Texas Instruments

TMS320C80 (MVP) Parallel Processor

User's Guide

1995 Digital Signal Processing Products

Printed in U.S.A., March 1995

SPRU110A

User's Guide

TMS320C80 (MVP) Parallel Processor

1995

TMS320C80 (MVP) Parallel Processor User's Guide

SPRU110A
March 1995

PRINTED WITH SOY INK

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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 of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Copyright © 1995, Texas Instruments Incorporated

Preface

Read This First

About This Manual

The TMS320C80 MVP (multimedia video processor) is Texas Instruments first single-chip multiprocessor DSP (digital signal processor) device. The MVP contains five powerful, fully programmable processors: a master processor (MP) and four parallel processors (PPs). The MP is a 32-bit RISC (reduced instruction set computer) with an integral, high-performance IEEE-754 floating-point unit. Each PP is an advanced 32-bit DSP; thus, in addition to having similar processing capabilities as conventional DSPs, each PP has advanced features to accelerate operation on a variety of data types.

The MVP supports a variety of parallel-processing configurations, which facilitates a wide range of multimedia and other applications that require high processing speeds. Applications include image processing, two- and three-dimensional and virtual reality graphics, audio/video digital compression, and telecommunications.

This manual describes the MVP parallel processor (PP). The PP is a programmable advanced DSP that performs digital signal processing and bit-field and multiple-pixel manipulation. It supports the filtering and frequency domain operations required for image-processing applications. The PP can perform the equivalent of ten RISC-equivalent operations in a single clock cycle, thus achieving high speeds of operation. This manual provides information about the PP features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

Notational Conventions

This document uses the following conventions.

Term/Convention	Description
dst	Destination data register
LSB, MSB	Least significant bit, most significant bit
LSbyte, MSbyte	Least significant byte, most significant byte
LShalfword, MShalfword	Least significant halfword, most significant halfword
LSword, MSword	Least significant word, most significant word
|, OR	Bitwise inclusive OR
register⟦bit⟧	Identifies a bit within a register. For example, sr⟦N⟧ refers to the N bit in the sr register.
src1	Data from source register 1
src2	Data from source register 2
u	32-bit unsigned integer precision
XNOR	Bitwise exclusive NOR
XOR	Bitwise exclusive OR
0x	Identifies a hexadecimal value. For example, 0x1234 5678=1234 5678_{16}.
:	Signals end of a label
;	Marks beginning of a single-line comment
&, AND	Bitwise AND
~, NOT	Unary ones complement NOT
/ /	Rotate right
\ \	Rotate left
< <	Shift left
> >	Shift right
$	Current instruction location (used by assembler)
[ ]	Identify an optional parameter extension. Here's an example of an instruction with an optional extension: d2 = [u]b * a1 You can use either the instruction **d2 = ub *a1** or **d2 = b * a1**

Term/Convention	Description
⟦ ⟧	In instructions, represent square brackets ([ ]) that you must use with you type the instruction: dst = [fmod] [⟦cond [.pro] ⟧] ALU_EXPRESSION In this instruction, use must enter square brackets around the **cond** portion of the instruction. For example, use dst = fmod [cond.pro] ALU_EXPRESSION
boldface text	Serves two purposes. In text, **boldface** identifies a key term that is being defined and emphasizes important explanations. In instruction syntax, **boldface** identifies the part of the instruction that you must enter as shown. For example, enter **nop** exactly as shown.
`special font`	Identifies program listings, code examples, filenames, and symbol names.
`||`	Indicates that the following operation is to be performed in parallel and in the same instruction as the previous operation(s).

Information About Cautions

> **This is an example of a caution statement.**
>
> **A caution statement describes a situation that could potentially damage your software or equipment.**

Please read each caution statement carefully.

Related Documentation From Texas Instruments

The following books describe the TMS320C80 MVP and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C80 Multimedia Video Processor Data Sheet (literature number SPRS023) describes the features of the ’C80 device and provides pinouts, electrical specifications, and timings for the device.

TMS320C80 Multimedia Video Processor (MVP) Technical Brief (literature number SPRU106) provides an overview of the ’C80 features, development environment, architecture, and memory organization.

TMS320C80 (MVP) C Source Debugger User’s Guide (literature number SPRU107) describes the ’C80 master processor and parallel processor C source debuggers. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

TMS320C80 (MVP) Code Generation Tools User’s Guide (literature number SPRU108) describes the ’C80 code generation tools. This manual provides information about the features and operation of the linker and the master processor (MP) and parallel processor (PP) C compilers and assemblers. It also includes a description of the common object file format (COFF) and shows you how to link MP and PP code.

TMS320C80 (MVP) Master Processor User's Guide (literature number SPRU109) describes the 'C80 master processor (MP). This manual provides information about the MP features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

TMS320C80 (MVP) Multitasking Executive User's Guide (literature number SPRU112) describes the 'C80 multitasking executive software. This manual provides information about the multitasking executive software features, operation, and interprocessor communications; it also includes a list of task error codes.

TMS320C80 (MVP) System-Level Synopsis (literature number SPRU113) contains the 'C80 system-level synopsis, which describes the 'C80 features, development environment, architecture, memory organization, and communication network (the crossbar).

TMS320C80 (MVP) Transfer Controller User's Guide (literature number SPRU105) describes the 'C80 transfer controller (TC).This manual provides information about the TC features, functional blocks, and operation; it also includes examples of block write operations for big- and little-endian modes.

TMS320C80 (MVP) Video Controller User's Guide (literature number SPRU111) describes the 'C80 video controller (VC). This manual provides information about the VC features, architecture, and operation; it also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

If You Need Assistance. . .

If you want to. . .	**Do this. . .**
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320** **FAX: (713) 274–2324**
Report mistakes in this document or any other TI documentation	Fill out and return the reader response card at the end of this book, or send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443 Electronic mail: **comments@books.sc.ti.com**

Contents

1 Overview of the Parallel Processor PP:1-1

Provides an overview of the MVP parallel processor, including key features, typical applications, and a description of the parallel processor's architecture.

1.1 Description of the Parallel Processor PP:1-2
1.1.1 Key Features PP:1-2
1.2 Typical Applications PP:1-4
1.3 An Overview of the PP Architecture PP:1-5
1.3.1 The PP Data Unit PP:1-7
1.3.2 Global and Local Address Units PP:1-10
1.3.3 Bus Structure Overview PP:1-12
1.3.4 The Program Flow Control Unit PP:1-13
1.3.5 The PP Registers PP:1-15
1.3.6 The Instruction Set PP:1-18

2 Understanding the PP Memory Organization PP:2-1

Discusses the PP memory map, the crossbar, and the PP instruction caches. This chapter also describes the differences between big-endian and little-endian addressing.

2.1 Understanding the PP Memory Map PP:2-2
2.1.1 On-Chip Memory PP:2-3
2.1.2 Off-Chip Memory PP:2-4
2.2 Understanding the Crossbar PP:2-7
2.3 Big-Endian and Little-Endian Addressing PP:2-9
2.3.1 Byte Ordering PP:2-10

3 PP Instruction-Cache Operation and Interprocessor Communications PP:3-1

Describes the PP's instruction cache and provides an overview of inter-processor-communication commands.

3.1 PP Instruction Cache PP:3-2
3.1.1 Cache Architecture PP:3-2
3.1.2 Cache Operation PP:3-4
3.1.3 Cache Replacement Algorithm PP:3-6
3.1.4 Cache Reset PP:3-6

3.1.5 Code Alignment PP:3-7
3.1.6 Self-Modifying Code PP:3-7
3.2 Interprocessor Commands PP:3-8
3.2.1 Issuing an Interprocessor Command PP:3-9
3.2.2 Using Command Words PP:3-11

4 Understanding the Data Unit PP:4-1

The data unit supports the massive processing associated with algorithms such as frequency domain transforms, correlation, and filters. It also supports the bit-field and pixel manipulations required by image coding and computer graphics. This chapter describes the architecture of the data unit.

