflow occurs: MPYF32, ADDF32, SUBF32, MACF32, EINVF32, EISQRTF32  
|      |       | 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  
|      |       | The following instructions will set this flag to 1 if an overflow occurs: MPYF32, ADDF32, SUBF32, MACF32, EINVF32, EISQRTF32  
|      |       | 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.  

(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.
2.1.2 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 2-2.

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

; 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 2-3. Repeat Block Register (RB)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31-30 | RAS | 0-1 | Repeat Block Active Shadow Bit
| | | 0 | When an interrupt occurs the repeat active, RA, bit is copied to the RAS bit and the RA bit is cleared.
| | | 1 | A repeat block was active when the interrupt was taken.
| | | R-0 | |
| 29-23 | RSIZE | 0-7 | Repeat Block Size
| | | 0-7 | 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.
| | | 8/9-0x7F | Illegal block size.
| | | R-0 | 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. |
### 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>
</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.  |
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Pipeline Overview</td>
<td>23</td>
</tr>
<tr>
<td>3.2 General Guidelines for Floating-Point Pipeline Alignment</td>
<td>23</td>
</tr>
<tr>
<td>3.3 Moves from FPU Registers to C28x Registers</td>
<td>24</td>
</tr>
<tr>
<td>3.4 Moves from C28x Registers to FPU Registers</td>
<td>24</td>
</tr>
<tr>
<td>3.5 Parallel Instructions</td>
<td>25</td>
</tr>
<tr>
<td>3.6 Invalid Delay Instructions</td>
<td>25</td>
</tr>
<tr>
<td>3.7 Optimizing the Pipeline</td>
<td>28</td>
</tr>
</tbody>
</table>
### 3.1 Pipeline Overview

The C28x FPU pipeline is identical to the C28x pipeline for all standard C28x instructions. In the decode 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 3-1. 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 3-1. FPU Pipeline](image)

### 3.2 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 3-1. 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.
3.3 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 3-2 and Example 3-3.

Example 3-2. 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 3-2. 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 3-3.

Example 3-3. 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
MOV32 @ACC, R0H ; Copy R0H to ACC
```

3.4 Moves from C28x Registers to FPU Registers

Transfers from the standard C28x CPU registers to the floating-point registers require four alignment cycles. In this case the four alignment cycles can be filled with NOPs or any non-conflicting instruction except for FRACF32, U16TOF32, I16TOF32, F32TOI32, and F32TOI32. These instructions cannot replace any of the four alignment NOPs.

Example 3-4. 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
```
3.5 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 3-5.

Example 3-5. 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 3-6.

Example 3-6. 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
```

3.6 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 3-7.

In Example 3-7 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 3-8.
Example 3-7. 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 3-8. Destination Register Conflict Resolved

; Valid delay instruction
MPYF32  R2H, R1H, R0H ; 2p instruction
MOV32   R1H, 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 3-9. For parallel instructions, the current value of a register can be used in the parallel operation before it is overwritten as shown in Example 3-11.

In Example 3-9 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 3-10.

Example 3-9. 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 3-10. 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 3-11 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 3-11. Parallel Instruction Destination/Source Exception

; 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 3-12 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 3-12. Parallel Instruction Destination/Source Exception

; Valid parallel instruction
  MPYF32 R2H, R1H, R0H ; 2p/1 instruction
  MOV32 R1H, mem32 ; Valid
                   ; <-- 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 3-13.

If both operations within a parallel instruction try to update the same destination register as shown in Example 3-13 the assembler will issue an error.

Example 3-13. 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.
3.7 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 3-14, 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 3-14. 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 @Y2,R1H ; Save R1H in Y2
;
; 14 cycles
; 48 bytes
```

The code shown in Example 3-15 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 9 cycles.
Example 3-15. 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
MOV32 R2H,@B2 ; Load R2H with B2 - single cycle
; <-- MPYF32 completes, R0H is valid
ADDF32 R0H,R2H,R0H ; R0H = B2 + M2*X2 - 2p operation
MOV32 @Y1,R1H ; Store Y1
; <-- ADDF32 completes, R1H is valid
MOV32 @Y2,R0H ; Store Y2

; 9 cycles
; 36 bytes
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).

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Instruction Descriptions</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Instructions</td>
<td>34</td>
</tr>
</tbody>
</table>
4.1 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 4-1. For information on the operands of standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430).

Table 4-1. 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>#16FHiiHex</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)</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 Chapter 3.

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.
4.2 Instructions

The instructions are listed alphabetically, preceded by a summary.