4.1 Understanding the PP Data Unit PP:4-2
4.1.1 The Data Unit Registers PP:4-3
4.1.2 Inputs and Outputs PP:4-4
4.2 Understanding the ALU Data Path Hardware PP:4-6
4.2.1 Using the Barrel Rotator PP:4-7
4.2.2 Using the Mask Generator Function PP:4-8
4.2.3 Using the Expand Operator and the mf Register PP:4-9
4.2.4 Using Bit-Detection Logic PP:4-10
4.3 Using the Three-Input ALU PP:4-11
4.3.1 Understanding the Base Set and Extended Set of Operations PP:4-12
4.3.2 Setting the ALU Status Register PP:4-13
4.3.3 Using Natural Arithmetic Functions PP:4-14
4.3.4 Using Three-Input Arithmetic Functions PP:4-18
4.3.5 Using Sign-Extend and Absolute Value Functions PP:4-27
4.3.6 Using Multiple Arithmetic Functions PP:4-28
4.4 Using Data Path Multiplexers PP:4-31
4.5 Understanding ALU Data Path Classes PP:4-32
4.6 Combining Two or More ALU Data Path Functions PP:4-38
4.6.1 Using Shifts With Addition and Subtraction PP:4-39
4.6.2 Absolute Value: |value| PP:4-52
4.6.3 Using Field Extraction to Right-Justify Fields PP:4-53
4.6.4 Color Expansion PP:4-54
4.6.5 Using the mf Rotator and Split ALU to Match Pixel Values PP:4-56
4.6.6 Threshold Detection Using Multiple Arithmetic PP:4-57
4.6.7 Using Multiple Arithmetic and the Expander PP:4-59
4.6.8 Adding Two Masked Inputs PP:4-64
4.6.9 Using Multiple Arithmetic With Conditional Function Control PP:4-65
4.7 Using the Multiplier Data Path PP:4-67
4.7.1 Performing 16x16 Multiply Operations PP:4-68
4.7.2 Rounding and Scaling 16x16 Multiply Operations PP:4-69
4.7.3 Performing Split Multiply PP:4-71

5 Understanding the Program Flow Control Unit PP:5-1

The program flow control unit performs instruction fetching and decoding, handles any necessary handshaking with the transfer controller, and facilitates interrupt response and prioritization. This chapter presents an overview of the hardware associated with the program flow control unit.

5.1 Overview of the Program Flow Control Unit PP:5-2
5.2 FAE Instruction Pipeline PP:5-3
5.2.1 Program Counter-Related Registers PP:5-4
5.2.2 Pipeline Implications PP:5-6
5.2.3 Pipeline Stall Conditions PP:5-10
5.3 Loop Controllers PP:5-12
5.3.1 Hardware-Looping Mechanism PP:5-12
5.3.2 Multiple Nested Loops PP:5-13
5.3.3 Hardware Branching PP:5-13
5.3.4 Software Branching Within a Loop PP:5-14
5.4 Cache Controller PP:5-15

6 Understanding the Address Units PP:6-1

The global and local address units compute addresses for memory accesses and control all register movement on the global source, global destination, and local destination/source buses for the PP. This chapter describes the address unit hardware, the differences between the two address units, and data flow and control by the address units.

6.1 Overview of the Address Units PP:6-2
6.1.1 Registers PP:6-3
6.1.2 Index Multiplexer PP:6-3
6.1.3 Index Scaler PP:6-4
6.1.4 Adder/Subtracter Unit PP:6-4
6.1.5 Preindex/Postindex Multiplexer PP:6-4
6.1.6 PP-Relative Multiplexer PP:6-4
6.1.7 Right-Align/Sign Extend Hardware PP:6-5
6.1.8 Associated Bus Structure PP:6-5
6.2 Bus Structure PP:6-6
6.2.1 Hardware for Operations on Data Less Than 32 Bits PP:6-8
6.2.2 Buses Associated With D Registers PP:6-9
6.2.3 Buses Associated With All Registers PP:6-10
6.3 Differences Between the Global and Local Address Units PP:6-11
6.3.1 Address and Index Registers PP:6-11
6.3.2 Accessing Memory in the Same Instruction PP:6-11
6.3.3 Performing Parallel Operations PP:6-12
6.4 Data Flow and Control PP:6-13
6.4.1 Memory Accesses (Loads and Stores) PP:6-13
6.4.2 Register-to-Register Moves PP:6-20
6.4.3 Address Unit Arithmetic PP:6-21

7 Summary of PP Registers PP:7-1

Provides a detailed discussion of the PP registers, including D and non-D registers, and upper and lower registers. This chapter also provides a table showing the different register codes.

7.1 The PP Register Codes PP:7-2
7.2 Alphabetical Summary of the PP Registers PP:7-4

8 The PP Assembly Language Instruction Set PP:8-1

Describes the PP assembly language instruction set.

8.1 Overview of the Assembly Language Instruction Set PP:8-2
8.1.1 Data Unit Operations PP:8-3
8.1.2 Parallel Transfers PP:8-3
8.2 PP Data Unit Operators PP:8-12
8.2.1 Operator Precedence PP:8-13
8.2.2 The Assignment Operators PP:8-15
8.2.3 Conditional Source Selection Operator PP:8-17
8.2.4 The Arithmetic Operators PP:8-18
8.2.5 The Multiplication Operators PP:8-19
8.2.6 The Bitwise Boolean Operators PP:8-20
8.2.7 The Expand Operator PP:8-22
8.2.8 The Mask Generator Operator PP:8-24
8.2.9 The Rotate Operator PP:8-26
8.2.10 The Shift Operators PP:8-27
8.3 The Base Set of ALU Operations PP:8-30
8.3.1 ALU Operands PP:8-32
8.3.2 Boolean and Arithmetic Functions PP:8-38
8.3.3 Operation Classes PP:8-43
8.3.4 ALU Function Modifiers PP:8-49
8.3.5 Summary of ALU Function Modifier Codes PP:8-60
8.4 The Extended Set of ALU Operations (EALU) PP:8-61
8.4.1 Using the EALU Data Path PP:8-63
8.4.2 Understanding the EALU Syntax PP:8-65
8.4.3 Using the Generic EALU Syntax PP:8-66
8.4.4 Using the Explicit EALU Syntax PP:8-69
8.4.5 Using Explicit EALU Expressions PP:8-74
8.4.6 Using EALU Operations Without a Parallel Multiply PP:8-83
8.5 Multiply Operations PP:8-87
8.5.1 Using the MPY||ADD and MPY||SADD Operations PP:8-88
8.5.2 Multiplying in Parallel With an EALU Operation PP:8-90

8.6 The Divide Operation PP:8-96
8.6.1 Mathematical Explanation of the divi Process PP:8-97
8.6.2 Using the divi Operation PP:8-98
8.6.3 Examples of Single divi Instructions PP:8-104
8.6.4 Examples of divi Subroutines PP:8-105
8.6.5 Signed Division PP:8-107
8.6.6 Divide Overflow PP:8-109
8.6.7 Placement of Data for Divide Iterate PP:8-110
8.6.8 Number of Iterations for Divide Iterate PP:8-112
8.7 Miscellaneous Operations PP:8-113
8.7.1 dint PP:8-113
8.7.2 eint PP:8-114
8.7.3 nop PP:8-115
8.8 Parallel Transfers PP:8-116
8.8.1 Addressing Modes PP:8-117
8.8.2 Memory-to-Register Transfers (Loads) PP:8-126
8.8.3 Register-to-Memory Transfers (Stores) PP:8-130
8.8.4 Register-to-Register Transfers (Moves) PP:8-134
8.8.5 Address Unit Arithmetic Operations PP:8-139
8.8.6 Stack Operations PP:8-142
8.9 Conditional Operations PP:8-144
8.9.1 Condition Codes PP:8-145
8.9.2 Conditional Assignment PP:8-146
8.9.3 Conditional Source Selection PP:8-148
8.9.4 Status Bit Protection PP:8-150
8.10 Parallel-Operation Combinations PP:8-156
8.10.1 Understanding the Basic Rules of Parallel Operations PP:8-157
8.10.2 Restrictions With Data Unit Operations PP:8-158
8.10.3 Restrictions With Global Transfers PP:8-159
8.10.4 Restrictions With Local Transfers PP:8-160
8.10.5 Restrictions With Two Parallel Transfers PP:8-161
8.10.6 Summary of Parallel Operations PP:8-161
8.10.7 Programming Exercises PP:8-167
8.11 Parallel Write Priority PP:8-171
8.11.1 Discarding Results Using Fixed Write Priority PP:8-172
8.11.2 Using Conditionals and Fixed Write Priority (If-Else) . . PP:8-173
8.12 Programming Restrictions PP:8-175

9 Interrupts and Reset PP:9-1
Explains the PP interrupt sources and the state of the PP at reset.

9.1 Interrupt-Related Registers PP:9-2
9.1.1 The pc Register PP:9-3
9.1.2 The inten Register PP:9-4
9.1.3 The intflg Register PP:9-6
9.2 Enabling and Disabling Interrupts PP:9-8
9.2.1 Pipeline Considerations PP:9-9
9.3 Interrupt Flags PP:9-10
9.3.1 Clearing Interrupt Flags PP:9-10
9.3.2 Saving and Restoring an Interrupt State PP:9-11
9.4 Interrupt Priorities and Vector Addresses PP:9-13
9.5 Interrupt Processing PP:9-15
9.5.1 Interrupt Service Routine Guidelines PP:9-16
9.5.2 Returning From an Interrupt Service Routine PP:9-17
9.6 Reset PP:9-18
9.6.1 Activity During Reset PP:9-19
9.6.2 Initial State Following Reset PP:9-20
9.6.3 Activity Following Reset PP:9-21

10 Understanding Branches and Calls PP:10-1
Discusses branches and calls and how they differ. Also described are different types of branches and calls.

10.1 Software Branching PP:10-2
10.2 Understanding the Difference Between Branches . . . . . and Calls PP:10-3
10.3 Branch Delay Slots PP:10-4
10.4 Subroutine Call and Return Sequence PP:10-8
10.4.1 Nested Calls PP:10-10
10.5 Absolute Versus Relative Branching PP:10-11
10.5.1 Absolute Branch PP:10-11
10.5.2 Relative Branches PP:10-12
10.6 Conditional Branches and Calls PP:10-13
10.7 Branch Tables PP:10-15
10.8 Two-Input ALU Operations With pc Destination PP:10-17

11 Understanding the Loop Control Logic PP:11-1
The program flow control unit includes three loop controllers that support up to three simultaneous hardware-controlled loops. This chapter describes the basic loop control and tells you how to set up loops.

11.1 Looping PP:11-2
11.2 Basic Loop Control PP:11-3
11.2.1 Loop Control Registers PP:11-3
11.2.2 Loop End Comparison PP:11-5
11.2.3 Loop Count Decrement/Reload PP:11-5

11.3 Setting Up the Loop Control Registers PP:11-6
11.3.1 General Loop Control Initialization PP:11-6
11.3.2 Shortform Loop Control Initialization PP:11-10
11.4 Nested Loops PP:11-14
11.4.1 Loop End Priority PP:11-16
11.5 Uses of Loop Control Flexibility PP:11-19
11.5.1 Multiple Ends Associated With the Same Loop Counter PP:11-19
11.5.2 Hardware Branching PP:11-21
11.5.3 Conditional Software Branching PP:11-24
11.5.4 Conditional Hardware Branching PP:11-26

12 Packet Transfers PP:12-1

Describes the format of packet transfers, their linked-list addressing, the polling and submitting of packet transfer requests to the TC, and the determination of when the data transfer has completed. The chapter concludes with a description and examples of dimensioned and guided transfers.

12.1 Packet Transfer Request Protocol PP:12-2
12.2 Setting Up and Requesting a Packet Transfer PP:12-3
12.2.1 Step 1: Setting Up the Packet Transfer Parameters PP:12-3
12.2.2 Step 2: Setting the Linked-List Pointer PP:12-4
12.2.3 Step 3: Issuing a Packet Transfer Request to the TC . . . PP:12-5
12.2.4 Packet Transfer Handshake Signals PP:12-6
12.3 Waiting for Packet Transfer Request Completion PP:12-9
12.3.1 Polling PP:12-10
12.3.2 Interrupts PP:12-12
12.4 Dimensioned Transfers PP:12-16
12.4.1 Dimensioned-to-Dimensioned Transfer Example PP:12-17
12.4.2 Using the Double-Buffer Transfer Model PP:12-23
12.4.3 Row-Based Processing: Robert's Edge Detection Example PP:12-24
12.5 Guided Transfers PP:12-27
12.5.1 Line Draw Using Fill-With-Value PP:12-28
12.5.2 Line Draw With Pixel Processing PP:12-31

13 Advanced EALU PP:13-1

Describes the bit fields in d0 that define the EALU function code, the bits that determine the carry-in behavior and sign extension, and a bit that controls the input to the C port of the ALU.

13.1 The EALU Function Code PP:13-2
13.1.1 ALU Boolean Operations PP:13-4
13.1.2 EALU Arithmetic Operations PP:13-6

13.2 EALU Carry-In and Sign Extension PP:13-7
13.2.1 Carry-In Starting Value (C Bit) PP:13-8
13.2.2 Invert Carry-In (I Bit) PP:13-9
13.2.3 Sign-Extend Selection (S Bit) PP:13-10
13.2.4 Explicit-Multiple Carry-In (E Bit) PP:13-11
13.2.5 Expanded Multiple Flag Option (F Bit) PP:13-12
13.2.6 Summary of Carry-In Determination PP:13-12

14 Software Optimizations PP:14-1

Describes programming techniques and hardware features that you can use to optimize software for the PP.

14.1 Rolling Code Into a Pipelined Tight Loop PP:14-2
14.1.1 Individual LUT Access PP:14-3
14.1.2 Pipelined LUT Accesses PP:14-5
14.2 MSE Optimization Example PP:14-6
14.2.1 Straight-Line MSE Code PP:14-7
14.2.2 Pipelined MSE Tight Loop (No Multiple Arithmetic) PP:14-8
14.2.3 Straight-Line MSE Code Using Address Unit Arithmetic PP:14-9
14.2.4 Pipelined MSE Tight Loop Combining Two Computation Methods PP:14-10
14.2.5 Straight-Line Code Using Multiple Arithmetic PP:14-11
14.2.6 Pipelined Tight Loop Using Multiple Arithmetic and Address Unit Arithmetic PP:14-13
14.3 Programming Hints PP:14-17

15 Software Applications PP:15-1

Provides application examples that illustrate the flexibility, power, and features of the PP instruction set.

15.1 Graphics Applications PP:15-2
15.1.1 Color Expand: Bit-to-Byte/Halfword/Word Translations PP:15-3
15.1.2 Bresenham Line Drawing PP:15-6
15.1.3 Pixel Block Transfers (PIXBLTs) PP:15-7
15.2 Image-Processing Applications PP:15-15
15.2.1 Thresholding: Word/Halfword/Byte-to-Bit Translations PP:15-16
15.2.2 Binary Morphology PP:15-18
15.2.3 MAD—Mean Absolute Difference (Motion Estimation) PP:15-24
15.2.4 Discrete Cosine Transform (DCT) PP:15-28
15.3 Bitstream Decoding Applications PP:15-32
15.3.1 Tracking the Bitstream Pointer PP:15-33
15.3.2 Iterative Look-Up Tables PP:15-35
15.3.3 Using Bit-Detection Logic PP:15-38

A The PP Opcodes PP:A-1
Describes the PP opcode in detail.

A.1 PP Opcode Summary PP:A-2
A.2 Alphabetical Summary of Opcode Mnemonics PP:A-4
A.3 Data Unit Opcode Formats PP:A-8
A.3.1 Data Unit Opcode Format A: Six-Operand PP:A-8
A.3.2 Data Unit Opcode Format B: Base Set ALU With 5-Bit Immediate src2 . . . . . PP:A-12
A.3.3 Data Unit Opcode Format C: Base Set ALU With Register src2 . . . . . PP:A-18
A.3.4 Data Unit Opcode Format D: Base Set ALU With 32-Bit Immediate PP:A-19
A.3.5 Data Unit Opcode Format E: Miscellaneous Operations PP:A-20
A.4 Parallel Transfer Opcode Formats PP:A-21
A.4.1 Parallel Transfer Opcode Format 1: Double Parallel PP:A-22
A.4.2 Parallel Transfer Opcode Format 2: Move||Local PP:A-25
A.4.3 Parallel Transfer Opcode Format 3: Field Move||Local PP:A-27
A.4.4 Parallel Transfer Opcode Format 4: Local (Long Offset) PP:A-29
A.4.5 Parallel Transfer Opcode Format 5: Global (Long Offset) PP:A-30
A.4.6 Parallel Transfer Opcode Format 6: Non-D DU||Local PP:A-31
A.4.7 Parallel Transfer Opcode Format 7: Conditional Move PP:A-32
A.4.8 Parallel Transfer Opcode Format 8: Conditional Field Move PP:A-34
A.4.9 Parallel Transfer Opcode Format 9: Conditional DU||Conditional Global PP:A-35
A.4.10 Parallel Transfer Opcode Format 10: Conditional Non-D DU PP:A-36

B packetpp.i Include File PP:B-1
Contains the packetpp.i include file. The file has two big-endian structure definitions for setting up packet transfer parameters.

C Understanding the Binary Number System PP:C-1
Offers a brief overview of the binary number system and describes how binary numbers behave under certain conditions.

C.1 Carry-Outs From Addition and Subtraction PP:C-2
C.2 Signed and Unsigned Numbers PP:C-4
C.3 Unsigned Number Overflow PP:C-5
C.3.1 Unsigned Addition Overflow PP:C-5
C.3.2 Unsigned Subtraction Overflow PP:C-6

C.4 Signed Number Overflow PP:C-7
C.5 Extended-Precision Support for Signed Operations With Multiple Arithmetic PP:C-8
C.5.1 Examples of the Math Involved PP:C-8
C.5.2 Generating the 17th Bit of Precision PP:C-10

D Quick Reference PP:D-1

Once you have familiarized yourself with the MVP parallel processor and its instruction set, you can use this appendix as a reference guide.