Table 4-2. 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>36</td>
</tr>
<tr>
<td>ADDF32 RaH, #16FHi, RbH —32-bit Floating-Point Addition</td>
<td>37</td>
</tr>
<tr>
<td>ADDF32 RaH, RbH, #16FHi —32-bit Floating-Point Addition</td>
<td>39</td>
</tr>
<tr>
<td>ADDF32 RaH, RbH, RcH —32-bit Floating-Point Addition</td>
<td>41</td>
</tr>
<tr>
<td>ADDF32 RdH, ReH, RfH</td>
<td></td>
</tr>
<tr>
<td>ADDF32 RdH, ReH, RfH</td>
<td></td>
</tr>
<tr>
<td>CMPF32 RaH, RbH —32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>47</td>
</tr>
<tr>
<td>CMPF32 RaH, #16FHi —32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>48</td>
</tr>
<tr>
<td>CMPF32 RaH, #0.0 —32-bit Floating-Point Compare for Equal, Less Than or Greater Than</td>
<td>50</td>
</tr>
<tr>
<td>EINV32 RaH, RbH —32-bit Floating-Point Reciprocal Approximation</td>
<td>51</td>
</tr>
<tr>
<td>EISQRTF32 RaH, RbH —32-bit Floating-Point Square-Root Reciprocal Approximation</td>
<td>53</td>
</tr>
<tr>
<td>F32TOI16 RaH, RbH —Convert 32-bit Floating-Point Value to 16-bit Integer</td>
<td>55</td>
</tr>
<tr>
<td>F32TOI16R RaH, RbH —Convert 32-bit Floating-Point Value to 16-bit Integer and Round</td>
<td>56</td>
</tr>
<tr>
<td>F32TOI32 RaH, RbH —Convert 32-bit Floating-Point Value to 32-bit Integer</td>
<td>57</td>
</tr>
<tr>
<td>F32TOU16 RaH, RbH —Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer</td>
<td>58</td>
</tr>
<tr>
<td>F32TOU16R RaH, RbH —Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round</td>
<td>59</td>
</tr>
<tr>
<td>F32TOU32 RaH, RbH —Convert 32-bit Floating-Point Value to 32-bit Unsigned Integer</td>
<td>60</td>
</tr>
<tr>
<td>FRACF32 RaH, #16FHi —Fractional Portion of a 32-bit Floating-Point Value</td>
<td>61</td>
</tr>
<tr>
<td>I16TOF32 RaH, RbH —Convert 16-bit Integer to 32-bit Floating-Point Value</td>
<td>62</td>
</tr>
<tr>
<td>I16TOF32 RaH, mem16 —Convert 16-bit Integer to 32-bit Floating-Point Value</td>
<td>63</td>
</tr>
<tr>
<td>I32TOF32 RaH, mem32 —Convert 32-bit Integer to 32-bit Floating-Point Value</td>
<td>64</td>
</tr>
<tr>
<td>I32TOF32 RaH, RbH —Convert 32-bit Integer to 32-bit Floating-Point Value</td>
<td>65</td>
</tr>
<tr>
<td>MACF32 R3H, R2H, RdH, ReH, RfH —32-bit Floating-Point Multiply with Parallel Add</td>
<td>66</td>
</tr>
<tr>
<td>MACF32 R3H, R2H, RdH, ReH, RfH</td>
<td></td>
</tr>
<tr>
<td>MACF32 R7H, R3H, mem32, *XAR7++ —32-bit Floating-Point Multiply and Accumulate</td>
<td>70</td>
</tr>
<tr>
<td>MACF32 R7H, R6H, RdH, ReH, RfH —32-bit Floating-Point Multiply with Parallel Add</td>
<td>72</td>
</tr>
<tr>
<td>MACF32 R7H, R6H, RdH, ReH, RfH</td>
<td></td>
</tr>
<tr>
<td>MAXF32 RaH, RbH —32-bit Floating-Point Maximum</td>
<td>76</td>
</tr>
<tr>
<td>MAXF32 RaH, #16FHi —32-bit Floating-Point Maximum</td>
<td>77</td>
</tr>
<tr>
<td>MAXF32 RaH, RbH</td>
<td></td>
</tr>
<tr>
<td>MINF32 RaH, RbH —32-bit Floating-Point Minimum</td>
<td>79</td>
</tr>
<tr>
<td>MINF32 RaH, #16FHi —32-bit Floating-Point Minimum</td>
<td>80</td>
</tr>
<tr>
<td>MINF32 RaH, RbH</td>
<td></td>
</tr>
<tr>
<td>MOV16 mem16, RaH —Move 16-bit Floating-Point Register Contents to Memory</td>
<td>82</td>
</tr>
<tr>
<td>MOV32 *(:016bitAddr), loc32 —Move the Contents of loc32 to Memory</td>
<td>83</td>
</tr>
<tr>
<td>MOV32 ACC, RaH —Move 32-bit Floating-Point Register Contents to ACC</td>
<td>84</td>
</tr>
<tr>
<td>MOV32 loc32, *(0:16bitAddr) —Move 32-bit Value from Memory to loc32</td>
<td>85</td>
</tr>
<tr>
<td>MOV32 mem32, RaH —Move 32-bit Floating-Point Register Contents to Memory</td>
<td>86</td>
</tr>
<tr>
<td>MOV32 mem32, STF —Move 32-bit STF Register to Memory</td>
<td>88</td>
</tr>
<tr>
<td>MOV32 P, RaH —Move 32-bit Floating-Point Register Contents to P</td>
<td>89</td>
</tr>
<tr>
<td>MOV32 RaH, ACC —Move the Contents of ACC to a 32-bit Floating-Point Register</td>
<td>90</td>
</tr>
<tr>
<td>MOV32 RaH, mem32</td>
<td></td>
</tr>
<tr>
<td>MOV32 RaH, P —Move the Contents of P to a 32-bit Floating-Point Register</td>
<td>93</td>
</tr>
</tbody>
</table>
Table 4-2. Summary of Instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV32 RaH, RbH {, CNDF}</td>
<td>Conditional 32-bit Move</td>
<td>94</td>
</tr>
<tr>
<td>MOV32 RaH, XARn</td>
<td>Move the Contents of XARn to a 32-bit Floating-Point Register</td>
<td>96</td>
</tr>
<tr>
<td>MOV32 RaH, XT</td>
<td>Move the Contents of XT to a 32-bit Floating-Point Register</td>
<td>97</td>
</tr>
<tr>
<td>MOV32 STF, mem32</td>
<td>Move 32-bit Value from Memory to the STF Register</td>
<td>98</td>
</tr>
<tr>
<td>MOV32 XARn, RaH</td>
<td>Move 32-bit Floating-Point Register Contents to XARn</td>
<td>99</td>
</tr>
<tr>
<td>MOV32 XT, RaH</td>
<td>Move 32-bit Floating-Point Register Contents to XT</td>
<td>100</td>
</tr>
<tr>
<td>MOV3D32 RaH, mem32</td>
<td>Move 32-bit Value from Memory with Data Copy</td>
<td>101</td>
</tr>
<tr>
<td>MOVIZ RaH, #16FiHex</td>
<td>Load the Upper 16-bits of a 32-bit Floating-Point Register</td>
<td>104</td>
</tr>
<tr>
<td>MOVIZF32 RaH, #16Hi</td>
<td>Load the Upper 16-bits of a 32-bit Floating-Point Register</td>
<td>105</td>
</tr>
<tr>
<td>MOVST0 FLAG</td>
<td>Load Selected STF Flags into ST0</td>
<td>106</td>
</tr>
<tr>
<td>MOVXI RaH, #16FLoHex</td>
<td>Move Immediate to the Low 16-bits of a Floating-Point Register</td>
<td>108</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RcH</td>
<td>32-bit Floating-Point Multiply</td>
<td>109</td>
</tr>
<tr>
<td>MPYF32 RaH, #16FiHi, RbH</td>
<td>32-bit Floating-Point Multiply</td>
<td>110</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, #16FiHi</td>
<td>32-bit Floating-Point Multiply</td>
<td>112</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RcH</td>
<td></td>
<td>ADDF32 RdH, ReH, RfH</td>
</tr>
<tr>
<td>MPYF32 RdH, ReH, RfH</td>
<td></td>
<td>MOV32 RaH, mem32</td>
</tr>
<tr>
<td>MPYF32 RdH, ReH, RfH</td>
<td></td>
<td>MOV32 mem32, RaH</td>
</tr>
<tr>
<td>MPYF32 RaH, RbH, RcH</td>
<td></td>
<td>SUBF32 RdH, ReH, RfH</td>
</tr>
<tr>
<td>NEGF32 RaH, RbH{, CNDF}</td>
<td>Conditional Negation</td>
<td>120</td>
</tr>
<tr>
<td>POP RB</td>
<td>Pop the RB Register from the Stack</td>
<td>121</td>
</tr>
<tr>
<td>PUSH RB</td>
<td>Push the RB Register onto the Stack</td>
<td>123</td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore the Floating-Point Registers</td>
<td>125</td>
</tr>
<tr>
<td>RPTB label, loc16</td>
<td>Repeat A Block of Code</td>
<td>127</td>
</tr>
<tr>
<td>RPTB label, #RC</td>
<td>Repeat a Block of Code</td>
<td>129</td>
</tr>
<tr>
<td>SAVE FLAG, VALUE</td>
<td>Save Register Set to Shadow Registers and Execute SETFLG</td>
<td>131</td>
</tr>
<tr>
<td>SETFLG FLAG, VALUE</td>
<td>Set or clear selected floating-point status flags</td>
<td>133</td>
</tr>
<tr>
<td>SUBF32 RaH, RbH, RcH</td>
<td>32-bit Floating-Point Subtraction</td>
<td>134</td>
</tr>
<tr>
<td>SUBF32 RaH, #16FiHi, RbH</td>
<td>32-bit Floating Point Subtraction</td>
<td>135</td>
</tr>
<tr>
<td>SUBF32 RdH, ReH, RfH</td>
<td></td>
<td>MOV32 RaH, mem32</td>
</tr>
<tr>
<td>SUBF32 RdH, ReH, RfH</td>
<td></td>
<td>MOV32 mem32, RaH</td>
</tr>
<tr>
<td>SWAPF RaH, RbH{, CNDF}</td>
<td>Conditional Swap</td>
<td>140</td>
</tr>
<tr>
<td>TESTTF CNDF</td>
<td>Test STF Register Flag Condition</td>
<td>141</td>
</tr>
<tr>
<td>UI16TOF32 RaH, mem16</td>
<td>Convert unsigned 16-bit integer to 32-bit floating-point value</td>
<td>142</td>
</tr>
<tr>
<td>UI16TOF32 RaH, RbH</td>
<td>Convert unsigned 16-bit integer to 32-bit floating-point value</td>
<td>143</td>
</tr>
<tr>
<td>UI32TOF32 RaH, mem32</td>
<td>Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value</td>
<td>144</td>
</tr>
<tr>
<td>UI32TOF32 RaH, RbH</td>
<td>Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value</td>
<td>145</td>
</tr>
<tr>
<td>ZERO RaH</td>
<td>Zero the Floating-Point Register RaH</td>
<td>146</td>
</tr>
<tr>
<td>ZEROA</td>
<td>Zero All Floating-Point Registers</td>
<td>147</td>
</tr>
</tbody>
</table>
ABSF32 RaH, RbH  32-bit Floating-Point Absolute Value