D.1 The Opcode Formats and Parallel Transfer Formats PP:D-2
D.2 The Data Unit Operations PP:D-5
D.2.1 Data Unit Operators and Operation Syntax PP:D-5
D.2.2 The Base Set of ALU Operations PP:D-10
D.3 The EALU Operations PP:D-14
D.3.1 The Supported EALU Functions PP:D-14
D.3.2 The Boolean Operations PP:D-15
D.3.3 The Arithmetic Operations PP:D-16
D.3.4 Special Operations Supported by the EALU PP:D-17
D.4 The Parallel Transfers PP:D-18
D.5 The Interprocessor Commands PP:D-23
D.6 The Condition Codes PP:D-24
D.7 The Registers PP:D-25
D.7.1 The comm Register PP:D-26
D.7.2 The d0 Register PP:D-26
D.7.3 The inten and intflg Registers PP:D-27
D.7.4 The lctl Register PP:D-28
D.7.5 The sr register PP:D-28
D.8 The Memory Maps PP:D-31

E Glossary PP:E-1

Defines acronyms and key terms used in this book.

Figures

1–1 The PP Block Diagram PP:1-6
1–2 Accessible PP Registers PP:1-17
1–3 General Opcode Format PP:1-19
1–4 32-Bit Immediate Format PP:1-19
1–5 Non-D ALU Operand Format PP:1-19
1–6 Conditional Data Unit and/or Global Transfer Format PP:1-20
1–7 Conditional Data Unit With Non-D Operand Format PP:1-20
1–8 Long Offset Formats PP:1-20
2–1 MVP Memory Map PP:2-5
2–2 PP Parameter RAM Contents PP:2-6
2–3 MVP Crossbar PP:2-8
2–4 Byte Ordering—Register View PP:2-10
3–1 PP Instruction Cache PP:3-3
3–2 pc Register Bit Usage for Cache Operations PP:3-4
3–3 Command Word Bit Assignment PP:3-11
4–1 Data Unit Block Diagram PP:4-2
4–2 Multiple-Byte Arithmetic Data Flow PP:4-29
4–3 Base Set ALU Operation Classes PP:4-34
4–4 Six-Operand Data Path Classes PP:4-35
4–5 16 x 16 Multiplier Data Formats PP:4-68
4–6 Rounded Result Format PP:4-69
4–7 Signed Split Multiply Data Formats PP:4-71
4–8 Unsigned Split Multiply Data Formats PP:4-72
5–1 Program Flow Control Unit Block Diagram PP:5-2
5–2 FAE (Instruction) Pipeline PP:5-4
5–3 Program Counter History PP:5-5
6–1 Address Units Block Diagram PP:6-2
8–1 Base Set ALU Opcode Formats PP:8-30
8–2 ALU D Register Operands PP:8-33
8–3 Non-D Register ALU Operand Opcode Fields PP:8-34
8–4 Lower Register Source Operand for Base Set ALU With 32-Bit Immediate PP:8-35
8–5 Companion Registers PP:8-36
8–6 ALU Function Code PP:8-38

8–7 Operation Class Field PP:8-43
8–8 Base Set Arithmetic Function Modifier PP:8-49
8–9 EALU Data Path Options PP:8-63
8–10 Karnaugh Map PP:8-75
8–11 ALU Function Signal and Karnaugh Map PP:8-76
8–12 EALU||ROTATE Data Routings PP:8-83
8–13 Arbitrary Field Extract PP:8-84
8–14 d0 Multiply Fields PP:8-90
8–15 MPY||EALU Data Routing PP:8-91
8–16 Step-by-Step Division PP:8-97
8–17 ALU Configuration for divi Iterations PP:8-102
8–18 The divi Encoding PP:8-103
8–19 32-Bit Dividend/32-Bit Divisor = 32-Bit Quotient + 32-Bit Remainder in 32 Iterations PP:8-110
8–20 32-Bit Dividend/16-Bit Divisor = 16-Bit Quotient + 16-Bit Remainder in 16 Iterations PP:8-110
8–21 16-Bit Dividend/16-Bit Divisor = 16-Bit Quotient + 16-Bit Remainder in 16 Iterations PP:8-111
8–22 Alternate Placement 16-Bit Dividend/16-Bit Divisor = 16-Bit Quotient + 16-Bit Remainder in 16 Iterations PP:8-112
9–1 pc Register: Global Interrupt Enable Bit PP:9-3
9–2 inten and intflg Registers PP:9-5
9–3 Interrupt Vector Locations PP:9-13
9–4 Stack Condition after PS1–PS5 . . . . . PP:9-15
9–5 Stack Condition Following Software Reset PP:9-20
10–1 Software Branch Program Flow PP:10-5
10–2 Branch in Delay Slot of a Branch PP:10-7
10–3 Software Call and Return Program Flow PP:10-9
11–1 lctl Register PP:11-9
11–2 Program Flow for Example 11–3 . . . . . PP:11-11
11–3 Three Loops Ending at Same Address PP:11-18
11–4 Two Ends to Same Loop PP:11-20
11–5 Program Flow for PIXBLT Fast Subroutine Call PP:11-23
11–6 Conditional Software Branch in its Own Delay Slot PP:11-25
11–7 Conditional Hardware Branch out of a Single-Instruction Loop PP:11-28
12–1 Packet Request Handshake Signals PP:12-7
12–2 Packet Transfer Request Protocol PP:12-8
12–3 Typical Packet Transfer Request/Processing Loop Flow PP:12-12
12–4 Packet Transfer Request/Processing Flow With Polling PP:12-13
12–5 Transfer From 1-D to 2-D Space PP:12-18
12–6 Dimensioned src and dst Packet Transfer Parameters (Big-Endian Format) PP:12-19
12–7 Input, Compute, Output Pipeline Stages PP:12-24
12–8 Sample Memory Allocation (Double Buffering) PP:12-25
12–9 Fill-With-Value to Fixed-Patched Delta-Guided Packet Transfer . . PP:12-29

12–10 Fixed-Patched Delta-Guided Source to Dimensioned Destination Packet Transfer PP:12-32
13–1 d0 Fields Specifying the EALU Operation PP:13-2
13–2 Truth Derivation of A XOR B Function Coding PP:13-4
13–3 Determining Boolean Operation Performed by Function Code 0x23 . . . . . PP:13-5
14–1 Right-Justified Byte Pair → Halfword Pair Using Split Multiply PP:14-19
15–1 Three-Operand PIXBLTs PP:15-7
15–2 PIXBLT PP:15-7
15–3 Sample Data for a Motion Estimation Inner Loop PP:15-27
15–4 Default Multiply Shift PP:15-29
A–1 Opcode Formats PP:A-3
A–2 Data Unit Opcode Format A: Six-Operand PP:A-8
A–3 Data Unit Opcode Format B: Base Set ALU With 5-Bit Immediate src2 . . . . . PP:A-12
A–4 Data Unit Opcode Format C: Base Set ALU With Register src2 . . . PP:A-18
A–5 Data Unit Format D: Base Set ALU With 32-Bit Immediate PP:A-19
A–6 Data Unit Opcode Format E: Miscellaneous Operations PP:A-20
A–7 Parallel Transfer Opcode Format 1: Double Parallel PP:A-22
A–8 Parallel Transfer Opcode Format 2: Move||Local PP:A-25
A–9 Parallel Transfer Opcode Format 3: Field Move||Local PP:A-27
A–10 Parallel Transfer Opcode Format 4: Local (Long Offset) PP:A-29
A–11 Parallel Transfer Opcode Format 5: Global (Long Offset) PP:A-30
A–12 Parallel Transfer Opcode Format 6: Non-D DU||Local PP:A-31
A–13 Parallel Transfer Opcode Format 7: Conditional Move PP:A-32
A–14 Parallel Transfer Opcode Format 8: Conditional Field Move PP:A-34
A–15 Parallel Transfer Opcode Format 9: Conditional DU||Conditional Global PP:A-35
A–16 Parallel Transfer Format 10: Conditional Non-D DU PP:A-36
D–1 Opcode Formats PP:D-3
D–2 Command Word Bit Assignment PP:D-23
D–3 The comm Register Format PP:D-26
D–4 The d0 Register Format PP:D-26
D–5 The inten Register Format PP:D-27
D–6 The intflg Register Format PP:D-27
D–7 The lctl Register Format PP:D-28
D–8 The sr Register Format PP:D-28
D–9 PP Parameter RAM Contents PP:D-31
D–10 Interrupt Vector Locations PP:D-32

Tables

1–1 PP Register Summary PP:1-16
4–1 Some Useful Three-Input ALU Arithmetic Functions PP:4-19
4–2 The 16 Possible Functions of B and C (Assuming Carry-In = 0) PP:4-21
6–1 Minimum DEA Latency PP:6-18
7–1 The Register Codes PP:7-2
7–2 ALU Result Bits Used to Set mf Bits PP:7-29
7–3 mf Bits Supplied to ALU C Port by mf Expander PP:7-30
7–4 MSS Field Values PP:7-39
7–5 Msize and R-Bit Values and Effects PP:7-40
7–6 Asize Values and Effects PP:7-41
8–1 Summary of Data Unit Operations PP:8-4
8–2 Summary of Parallel Transfers PP:8-8
8–3 Syntax For Various PP Functions PP:8-10
8–4 Discussion of the PP Assembly Language Operators PP:8-12
8–5 PP ALU Operator Precedence PP:8-14
8–6 Output From the Expand Operator (@mf) PP:8-23
8–7 Mask Generator Output PP:8-25
8–8 Discussion of the Base Set of ALU Operations PP:8-31
8–9 Companion D Registers, Lower and Upper Registers PP:8-36
8–10 Some Interesting Three-Input Booleans PP:8-39
8–11 Class-Independent Base Set Arithmetic Functions PP:8-40
8–12 Class-Specific Base Set Arithmetic Functions PP:8-41
8–13 Base Set ALU Operation Class Summary PP:8-43
8–14 A + B for Each Operation Class
With dst = d7, src1 = d3, src2 = d4 . . . . . PP:8-45
8–15 A + C for Each Operation Class
With dst = d7, src1 = d3, src2 = d4 . . . . . PP:8-45
8–16 of Class-Independent Base Set Arithmetic Operations PP:8-47
8–17 of Class–Specific Base Set Arithmetic Operations PP:8-48
8–18 Multiple Arithmetic Modifiers PP:8-51
8–19 The lmo, rmo, lmbc, and rmbc Operation Results PP:8-53
8–20 Modified Base Set ALU Operation Classes for
Bit-Detection Operations PP:8-53
8–21 Some Useful Arithmetic Functions That
Use the A Port = 0 Function Modifier PP:8-57

8–22 ALU Function Modifier Codes PP:8-60
8–23 Discussion of the Extended Set of ALU Operations PP:8-62
8–24 Data Path for EALU Operations PP:8-63
8–25 EALU and EALUf Combinations PP:8-68
8–26 ALU Function Code Signals PP:8-76
8–27 Possible f1(B,C) or f2(B,C) Functions PP:8-77
8–28 Discussion of the Assembly Language Multiply Operations PP:8-87
8–29 Discussion of the Divide Operation PP:8-96
8–30 Discussion of Parallel Transfers PP:8-116
8–31 PP Addressing Modes PP:8-118
8–32 Supported Addressing Shortforms PP:8-119
8–33 Immediate Index Range PP:8-121
8–34 PP-Relative Addressing PP:8-124
8–35 Discussion of PP Assembly Language Conditional Operations . . . PP:8-144
8–36 Condition Codes PP:8-145
8–37 Discussion of PP Assembly Language Parallel Operations PP:8-156
8–38 Summary of Parallel Transfer Formats PP:8-162
8–39 Discussion of the PP Parallel Write Priority PP:8-171
10–1 Supported ALU Operations to the pc PP:10-17
13–1 The ALU’s Boolean Products PP:13-3
13–2 EALU Carry-In and Sign-Extension Specification PP:13-13
15–1 Coded Block Pattern VLC Table PP:15-36
A–1 Data Unit Mnemonics PP:A-5
A–2 Parallel Transfer Mnemonics PP:A-6
A–3 Base Set ALU Operation Class Summary PP:A-13
A–4 Base Set ALU Boolean Function Codes PP:A-14
A–5 Base Set Arithmetics PP:A-15
A–6 Class 0 to 7 Function Modifier Opcodes PP:A-16
A–7 Addressing Mode Codes PP:A-22
A–8 Condition Codes PP:A-32
D–1 Summary of Parallel Transfer Formats PP:D-4
D–2 PP Operator Precedence PP:D-5
D–3 Summary of Data Unit Operations PP:D-6
D–4 Base Set ALU Operation Class Summary PP:D-10
D–5 Combinations of Base Set Boolean and Arithmetic Operations PP:D-11
D–6 ALU Function Modifier Codes PP:D-12
D–7 The lmo, rmo, lmbc, and rmbc Operation Results PP:D-13
D–8 Modified Base Set ALU Operation Classes for Bit-Detection Operations PP:D-13
D–9 Data Path for EALU Operations PP:D-14
D–10 ALU Function Code Signals PP:D-15
D–11 ALU Function Code Signal and Karnaugh Map PP:D-15
D–12 Possible f1(B, C) or f2(B, C) Functions PP:D-16
D–13 PP Addressing Modes PP:D-18

D–14 Summary of Parallel Transfers PP:D-19
D–15 Supported Addressing Shortforms PP:D-21
D–16 Immediate Index Range PP:D-21
D–17 PP-Relative Addressing Modes PP:D-22
D–18 The Condition Codes PP:D-24
D–19 The Register Codes PP:D-25
D–20 MSS Field Values PP:D-29

Examples

2–1 Endian-Dependent Code PP:2-10
3–1 Assembler Command Keyword PP:3-9
3–2 Interprocessor Command With Parallel Operations PP:3-9
3–3 Interprocessor Command Delay Slots (Assuming No Interrupts) . . . PP:3-10
3–4 Set-Up and Submission of Halt Command PP:3-12
4–1 Rotate Left by 4 . . . . . PP:4-7
4–2 Mask Generator Functions PP:4-8
4–3 Expand Functions PP:4-9
4–4 Shift Left by Four Bits PP:4-40
4–5 Shift Left and Add PP:4-42
4–6 Shift Left and Subtract PP:4-43
4–7 Unsigned Shift Right PP:4-45
4–8 Unsigned Shift Right and Add PP:4-47
4–9 Signed Shift Right PP:4-49
4–10 Absolute Value PP:4-52
4–11 Field Extraction PP:4-53
4–12 Color Expansion PP:4-54
4–13 Color Expansion With Transparency PP:4-55
4–14 Using the mf Rotator to Accumulate the Results From Two Instructions PP:4-56
4–15 Threshold Based Conversion From Eight-Bit to One-Bit Space PP:4-58
4–16 Transparency Operation on Four 8-Bit Pixels in Two Cycles PP:4-61
4–17 The MAX Function on Four 8-Bit Numbers PP:4-62
4–18 The Add-With-Saturate Function on Four 8-Bit Numbers PP:4-63
4–19 Totaling Four 8-Bit Numbers in Two Instructions PP:4-64
4–20 Taking Absolute Value of the Difference of Four 8-bit Numbers PP:4-66
4–21 Rounded Multiply PP:4-70
8–1 Conditional Assignment PP:8-15
8–2 Status Protection of the Negative Status Bit PP:8-16
8–3 Conditional Source Selection PP:8-17
8–4 Bitwise AND Operations PP:8-20
8–5 Bitwise OR Operations PP:8-20
8–6 Bitwise XOR Operation PP:8-20
8–7 1s-Complement Operation PP:8-21

8–8 Merge Operation PP:8-21
8–9 Merge Operation Using the Expand Operator (@mf) PP:8-23
8–10 Using a Mask Generator Operator (%)
With a Shift Right Mask Modifier (!) PP:8-24
8–11 Using the Unsigned Shift Right Pseudo-Operator
With a Register Shift Amount PP:8-28
8–12 Using the Unsigned Shift Right Pseudo-Operator
With a Constant Shift Amount PP:8-28
8–13 Determine Size of Magnitude Using lmo PP:8-53
8–14 Split Carry Path for Bit-Detection Operation PP:8-54
8–15 Compare Result of Bit Detection PP:8-56
8–16 64-Bit Add PP:8-58
8–17 64-Bit Subtract PP:8-58
8–18 Explicit EALU Syntax PP:8-69
8–19 Shift and Add Using Explicit EALU Syntax PP:8-73
8–20 Boolean Explicit EALU PP:8-75
8–21 Arithmetic EALU Specified by Explicit Syntax PP:8-78
8–22 Multiply With Parallel Absolute Value PP:8-79
8–23 Multiply With Parallel Shift Left PP:8-80
8–24 Add or Subtract B, As Determined by Input to C PP:8-80
8–25 Set-Up Field Extract EALU PP:8-85
8–26 Unrelated Rotate and ALU Operations PP:8-86
8–27 Shifted Multiply PP:8-92
8–28 Split Multiply PP:8-93
8–29 Rounded Multiply PP:8-94
8–30 Using ealu and ealuf PP:8-95
8–31 First divi Instruction PP:8-104
8–32 Typical divi Instruction PP:8-104
8–33 Typical divi Instruction With Conditional PP:8-104
8–34 Typical divi Instruction With Conditional and Protected Status PP:8-104
8–35 Unsigned Divide Iterate (32 Bits) PP:8-105
8–36 Unsigned Divide Iterate (16 Bits) PP:8-106
8–37 Signed Divide Iterate PP:8-108
8–38 Interrupt Enable Sequence PP:8-114
8–39 Assembler Interpretation of Negative Immediate Index PP:8-122
8–40 Assembler Conversion Operation for Immediate Indices PP:8-123
8–41 PP-Relative Addressing PP:8-125
8–42 Using the Load Operation PP:8-129
8–43 Conditional Store Effect PP:8-131
8–44 Using the Store Operation PP:8-133
8–45 Ambiguous Move Syntax PP:8-135
8–46 Conditional Swap PP:8-135
8–47 Field Extract Move PP:8-137
8–48 Field Replicate Moves PP:8-138