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 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.

if (RbH < 0) {RaH = -RbH}
else {RaH = 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>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The STF register flags are modified as follows:

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

MOVIZF32 ROH, #0.0   ; ROH = 0.0
ABSF32 R1H, ROH    ; 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>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>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, #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 ; 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
```
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, #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 #0/BFC0.

RaH = RbH + #16FHi

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>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, #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
```
Instructions

See also

ADDF32 RaH, #16FH, RbH
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

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0111 0001 0000</td>
<td>0000 000c ccbb baaa</td>
</tr>
</tbody>
</table>

Description

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

RaH = RbH + RcH

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 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, #16FHi, RbH
- ADDF32 RaH, RbH, #16FHi
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 RdH, ReH, RfH || MOV32 mem32, RaH  32-bit Floating-Point Addition with Parallel Move

Operands

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RdH</td>
<td>floating-point destination register for the ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>ReH</td>
<td>floating-point source register for the ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register for the ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location. This will be the destination of the MOV32.</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point source register for the MOV32 (R0H to R7H)</td>
</tr>
</tbody>
</table>

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
; <-- MOV32 completes, mem32 updated
NOP ; 1 cycle delay or non-conflicting instruction
; <-- ADDF32 completes, RdH updated
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] ;
; <-- 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
See also

ADDF32 RaH, #16FHHi, RbH
ADDF32 RaH, RbH, #16FHHi
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RdH</td>
<td>floating-point destination register for the ADDF32 (R0H to R7H). RdH cannot be the same register as RaH.</td>
</tr>
<tr>
<td>ReH</td>
<td>floating-point source register for the ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RfH</td>
<td>floating-point source register for the ADDF32 (R0H to R7H)</td>
</tr>
<tr>
<td>RaH</td>
<td>floating-point destination register for the MOV32 (R0H to R7H). RaH cannot be the same register as RdH.</td>
</tr>
<tr>
<td>mem32</td>
<td>pointer to a 32-bit memory location. This is the source for the MOV32.</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{MOVL} & \quad \text{XAR4,} \ #A \\
\text{MOV32} & \quad \text{R0H,} \ *\text{XAR4} \quad ; \text{Load R0H with A} \\
\text{MOVL} & \quad \text{XAR4,} \ #B \\
\text{MOV32} & \quad \text{R1H,} \ *\text{XAR4} \quad ; \text{Load R1H with B} \\
\text{MOVL} & \quad \text{XAR4,} \ #C \\
\text{ADDF32} & \quad \text{R0H,R1H,R0H} \quad ; \text{Add A + B and in parallel} \\
\text{MOV32} & \quad \text{R2H,} \ *\text{XAR4} \quad ; \text{Load R2H with C} \\
\text{MOV32} & \quad \text{XAR4,} \ #Y \\
\text{MOVL} & \quad \text{XAR4,} \ #Y \\
\text{ADDF32} & \quad \text{complete} \quad ; \text{ADDF32 complete} \\
\text{MOV32} & \quad \text{XAR4,} \ #Y \\
\text{MACF32} & \quad \text{R3H,} \ #16FHi, \ RbH, \ R2H, \ R0H, \ R1H \\
\text{ADDF32} & \quad \text{RdH,} \ ReH, \ RfH \quad ; \text{MOV32 complete, RaH} \\
\text{MPYF32} & \quad \text{R0H,} \ RbH, \ RcH \quad ; \text{ADDF32 complete, RdH, ReH, RfH} \\
\text{SUBF32} & \quad \text{R0H,R0H,R2H} \quad ; \text{Subtract C from (A + B)} \\
\text{NOP} & \quad \text{complete} \quad ; \text{SUBF32 completes} \\
\text{MOV32} & \quad \text{XAR4,} \ #Y \\
\text{ADDF32} & \quad \text{RaH,} \ #16FHi, \ RbH, \ #16FHi \\
\text{ADDF32} & \quad \text{RaH,} \ RbH, \ R0H, \ R1H, \ R2H \\
\text{ADDF32} & \quad \text{RdH,} \ ReH, \ RfH \quad ; \text{MOV32 complete, RaH} \\
\text{MACF32} & \quad \text{R3H,} \ R2H, \ RdH, \ ReH, \ R0H, \ R1H \\
\text{MPYF32} & \quad \text{R0H,} \ RbH, \ RcH \quad ; \text{ADDF32 complete, RdH, ReH, RfH} \\
\text{MOV32} & \quad \text{XAR4,} \ #Y \\
\end{align*}
\]

See also
CMPF32 RaH, RbH  32-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

<table>
<thead>
<tr>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source 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 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

```
MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
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 #16FHi 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 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></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
MOVIZ32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZ32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
MOVIZ32 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
MOVS10 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>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>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0110 1001 0011</td>
<td>0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

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:

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

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).