8–49 Load-Immediate Using Address Unit Arithmetic With a15/7 . . . . . PP:8-141
8–50 Using an Address Unit Arithmetic Operation PP:8-141
8–51 Example Push and Pop Instructions PP:8-142
8–52 Parallel Stack Operations PP:8-143
8–53 Sample Conditional Assignment Combinations PP:8-146
8–54 Conditional Swap PP:8-147
8–55 Inner Loop for a Bresenham Line PP:8-148
8–56 Single Instruction Using Conditional Assignment and Source Selection PP:8-149
8–57 Protection of C and V Status Bits PP:8-150
8–58 Status Protection for Some Unconditional ALU Operations PP:8-151
8–59 Same Status Used by Several Instructions PP:8-152
8–60 Using Status Protection for Combinatorial Conditions PP:8-153
8–61 Status Protection for Data Unit Operations to the sr Register PP:8-154
8–62 Status Protection With Parallel Transfers to the sr Register PP:8-155
8–63 Sample Parallel Operations PP:8-164
8–64 Legal Instruction PP:8-165
8–65 Global Transfer PP:8-166
8–66 Compare Result Discard PP:8-172
8–67 Discard Local Transfer Write PP:8-172
8–68 If-Else Statement Using Write Priority PP:8-173
8–69 Run-Length Update Using Write Priority PP:8-174
9–1 Interrupt Enable Sequence PP:9-9
9–2 Interrupt Flag Clearing, W = 0 . . . . . PP:9-10
9–3 Saving and Restoring Interrupt Status PP:9-12
9–4 Setting up an Interrupt Vector PP:9-14
10–1 Branch Delay Slot Instructions PP:10-4
10–2 A Branch in the Second Delay Slot of a Branch PP:10-6
10–3 Call and Return Sequence PP:10-8
10–4 Use of Delay Slot Instructions PP:10-9
10–5 Saving the iprs Value Before a Nested Call PP:10-10
10–6 Absolute Call PP:10-11
10–7 Relative Call PP:10-12
10–8 Conditional Branch PP:10-13
10–9 Conditional Branch PP:10-14
10–10 Branch Table PP:10-16
11–1 Loop Set Up PP:11-7
11–2 Code-Position Independent Loop End Specification PP:11-8
11–3 Single-Instruction Loop Shortform Initialization PP:11-11
11–4 Multi-Instruction Loop Shortform Initialization PP:11-12
11–5 Three Levels of Nested Looping PP:11-15
11–6 Three Levels of Nested Looping With Same Loop End PP:11-17
11–7 Bitstream Decoding With Two Ends to Same Loop PP:11-20
11–8 PIXBLT Example With Fast Subroutine Call PP:11-22

11–9 Character Search Using Conditional Software Branch Exit PP:11-24
11–10 Character Search Using Conditional Hardware Branch PP:11-27
12–1 Issuing a Packet Transfer Request: Setting the P Bit PP:12-5
12–2 Polling for Completed Packet Transfer Request PP:12-10
12–3 Three-Cycle Polling for Completed Packet Transfer Request PP:12-10
12–4 Packet Transfer Queued Interrupt Service Routine PP:12-14
12–5 PP Packet Transfer Parameters Set-Up Code PP:12-21
12–6 PP Packet Transfer Parameters Setup Code (Using Structures) . . PP:12-22
13–1 Basic Equations for the PP ALU PP:13-3
13–2 A+B ALU Terms PP:13-6
13–3 A–B ALU Terms PP:13-6
13–4 Using I and S Bits for A – |B| PP:13-9
13–5 Using S Bit for Signed Right Shift PP:13-10
13–6 Using E Bit for Mixed Multiple Operations PP:13-11
14–1 Straight-Line Coded LUT Access PP:14-4
14–2 Tight Loop for Pipelined LUT Accesses PP:14-5
14–3 MSE Sum: Straight-Line Code PP:14-7
14–4 MSE: Pipelined Tight Loop PP:14-8
14–5 MSE: Straight-Line Code Using Address Unit Arithmetic PP:14-9
14–6 MSE: Pipelined Loop Using Two MSE Sum Methods PP:14-10
14–7 MSE: Multiple Arithmetic Straight-Line Code PP:14-12
14–8 Mean Squared Error PP:14-15
14–9 Code With Two Conditions PP:14-18
15–1 Color Expand Code PP:15-3
15–2 Sample Pass Through Color Expand Loop0 . . . . . PP:15-5
15–3 Bresenham Line Draw Inner Loop PP:15-6
15–4 Setup of Prezero Packet Transfer PP:15-8
15–5 Merge MAX/MIN PP:15-10
15–6 Add-With-Saturate PP:15-11
15–7 Subtract-With-Saturate PP:15-12
15–8 Pixel Processing With Plane Masking PP:15-13
15–9 Byte-Bit Translation PP:15-17
15–10 Setup and Variable Assignments for Erosion Tight Loop PP:15-20
15–11 Erosion Tight Loop PP:15-21
15–12 Motion Estimation Inner Loop Code for Absolute Differences PP:15-26
15–13 Add and Subtract With Same EALU Setup PP:15-30
15–14 MSB Halfword Add, LSB Halfword Subtract PP:15-31
15–15 Merge 32-Bit Bitstream Word PP:15-34
15–16 CBP Decode Routine PP:15-37
15–17 h.261 MTYPE Decode Routine PP:15-39
C–1 Unsigned Subtraction With a Positive Result (Carry-Out = 1) PP:C-6
C–2 Unsigned Subtraction With a Negative Result (Carry-Out = 0) PP:C-6
C–3 Unsigned Subtraction With a Result of Zero (Carry-Out = 1) PP:C-6
C–4 Adding a Positive to a Positive With No Signed Overflow in the 16 LSBs PP:C-8

C–5 Adding Two Large Positive Numbers Returning Signed Overflow in the 16 LSBs PP:C-9
C–6 Adding a Negative to a Negative With No Signed Overflow in the 16 LSBs PP:C-9
C–7 Adding Two Very Negative Numbers Returning Signed Overflow in the 16 LSBs PP:C-9
C–8 Adding a (More) Negative to a Positive With No Signed Overflow in the 16 LSBs PP:C-9
C–9 Adding a (Less) Negative to a Positive With No Signed Overflow in the 16 LSBs PP:C-10
D–1 Using the cmnd Keyword PP:D-23
D–2 A Special Case Move to the a15 Register PP:D-23

Chapter 1

Overview of the Parallel Processor

This chapter provides an overview of the MVP parallel processor and its architecture.

Topics

1.1	Description of the Parallel Processor	PP: 1-2
1.2	Typical Applications	PP: 1-4
1.3	An Overview of the PP Architecture	PP: 1-5

1.1 Description of the Parallel Processor

The MVP parallel processor (PP) is a programmable DSP-like 32-bit integer processor with a 64-bit instruction word that is optimized for imaging and graphics applications. It supports the filtering and frequency domain operations required for image processing. The PP can execute in parallel a multiply, an ALU operation (such as a shift-and-add), and two memory accesses, within one single-cycle instruction.

The PP has a three-input ALU that supports all 256 Boolean combinations of three inputs and many combinations of arithmetic and Boolean functions. Data merging and bit-to-byte, bit-to-halfword, and bit-to-word translations are supported by hardware along the input data path to the ALU. These merging and translation operations allow the PP to accelerate graphics applications such as windowing environments. The internal parallelism allows a single PP to achieve over 500 million operations per second for certain algorithms.

1.1.1 Key Features

- High performance, fully programmable processor
- 64-bit instruction word supports many parallel operations, such as a multiply, an ALU operation, and two memory accesses in a single cycle
- Three-stage pipeline provides fast instruction cycle
- Registers
  - Eight data registers, ten address registers, and six index registers
  - Twenty other user-visible registers
- Data unit highlights
  - 16×16 integer multiplier (optional dual 8×8 multiplies)
  - Splittable three-input ALU
  - 32-bit barrel rotator
  - Mask generator
  - Multiple status flag expander facilitates translations to and from 1-bit-per-pixel space. It also supports transparency, max, min, saturation, z-buffering, and patterning.

- Conditional operations to reduce branch requirements and delays; operations include both conditional assignment of data unit result(s) (16 condition codes) and conditional source selection (based on negative status bit).
- Special processing hardware such as leftmost one and rightmost one detection and leftmost bit change and right most bit change detection.

❑ Memory addressing highlights

- Two address units (called the global unit and the local unit), allowing two 32-bit memory accesses in parallel with data unit operations
- Twelve basic address modes (variations of immediate and indexed addressing)
- 8-, 16-, and 32-bit data (or pixel) sizes for loads and stores
- Loads of 8-bit or 16-bit data are either sign- or zero-extended to 32 bits
- Indexed addressing can be scaled according to data size
- Big- and little-endian addressing supported
- Conditional assignment for loads (memory-to-register transfers) based on 1 of 16 condition codes
- Conditional source selection for stores (register-to-memory transfers) based on negative status bit

❑ Program flow highlights

- Three hardware loop controllers that support zero-overhead looping and/or branching, three levels of nested loops, one loop with multiple end points, and many other flexible looping combinations
- The program counter (pc register) is mapped into the register file. Either the ALU or the global address unit can write to the pc register conditionally or unconditionally to cause a branch or subroutine call.
- Interrupts for message passing and context switching
- Instruction cache for accelerating program execution on the PP

❑ Run-time parallel programming environment support

❑ Algebraic assembly language

1.2 Typical Applications

The PP serves as a high-speed pixel coprocessor for the RISC master processor (MP). Typical tasks performed by the PPs are:

- Pixel-intensive processing
  - Motion estimation
  - Convolution
  - PixBLTs
  - Warp
  - Histogram
  - Mean square error
- Domain transforms
  - DCT
  - FFT
  - Hough
- Core graphics functions
  - Line
  - Circle
  - Shaded fills
  - Fonts
- Image analysis
  - Segmentation
  - Feature extraction
- Bit stream coding/decoding
  - Data merging
  - Table lookups

1.3 An Overview of the PP Architecture

The parallel processor (PP) combines high-speed integer computations and bit-field processing capability in a fully programmable architecture. The PP is capable of performing the equivalent of many RISC-like instructions in a single cycle. Each PP executes its own instruction stream independently from any other on-chip processor in the MVP.

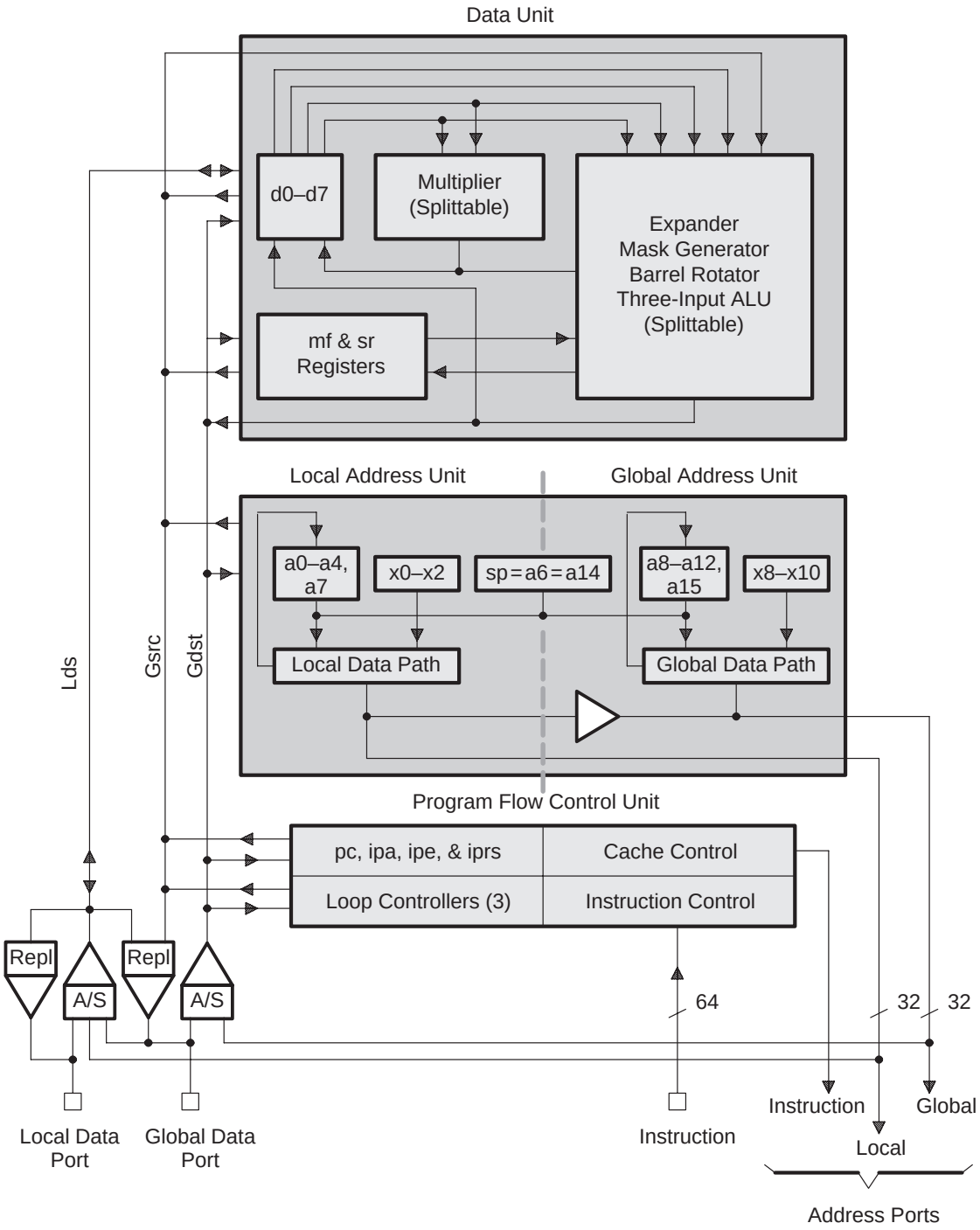
The PP has a long instruction word of 64 bits that makes much of the hardware directly accessible to you and supports many parallel operations. The PP contains a data-processing unit (the data unit) with a single-cycle multiplier, a three-input ALU, and other special hardware for signal, image, bit-field, and pixel processing. These hardware functions operate on registers and immediate data.

All PP memory accesses are performed with load and store operations. Two address units can each perform independent loads (memory-to-register transfers) and stores (register-to-memory transfers) in parallel with the data processing performed by the data unit. These address units can also perform some general-purpose data computations.

Up to three loops (nested or otherwise) can be set up and controlled by hardware so that no loop counting or branching overhead is incurred within loops. The long instruction word, special data-processing hardware, and hardware loop control combine to make the PPs adept at handling the operations associated with image, graphics, and audio processing. The architecture is particularly targeted at coding the inner loops of key algorithms in a small number of instructions.

Figure 1–1 shows a block diagram of the PP's four major functional units: the data unit, local address unit, global address unit, and program flow control unit. Each of these units is capable of several RISC-equivalent operations in a single cycle, as specified in the 64-bit instruction word.

Figure 1–1. The PP Block Diagram



Notes: 1) **Lds** stands for local destination/source bus

2) **Gsrc** stands for global source bus

3) **Gdst** stands for global destination bus

4) **Repl** stands for replicate hardware

5) **A/S** stands for align/sign-extend hardware

1.3.1 The PP Data Unit

The PP's data unit supports the massive processing associated with algorithms such as frequency domain transforms (for example, DCTs), correlation, and filters. It also supports the bit-field and pixel manipulations required by image coding and computer graphics. The combination of special hardware and a flexible data path allows a single-cycle data unit operation to perform the equivalent of many general-purpose processor instructions.

The major elements of the data unit are the:

- D registers (d0–d7)
- Multiple flags register (mf)
- Status register (sr)
- Multiplier data path
- ALU data path

Associated with the ALU data path are a 1-bit to 8-, 16-, or 32-bit expander, a mask generator, a barrel rotator, leftmost and rightmost bit-detection hardware, and a three-input ALU.

Additionally, a series of multiplexers route data through the various hardware functions. In a single cycle, the data unit can perform both multiplier and ALU data path operations.

Here are descriptions of the data unit components:

- **D registers**

  The data unit contains eight general-purpose data registers (d0–d7) that are referred to collectively as the D registers. These are the primary registers used by the data unit. Up to ten different accesses (composed of some combination of up to seven reads and four writes) can be made to the D registers in a single cycle.