```
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
NEGFM 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:

Ye = 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 = Estimate of 1/sqrt (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 = 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*Ye*X*0.5
NOP
MPYF32 R1H, R1H, R3H ; R2H = Ye - Ye*(1.5 - Ye*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*Ye*X*0.5
CMPF32 R0H, #0.0 ; Check if X == 0
MPYF32 R1H, R1H, R3H ; R2H = Ye - Ye*(1.5 - Ye*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)
LAETr
```

See also

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

Operands

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

Opcode

<table>
<thead>
<tr>
<th></th>
<th>LSW: 1110 0110 1000 1100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

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
; <-- F32TOI16 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 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 (0xFFFF)
; R3H(31:16) = (0xFFFF)

See also

F32TOI16R RaH, RbH
F32TOU16 RaH, RbH
F32TOU16R 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></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
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 (0x8FD9999A)
; <-- 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 (0xFFFFE)
; 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**

<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**

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

**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.

\[ RaH(15:0) = \text{F32ToUI16}(RbH) \]
\[ RaH(31:16) = 0x0000 \]

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:

\[
\begin{align*}
\text{F32TOUI16} & \quad \text{RaH, RbH} \quad ; \quad 2 \text{ pipeline cycles (2p)} \\
\text{NOP} & \quad ; \quad 1 \text{ cycle delay or non-conflicting instruction} \\
\text{NOP} & \quad ; \quad \text{<-- F32TOUI16 completes, RaH updated} \\
\text{NOP} & \quad ; \quad \text{<-- F32TOUI16 complete, R5H (15:0) = 9.0 (0x0009)} \\
\text{F32TOUI16 complete, R5H (15:0)} & \quad = \quad 9.0 \ (0x0009) \\
\text{R5H (31:16)} & \quad = \quad 0.0 \ (0x0000) \\
\text{R5H (31:16)} & \quad = \quad 0.0 \ (0x0000) \\
\text{F32TOUI16 complete, R7H (15:0)} & \quad = \quad 0.0 \ (0x0000) \\
\text{R7H (31:16)} & \quad = \quad 0.0 \ (0x0000) \\
\text{R7H (31:16)} & \quad = \quad 0.0 \ (0x0000) \\
\end{align*}
\]

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

Example

\[
\begin{align*}
\text{MOVIZF32} & \quad \text{R4H, #9.0} \quad ; \quad \text{R4H} = \quad 9.0 \ (0x41100000) \\
\text{F32TOUI16} & \quad \text{R5H, R4H} \quad ; \quad \text{R5H (15:0)} = \quad \text{F32TOUI16} (\text{R4H}) \\
& \quad ; \quad \text{R5H (31:16)} = \quad 0x0000 \\
\text{MOVIZF32} & \quad \text{R6H, #-9.0} \quad ; \quad \text{R6H} = \quad -9.0 \ (0xC1100000) \\
& \quad \text{<-- F32TOUI16 complete, R5H (15:0)} = \quad 9.0 \ (0x0009) \\
& \quad ; \quad \text{R5H (31:16)} = \quad 0.0 \ (0x0000) \\
& \quad ; \quad \text{F32TOUI16 complete, R7H (15:0)} = \quad 0.0 \ (0x0000) \\
& \quad ; \quad \text{R7H (31:16)} = \quad 0.0 \ (0x0000) \\
\end{align*}
\]

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

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.

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></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
NOP ; <-- F32TOUI16R 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

MOVIZ R5H, #0x412C ; R5H = 0x412C
MOVXI 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

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

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:

```
FRACF32 RaH, RbH ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
```

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
```

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

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1000 1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

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  
U16TOF32 RaH, mem16  
U16TOF32 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>mem16</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
```

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, #0xFFFC ; [0x00A001] = -4.0 (0xFFFC)
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

<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>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 ; <-- 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

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

I32TOF32 R1H, @0 ; R1H = I32TOF32 (0x11111111)
NOP ; 1 cycle delay for I32TOF32 to complete
NOP ; <-- 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>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>
</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:

\[
\begin{align*}
&\text{MPYF32 } \text{RdH}, \text{RaH}, \text{RbH} \\
\text{||} &\text{ADDF32 } \text{R3H}, \text{R3H}, \text{R2H}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H</td>
<td>floating-point destination and source register for the ADDF32</td>
</tr>
<tr>
<td>R2H</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 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>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:

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

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
|| 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
MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
|| MOV32  R0H, *XAR4++
MOV32  R1H, *XAR5++ ; R1H = Y3
; R2H = A + B
; R3H = (A + B) + C
; R3H = A + B + C + D
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
;
; R2H = E = X4 * Y4
MACF32 R3H, R2H, R2H, R0H, R1H ; in parallel R3H = (A + B + C) + D
NOP ; Wait for MPYF32 || ADDF32 to complete
ADD532 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>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3H</td>
<td>floating-point destination/source register R3H for the add operation</td>
</tr>
<tr>
<td>R2H</td>
<td>floating-point source register R2H for the add operation</td>
</tr>
<tr>
<td>RdH</td>
<td>floating-point destination register (R0H to R7H) for the multiply operation</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 register as RdH.</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.

\[
R3H = R3H + R2H, \\
RdH = ReH \times RfH, \\
RaH = \text{[mem32]} 
\]

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 No Yes Yes Yes Yes Yes Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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:

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

Pipeline

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

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

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.
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

ADD0FF32 R3H, R3H, R2H
NOP ; Wait for MPYF32 | | ADD0FF32 to complete
ADD0FF32 R3H, R3H, R2H ; R3H = (A + B + C + D) + E
NOP ; Wait for ADD0FF32 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 | | ADD0FF32 RdH, ReH, RfH
**MACF32 R7H, R3H, mem32, *XAR7++  32-bit Floating-Point Multiply and Accumulate**

**Operands**

- R7H: floating-point destination register
- R3H: floating-point destination register
- mem32: pointer to a 32-bit source location
- *XAR7++: Indirect program-memory addressing using auxiliary register XAR7, can access full 4M x 16 program space range (0x000000 to 0x3FFFFF). XAR7 is post incremented.

**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++]

e tc...

**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:

```assembly
<instruction1> ; No restriction
<instruction2> ; Cannot be a 2p instruction that writes
    ; to R2H, R3H, R6H or R7H
RPT # (N - 1) ; Execute N times, where N is even
MACF32 R7H, R3H, *XAR6++, *XAR7++
<instruction3> ; 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:

- No restriction
- Cannot be a 2p instruction that writes to R2H, R3H, R6H or R7H

MACF32 R7H, R3H, *XAR6, *XAR7++ ; R3H = R3H + R2H, R2H = [mem32] * [XAR7++]

NOP

Example

ZERO R2H ; Zero the accumulation registers
ZERO R3H ; and temporary multiply storage registers
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 registers
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:

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

- **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)
- **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 dddd 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:
- 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:

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

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
; 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

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 ; 1 cycle
; <-- MOV32 completes, RaH updated
NOP ; 1 cycle delay
; <-- MACF32 completes, R7H, RdH updated
NOP

74 Instruction Set
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 ; Wait for MPYF32 | ADDF32 to complete
| NOP

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**

**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**

```c
MOVIZF32 R0H, #5.0 ; R0H = 5.0 (0x40A00000)
MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0x00BFC0000)
MOVIZF32 R2H, #-1.5 ; R2H = -1.5 (0x00BFC0000)
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>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source/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>
</tbody>
</table>

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.

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 #-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></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)
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>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source/destination register for the MAXF32 operation (R0H to R7H)</td>
</tr>
<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 RcH.

Opcode

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1001 1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 dddc cccb baaa</td>
</tr>
</tbody>
</table>

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 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>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 (0x8FC00000)
MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0xC0000000)
MAXF32 R0H, R1H ; R0H = R1H = 5.0, R3H = -1.5, ZF = 0, NF = 0
|| MOV32 R3H, R2H
MAXF32 R1H, R0H ; R1H = R0H = 5.0, R3H = -1.5, ZF = 0, NF = 1
|| MOV32 R3H, R2H
MAXF32 R0H, R1H ; R0H = R1H = 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source/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 0111  
**MSW:** 0000 0000 00bb baaa

## 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 ; ROH = 4.0, ZF = 0, NF = 0
MINF32 R1H, R2H ; RIH = −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, #16FH
- MINF32 RaH, #16FH
- 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.

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

#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 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.

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

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 = 5.0, 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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point source/destination register for the MIN32 operation (R0H to R7H)</td>
</tr>
<tr>
<td>RbH</td>
<td>floating-point source register for the MIN32 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>
<tr>
<td>RaH cannot be the same register as RcH</td>
<td></td>
</tr>
<tr>
<td>RbH cannot be the same register as RdH</td>
<td></td>
</tr>
<tr>
<td>RcH cannot be the same register as RaH</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 (0xBFC00000)
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.

[mem16] = 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**

```plaintext
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

<table>
<thead>
<tr>
<th>Acc</th>
<th>28x accumulator</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

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.

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 2 cycles
NOP ; Any instruction
```

Example

```
MOVIZF32 R0H, #2.5 ; R0H = 2.5 = 0x40200000
F32TOUT32 R0H, R0H
NOP ; Delay for conversion instruction
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

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

### Description

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

\[ [\text{loc32}] = [0:16bitAddr] \]

### 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

```assembly
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>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaH</td>
<td>floating-point register (R0H to R7H)</td>
</tr>
<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></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 ; R6H = A * X0 * Y0
| | MOV32 R0H, *XAR4++
| | MOV32 R1H, *XAR5++ ; R1H = Y1

| ||
| | MPYF32 R7H, R0H, R1H ; R7H = B * X1 * Y1
| | MOV32 R0H, *XAR4++
| | MOV32 R1H, *XAR5++ ; R1H = Y2

| ||
| | MACF32 R7H, R6H, R6H, R0H, R1H ; R3H = (A + B) + C
| | MOV32 R0H, *XAR4++
| | MOV32 R1H, *XAR5++ ; R1H = Y3

| ||
| | MACF32 R7H, R6H, R6H, R0H, R1H ; R6H = (A + B) + C
| | MOV32 R0H, *XAR4++
| | MOV32 R1H, *XAR5++ ; R1H = Y4

| ||
| | MPYF32 R6H, R0H, R1H ; R7H = (A + B + C) + D
| | ADDF32 R7H, R7H, R2H
| | 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

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.

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

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>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
MOVIZF32 R0H, #2.0 ; R0H = 2.0 (0x40000000)
MOVIZF32 R1H, #3.0 ; R1H = 3.0 (0x40040000)
CMPF32 R0H, R1H ; ZF = 0, NF = 1, STF = 0x00000004
MOV32 @0, STF ; [0x00A000] = 0x00000004

Example 2

MOV32 *SP++, STF ; Store STF in stack
MOV32 R2H, #3.0 ; R2H = 3.0 (0x40000000)
MOV32 R3H, #5.0 ; R3H = 5.0 (0x40A00000)
CMPF32 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**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P</strong></td>
<td>28x product register P</td>
</tr>
<tr>
<td><strong>RaH</strong></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:

```plaintext
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:

```plaintext
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**

```plaintext
MOVIZF32 R0H, #2.5  ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP            ; Delay for conversion instruction
NOP            ; -- Conversion complete, R0H valid
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

| RaH       | floating-point destination register (R0H to R7H) |
| ACC       | accumulator                                      |

### 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>
<th>Modified</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

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,@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

```assembly
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

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0010 1010 CNDF</td>
<td>0000 0aaa mem32</td>
</tr>
</tbody>
</table>

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</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>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

```c
if (CNDF == UNCF) {
    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;
} 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
MOVISF32 R3H, #7.0 ; R3H = 7.0 (0x40E00000)
MOVISF32 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               ;  
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.

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

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>

\[
\text{if(CNDF == UNCF)}
\]

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

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 IIII IIII IIII

**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 ;
ADDF32 R2H, R1H, R0H ; <= R0H is valid
```

**Example**

```
MOVL XAR1, #0x0200 ; XAR1 = 512
MOV32 R0H, XAR1
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**

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

**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.

```assembly
MOV32 R0H, XT ; Copy XT to R0H
NOP ; Wait 4 alignment cycles
NOP ; Do not use FRACF32, UI16TOF32
NOP ; I16TOF32, F32TOUI32 or F32TOI32
NOP ;
ADD F32 R2H,R1H,R0H ; instruction can use R0H as a source
```

**Example**

```assembly
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>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STF</td>
<td>floating-point unit status register</td>
</tr>
<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>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF</td>
<td>Yes</td>
</tr>
<tr>
<td>ZI</td>
<td>Yes</td>
</tr>
<tr>
<td>NI</td>
<td>Yes</td>
</tr>
<tr>
<td>ZF</td>
<td>Yes</td>
</tr>
<tr>
<td>NF</td>
<td>Yes</td>
</tr>
<tr>
<td>LUF</td>
<td>Yes</td>
</tr>
<tr>
<td>LVF</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)
CMPF32 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 IIIO IIII IIII

**Description**

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

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

**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:

```plaintext
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:

```plaintext
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 ; 2p completes, ACC is valid
NOP ; Any instruction
```

**Example**

```plaintext
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: 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>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 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                   ; -- ADDF32 completes, R2H is valid
MOV32 XT, 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
F32TOUTI32 R0H, R0H NOP ; Delay for conversion instruction
NOP                    ; -- Conversion complete, R0H valid
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

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 register (R0H to R7H)
#32FHex A 32-bit immediate value that represents an IEEE 32-bit floating-point value.

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>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

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, #0x00004040 ; R4H = 0x00004040
; 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. #-.15 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

MOVIZF32 ROH, #3.0 ; ROH = 3.0 = 0x40400000
MOVIZF32 RH1, #1.0 ; RH1 = 1.0 = 0x3F800000
MOVIZF32 RH2, #2.5 ; RH2 = 2.5 = 0x40200000
MOVIZF32 RH3, #-5.5 ; RH3 = -5.5 = 0xC0B00000
MOVIZF32 RH4, #0xC0B0 ; RH4 = -5.5 = 0xC0B00000

; Load R5H with pi = 3.141593 (0x40490000)
; MOVIZF32 R5H, #3.141593 ; R5H = 3.140625 (0x40490000)

; Load ROH with a more accurate pi = 3.141593 (0x40490FDB)
; MOVIZF32 ROH, #0x4049 ; ROH = 0x40490000
MOVXI ROH, #0x0FDB ; ROH = 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.

```plaintext
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 1111
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

; 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
- **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 0111 0000 0000
- **MSW**: 0000 000c ccbb baaa

### Description
Multiply the contents of two floating-point registers.

\[ RaH = RbH \times RcH \]

### 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 \( Y = A \times B \):

```
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 \times 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>RbH</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 #16FHi 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
Instructions

See also

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

**Operands**

- **RaH**: floating-point destination register (R0H to R7H)
- **RbH**: 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 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 (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 #0xBF0.

\[ RaH = RbH \times #16FHi:0 \]

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:

\[ \text{MPYF32 RaH, RbH, #16FHi ; 2 pipeline cycles (2p)} \]
\[ \text{NOP ; 1 cycle delay or non-conflicting instruction} \]
\[ \text{NOP \langle\rightarrow MPYF32 completes, RaH updated} \]
\[ \text{NOP} \]

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

**Example 1**

\[ \text{MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)} \]
\[ \text{MPYF32 R4H, R3H, #3.0 ; R4H = R3H \times 3.0} \]
\[ \text{MOVL XAR1, #0xB008 \langle\rightarrow Non conflicting instruction} \]
\[ \text{MOVL XAR1, #0xB008 \langle\rightarrow MPYF32 complete, R4H = 6.0 (0x40C00000)} \]
\[ \text{MOV32 *XAR1, R4H ; Save the result in memory location 0xB008} \]

**Example 2**

;Same as above example but #16FHi is represented in Hex
\[ \text{MOVIZF32 R3H, #2.0 ; R3H = 2.0 (0x40000000)} \]
\[ \text{MPYF32 R4H, R3H, #0x4040 ; R4H = R3H \times 0x4040} \]
\[ 3.0 \text{ is represented as 0x40400000 in IEEE 754 32-bit format} \]
\[ \text{MOVL XAR1, #0xB008 \langle\rightarrow Non conflicting instruction} \]
\[ \text{MOVL XAR1, #0xB008 \langle\rightarrow MPYF32 complete, R4H = 6.0 (0x40C00000)} \]
\[ \text{MOV32 *XAR1, R4H ; Save the result in memory location 0xB008} \]
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

| RaH   | floating-point destination register for MPYF32 (R0H to R7H) |
| RaH cannot be the same register as RdH |
| 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 ADDF32 (R0H to R7H) |
| RdH cannot be the same register as RaH |
| ReH   | floating-point source register for ADDF32 (R0H to R7H) |
| RfH   | floating-point source register for ADDF32 (R0H to R7H) |

Opcode

| LSW: 1110 0111 0100 00ff |
| MSW: feee dddc ccb 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

<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

; R2H = A = X0 * 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

; 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

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

| ADDF32 R3H, R3H, R2H ; R3H = (A + B + C + D) + E
| NOP ; Wait for ADDF32 to complete
| NOP ; Store the result

| MOV32 @Result, R3H

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**

```
LSW: 1110 0011 0000 fffe
MSW: eedd daaa mem32
```

**Description**

Multiply the contents of two floating-point registers and load another.

\[
\text{RdH} = \text{ReH} \times \text{RfH} \\
\text{RaH} = \text{[mem32]}
\]

**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></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:

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

**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 \\
; <-- 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 \( Y1 \) are all on the same data page.

\[
\begin{align*}
\text{MOVW} & \quad DP, \#M1 \quad ; \text{Load the data page} \\
\text{MOV32} & \quad R0H, @M1 \quad ; \text{Load } R0H \text{ with } M1 \\
\text{MOV32} & \quad R1H, @X1 \quad ; \text{Load } R1H \text{ with } X1 \\
\text{MPYF32} & \quad R1H, R1H, R0H \quad ; \text{Multiply } M1 \times X1 \\
| | \quad \text{MOV32} & \quad R0H, @B1 \quad ; \text{and in parallel load } R0H \text{ with } B1 \\
| | \quad \text{NOP} & \quad ; \text{Wait 1 cycle for MPYF32 to complete} \\
\text{ADDF32} & \quad R1H, R1H, R0H \quad ; \text{Add } M1 \times X1 \text{ to } B1 \text{ and store in } R1H \\
\text{NOP} & \quad ; \text{Wait 1 cycle for ADDF32 to complete} \\
\text{MOV32} & \quad @Y1, R1H \quad ; \text{Store the result}
\end{align*}
\]

Calculate \( Y = (A \times B) \times C \):

\[
\begin{align*}
\text{MOVL} & \quad XAR4, \#A \quad ; \text{Load } R0H \text{ with } A \\
\text{MOV32} & \quad R0H, *XAR4 \quad ; \text{Load } R0H \text{ with } A \\
\text{MOVL} & \quad XAR4, \#B \quad ; \text{Load } R1H \text{ with } B \\
\text{MOV32} & \quad R1H, *XAR4 \quad ; \text{Load } R1H \text{ with } B \\
\text{MOVL} & \quad XAR4, \#C \quad ; \text{Load } R2H \text{ with } C \\
\text{MPYF32} & \quad R1H, R1H, R0H \quad ; \text{Calculate } R1H = A \times B \\
| | \quad \text{MOV32} & \quad R0H, *XAR4 \quad ; \text{and in parallel load } R2H \text{ with } C \\
| | \quad \text{NOP} & \quad ; \text{Wait 1 cycle for MPYF32 to complete} \\
\text{MPYF32} & \quad R2H, R1H, R0H \quad ; \text{Calculate } Y = (A \times B) \times C \\
\text{NOP} & \quad ; \text{Wait 1 cycle for MPYF32 to complete} \\
\text{MOV32} & \quad @Y1, R2H \quad ; \text{Store the result}
\end{align*}
\]

See also  
\text{MPYF32 RdH, ReH, RfH || MOV32 mem32, RaH} \\
\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++}
**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.

\[ RdH = ReH \times RfH, \]
\[ \text{[mem32]} = RaH \]

**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:

\[
\begin{align*}
\text{MPYF32 } & \text{RdH, ReH, RfH} \quad ; \text{2 pipeline cycles (2p)} \\
\text{|| } & \text{MOV32 mem32, RaH} \quad ; \text{1 cycle} \\
& \text{; <-- MOV32 completes, mem32 updated} \\
& \text{NOP} \quad ; \text{1 cycle delay or non-conflicting instruction} \\
& \text{; <-- MPYF32 completes, RdH updated} \\
& \text{NOP} \\
\end{align*}
\]

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

**Example**

\[
\begin{align*}
\text{MOVXZ32 } & \text{R3H, #2.0} \quad ; \text{R3H = 2.0 (0x40000000)} \\
\text{MPYF32 } & \text{R3H, R3H, #5.0} \quad ; \text{R3H = R3H \times 5.0} \\
\text{MOVIZF32 } & \text{R1H, #5.0} \quad ; \text{R1H = 5.0 (0x40A00000)} \\
& \text{; <-- MPYF32 complete, R3H = 10.0 (0x41200000)} \\
\text{MPYF32 } & \text{R3H, R1H, R3H} \quad ; \text{R3H = R1H \times R3H} \\
& \text{; and in parallel store previous R3 value} \\
& \text{; MOV32 complete, [0xC003] = 0x4120,} \\
& \text{; [0xC002] = 0x0000} \\
& \text{NOP} \quad ; \text{1 cycle delay for MPYF32 to complete} \\
& \text{; <-- MPYF32, R3H = 50.0 (0x42480000)} \\
\end{align*}
\]

**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)
RaH cannot be the same register as RdH
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)
RdH cannot be the same register as RaH
ReH  floating-point source register for SUBF32 (R0H to R7H)
RfH  floating-point source register for SUBF32 (R0H to R7H)

Opcode

LSW: 1110 0111 0101 00ff
MSW: feee dddc ccbb baaa

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

MOVIZF32 R4H, #5.0  ; R4H = 5.0 (0x40A00000)
MOVIZF32 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

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</td>
<td>Unconditional with flag modification</td>
<td>None</td>
</tr>
</tbody>
</table>

Flag

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 (0x8BFC0000)
MPYF32 R4H, R1H, R2H ; R4H = -6.0
MPYF32 R5H, R0H, R1H ; R5H = 20.0
CMPF32 R4H, #0.0 ; NF = 1
CMPF32 R5H, #0.0 ; NF = 0
NEGF32 R4H, R4H, LT ; if NF = 1, R4H = 6.0
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)
_INTERRUPT:       ; RAS = RA, RA = 0
...                
    PUSH RB         ; Save RB register only if a RPTB block is used
    ...            ; ISR
    ...            ; 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
    ...            ; Enable interrupts only after saving RB
    CLRC INTM       ; 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 #RSIZE, RC
RPTB #RSIZE, 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
    ...
    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
    ...
    CLI          ; 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 #SIZE, RC
RPTB #SIZE, 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.

``` assembly
; 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.
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>Operand</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

- **LSW:** 1011 0101 0bbb bbbb
- **MSW:** 0000 0000 loc16

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 preceding 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 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
    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.
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 ; ISR may or may not include a RPTB block
...  
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>Operands</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>#RC</td>
<td>16-bit immediate value for the repeat count.</td>
</tr>
</tbody>
</table>

Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW: 1011 0101 1bbb bbbb</td>
<td></td>
</tr>
<tr>
<td>MSW: cccc cccc cccc cccc</td>
<td></td>
</tr>
</tbody>
</table>

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 int 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.

```
; 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
MOVVL 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.
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 (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
MOVM32 R1H,*XAR0++ ; 8 or 9 words <= block size <= 127 words
MAXF32 R0H,R1H
MOVST0 NF,2F
MOVL XAR6,ACC,LT

VECTOR_MAX_END: ; RE indicates the end address
; RA is cleared

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.

```plaintext
; The following is INVALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
SAVE RNDF32=1 ; INVALID, do not use SAVE in a delay slot
```

```plaintext
; The following is VALID
MPYF32 R2H, R1H, R0H ; 2 pipeline-cycle instruction (2p)
NOP ; 1 delay cycle, R2H updated after this instruction
SAVE 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 more legible, the assembler will accept a FLAG=VALUE syntax for the STFLG operation as shown below:

```plaintext
SAVE RNDF32=0, TF=1, ZF=0 ; FLAG = 01001000100, VALUE = X0XX0XXX1XX
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.
_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>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>reserved</td>
</tr>
<tr>
<td>9</td>
<td>RNDF32</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.

```assembly
; 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
```

```assembly
; 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 SETFLG operation as shown below:

```assembly
SETFLG RNDF32=0, TF=0, ZF=0 ; FLAG = 010010010000, 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

| Opcode | LSW: 1110 0111 0010 0000  
|        | MSW: 0000 000c ccb 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
    -- MOV32 complete
MOVL XAR4,#_xt
    -- 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**

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 (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.

**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, #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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RdH</td>
<td>floating-point destination register (R0H to R7H) for the SUBF32 operation</td>
</tr>
<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.

\[
\text{RdH} = \text{ReH} - \text{RfH}, \\
\text{RaH} = \text{[mem32]}
\]

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:

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

Pipeline

SUBF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

\[
\begin{align*}
\text{SUBF32} & \quad \text{RdH, ReH, RfH} \quad ; \text{2 pipeline cycles (2p)} \\
\text{|| MOV32} & \quad \text{RaH, mem32} \quad ; \text{1 cycle} \\
\text{NOP} & \quad ; \text{1 cycle delay or non-conflicting instruction} \\
\text{NOP} & \quad ; \text{MOV32 completes, RaH updated} \\
\text{NOP} & \quad ; \text{SUBF32 completes, RdH updated}
\end{align*}
\]

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.
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
MOV L XAR2, #0xE000 ;
  ; <-- (B) completes, R5H valid
MOV32 *XAR2, R5H ;
  ; <-- R5 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

- **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.

\[
RdH = ReH - RfH,
\]

\[
[mem32] = RaH
\]

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:

\[
\text{SUBF32} \quad \text{RdH, ReH, RfH} \quad ; \quad 2 \text{ pipeline cycles (2p)}
\]

\[
\text{MOV32} \quad \text{mem32, RaH} \quad ; \quad 1 \text{ cycle}
\]

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

Example

\[
\text{ADDF32} \quad \text{R3H, R6H, R4H} \quad ; \quad (A) \ R3H = R6H + R4H \ \text{and R7H = I3}
\]

\[
\text{MOV32} \quad \text{R7H, \ \text{SP}[2]} \quad ;
\]

\[
\text{SUBF32} \quad \text{R3H, R1H, R7H} \quad ; \quad (C) \ R3H = R1H - R7H \ \text{and store R3H (A)}
\]

\[
\text{MOV32} \quad \text{**XAR5[2], R3H} \quad ;
\]

\[
\text{ADDRO} \quad \text{R4H, R7H, R1H} \quad ; \quad (B) \ R4H = R7H + R1H \ \text{and store R6H (B)}
\]

\[
\text{MOV32} \quad \text{**XAR5[6], R6H} \quad ;
\]

\[
\text{MOV32} \quad \text{**XAR5[0], R3H} \quad ;
\]

\[
\text{MOV32} \quad \text{**XAR5[4], R4H} \quad ;
\]
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.

\[
\text{if } (\text{CNDF} \equiv \text{true}) \text{ swap RaH and RbH}
\]

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

```
MOV L XAR1, #0x8000 ;
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
CMPF32 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.

```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 (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>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
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
MOVS T0 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: 1110 0010 1100 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 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></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
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.ADCRESULT0 * m + b;
;
    MOVW DP @0x01C4
    UI16TOF32 R0H, @8              ; R0H = (float)AdcRegs.RESULT0
    MOV32 R1H, -*SP[6]             ; R1H = M
    MPYF32 R0H, R1H, R0H           ; R0H = (float)X * M
    MOV32 R1H, -*SP[8]             ; R1H = B
    ADDF32 R0H, R0H, R1H           ; R0H = Y = (float)X * M + B
    NOP
    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**

<table>
<thead>
<tr>
<th>LSW: 1110 0110 1000 1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW: 0000 0000 00bb baaa</td>
</tr>
</tbody>
</table>

**Description**

RaH = UI16ToF32[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>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

<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>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**

<table>
<thead>
<tr>
<th>LSW</th>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 0110 1000 1011</td>
<td>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

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

;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+1]);
;}
;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] * 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
ZEROA  
Zero All Floating-Point Registers

Operands
none

Opcode
LSW: 1110 0101 0110 0011

Description
Zero all floating-point registers:

\[
\begin{align*}
R0H &= 0 \\
R1H &= 0 \\
R2H &= 0 \\
R3H &= 0 \\
R4H &= 0 \\
R5H &= 0 \\
R6H &= 0 \\
R7H &= 0 \\
\end{align*}
\]

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>Mod</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

ZER0A ; 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]
    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]
    ADDDF32 R4H, R4H, R6H ; R4H += x[2*i] * y[2*i]
    MPYF32 R0H, R0H, R3H ; R0H = x[2*i] * y[2*i+1]
    ADDDF32 R5H, R5H, R1H ; R5H += y[2*i] * x[2*i+2]
    SUBDF32 R4H, R4H, R2H ; R4H -= x[2*i+1] * y[2*i+1]
    ADDDF32 R5H, R5H,R0H ; R5H += x[2*i] * y[2*i+1]
    BANZ LOOP , AR7--
```

See also
ZERO RaH
A.1 Changes

This revision history lists the technical changes made in the most recent revision.

**Table A-1. Technical Changes Made in This Revision**

<table>
<thead>
<tr>
<th>Location</th>
<th>Additions, Deletions, Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACF32R7HR3Hmem32XAR7</td>
<td>Modified the MACF32 R7H, R3H instructions; *XAR7 to *XAR7++; modified the description</td>
</tr>
<tr>
<td>RPTBRSIZEloc16</td>
<td>Modified the paragraph beginning &quot;The minimum size for the repeat block...&quot;</td>
</tr>
<tr>
<td>I16TOF32RaHmem16</td>
<td>Changed the Opcode LSW values</td>
</tr>
<tr>
<td>MACF32R7HR3Hmem32XAR7</td>
<td>Changed the Opcode MSW values; other typos</td>
</tr>
<tr>
<td>UI16TOF32RaHmem16</td>
<td>Delete the first sentence of the description; kept only RaH = UI16TOF32[mem16]</td>
</tr>
<tr>
<td>UI16TOF32RaHRbH</td>
<td>Delete the first sentence of the description; kept only RaH = UI16ToF32[RbH]</td>
</tr>
<tr>
<td>UI32TOF32RaHRbH</td>
<td>In the description, changed Rbh to [RbH]</td>
</tr>
</tbody>
</table>
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer’s risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

If TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

"Products" and "Applications" links:

- Audio: www.ti.com/audio
- Amplifiers: amplifier.ti.com
- Data Converters: dataconverter.ti.com
- DLP® Products: www.dlp.com
- DSP: dsp.ti.com
- Clocks and Timers: www.ti.com/clocks
- Interface: interface.ti.com
- Logic: logic.ti.com
- Power Mgmt: power.ti.com
- Microcontrollers: microcontroller.ti.com
- RFID: www.ti-rfid.com
- OMAP Applications Processors: www.ti.com/omap
- Wireless Connectivity: www.ti.com/wirelessconnectivity
- Automotive and Transportation: www.ti.com/automotive
- Communications and Telecom: www.ti.com/communications
- Computers and Peripherals: www.ti.com/computers
- Consumer Electronics: www.ti.com/consumer-apps
- Energy and Lighting: www.ti.com/energy
- Industrial: www.ti.com/industrial
- Medical: www.ti.com/medical
- Security: www.ti.com/security
- Space, Avionics and Defense: www.ti.com/space-avionics-defense
- Video and Imaging: www.ti.com/video
- TI E2E Community: e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2015, Texas Instruments Incorporated