  For example, in a single cycle D registers can provide:

  - Two sources and the destination for the multiplier
  - Three sources and the destination for the ALU data path
  - The source and destination for a register-to-register move
  - The source or destination for a memory access

  For EALU operations, d0 acts as the overall control register. The d0 register configures the ALU and multiply data paths for certain operations.

❑ Multiple flags register

Status from split-ALU operations is saved in the multiple flags register. This status information can then be used by the PP's hardware expander.

You can select automatic rotation or clearing of the mf register before performing a split-ALU operation. This facilitates the accumulation of status information from several split-ALU operations.

When using the expander, you can select automatic rotation of the mf register by the number of bits used. This facilitates the use of a full 32-bit word of status bits through repeated use of the expander.

❑ Status register

The status register contains zero-result, carry-out, overflow, and negative-result status bits that are set according to the output of the data unit's ALU. It also contains control bits that indicate how the ALU is split and the data size of the expander.

❑ Multiplier data path

The multiplier can perform either one 16×16 multiply with a 32-bit result or two 8×8 multiplies each with a 16-bit result. The multiplier data path hardware also supports rounding of 16×16 multiplies to a 16-bit result that allows you to perform back-to-back multiplies on packed halfwords. The multiply source and destination operands must be D registers.

❑ Three-input ALU

The three-input ALU can perform 256 Boolean and 256 mixed Boolean and arithmetic functions. The 32-bit ALU can optionally be split into segments to perform two 16-bit or four 8-bit arithmetic operations in parallel. Since byte-sized pixels are common, this can often provide up to a 4-fold speed-up.

The combination of the three-input ALU and the various ALU data path options allows over 2,000 different operations. This flexibility generally results in fewer instructions being required to perform a given algorithm. The ALU's data path allows one source operand to come from any PP register; the remaining register sources are D registers. The destination can be any PP register.

- **Expander**

  The expander can replicate 1, 2, or 4 mf bits 32, 16, or 8 times, respectively, to create a 32-bit word. This 32-bit word can act as a mask that effectively controls the arithmetic operation performed by the different segments of the split ALU on the basis of the status information in the mf register.

  The combination of the three-input split ALU, the mf register, and the expander supports transformations between 1- and n-bit representations of data (for uses such as bit-pixel color expand and z buffering).

- **Mask generator**

  The mask generator is directly accessible to you. The mask generator takes a 5-bit input that specifies the number of right-justified 1s desired and outputs the corresponding number of 1s, zero-extended to 32 bits. In other words, if *n* is the 5-bit input, the 32-bit output word is (2^n-1).

- **Barrel rotator**

  The ALU data path contains a 32-bit barrel rotator that performs a left-rotate by the number of bits specified. The barrel rotator wraps around bits that rotate out of the MSBs to the LSBs. Shifts are performed on the PP by inputting a left-rotated value into the ALU and using a mask based on the 5-bit rotate amount to strip off undesired bits or perform sign extension.

- **Bit-detection logic**

  The bit-detection logic can detect the most significant nonzero bit in a register (**lmo()**), the least significant nonzero bit in a register (**rmo()**), the MSB that is not the same as bit 31 (**lmbc()**), and the LSB that is not the same as bit 0 (**rmbc()**).

For more information about the data unit and its components, see Chapter 4, *Understanding the Data Unit*.

1.3.2 Global and Local Address Units

The PP address units compute addresses for memory accesses and also control all data movement among the various PP functional units. As shown in Figure 1–1, the PP has two address units:

- Global address unit
- Local address unit

The PP's data unit processes only data contained in registers or directly in the instruction word. Data in memory must first be loaded into a register before it can be processed in the data unit. The address units provide the ability to transfer data between registers and memory. The address units specify all memory accesses, using either load (memory-to-register transfer) or store (register-to-memory transfer) operations. A PP instruction can specify up to two independent memory accesses, one by each address unit. The ability to perform two memory accesses per cycle is important because the PP data unit is so powerful.

- **Address unit registers**

  The address units contain both address and index registers. By using one of twelve addressing modes supported by the address unit data path, address registers are combined with either an index register or an immediate offset (from the instruction word) to form an address for a memory access. The PP addressing modes include both preindexed and postindexed addressing, as described in subsection 8.8.1, *Addressing Modes*.

  In addition to the general-purpose address and index registers, each address unit has a zero register. Also, the two address units share a common stack pointer.

- **Loads and stores**

  The PP address units can load or store byte (8-bit), halfword (16-bit), or word (32-bit) data. For byte and halfword loads, the data is right-aligned, then either zero- or sign-extended to fill 32 bits. For byte and halfword stores, the rightmost data bits in a register are stored at the appropriate address.

- **Scaling capability**

  You can scale an index (specified by an index register or an immediate offset) by the data size before combining it with the address register value. To allow a wider access range using an offset, the assembler attempts to scale unscaled immediate offsets to the data size specified.

- **PP-relative addressing**

  Special PP-relative addressing allows the local data RAM0 or the local parameter RAM base address to be used as the base address for address computation. PP-relative addressing facilitates the development of PP-independent code (code that can run unmodified on any of the PPs).

- **Memory-access management**

  While the address units are referred to as local and global, neither address unit is restricted to accessing only local or nonlocal RAMs. However, two memory accesses specified in parallel in a single instruction word can occur in a single cycle only if the local address unit accesses one of the local RAMs.

  If the local address unit tries to access a nonlocal RAM in parallel with a global address unit operation, the instruction pipeline stalls until the global operation is completed; then the local address unit's access can occur.

- **Address unit arithmetic operations**

  The address units can perform general-purpose data computations referred to as address unit arithmetic. Address unit arithmetic operations can offload some of the data processing from the data unit for computation-intensive algorithms.

- **Register-to-register moves**

  The global address unit can specify register-to-register moves. Special field moves can transfer an individual byte or halfword from a source register to the destination register or replicate a byte or halfword to fill 32 bits.

Chapter 6, *Understanding the Address Units*, contains a more detailed description of the address unit hardware. Section 8.8, *Parallel Transfers*, describes the address unit operations supported by the PP assembly language.

1.3.3 Bus Structure Overview

The PP bus structure is shown in Figure 1–1. These buses transfer data between PP registers and memory, transfer data between the PP functional units, provide instruction addresses, and receive instructions.

Each PP has three ports to memory: the instruction port, global port, and local port.

- The instruction port is 64 bits wide and is dedicated to the instructions being fetched by the program flow control unit from the instruction-cache RAM.
- Both the global and local ports are 32 bits wide. They are used for loading and storing PP data.
  - The global port has crossbar connections to all of the shared RAMs on the chip.
  - The local port has crossbar connections to only the four shared RAMs that are local to the given PP.

Although the buses are distributed within the PP, much of the associated hardware is integral to the function of the address units. For more information about the bus structure and for a detailed description of each individual bus, see Section 6.2, *Bus Structure*.

1.3.4 The Program Flow Control Unit

The program flow control unit shown at the bottom of Figure 1–1 controls the PP instruction pipeline, performs instruction fetching and decoding, performs any necessary handshaking with the transfer controller, and handles interrupt response and prioritization.

The major hardware elements of the program flow control unit are the instruction decode control, the program counter related registers (pc, ipa, ipe, iprs), the instruction cache controller, and three hardware loop controllers. These allow the program flow control unit to support conditional branches, interrupts, zero-overhead looping, and sequential program execution.

- **Instruction controller**

  The instruction controller uses the 64-bit instruction that is fetched by the cache controller to generate the control signals that drive the PP. Instructions are processed by a pipeline that consists of three stages: instruction fetch; address unit computations; and data unit, memory access, and/or register-to-register move execution. At any given time, three instructions are in the pipeline (one at each stage) so that the net throughput is one cycle per instruction (assuming there are no stall conditions).

- **Program counter registers**

  The program counter (pc) register points to the next instruction currently being fetched. The hardware associated with the pc increments the pc each time the pipeline advances. Usually, the pc can be accessed like any other PP register. Branches or calls are performed when the data unit or an address unit writes to the pc register in the execute stage of the instruction pipeline.

  While it is physically one register, the pc can be written to using two different register codes. The call register code both selects the pc register for the destination of the write and saves a return address in the instruction pointer return from subroutine (iprs) register.

  The other register code, referred to as br (for branch), selects the pc register for writing but does not save a return address. Since branches and calls differ only in the saving of the iprs and both cause a branch in the program flow, they are both referred to as branches in this document except when there is a need to refer specifically to the call operation.

The PP instruction set supports conditional assignment of the ALU results; and conditional loading of data accessed from memory to a register. When the pc register is specified as the destination, these conditional operations can be used to perform conditional branches in the program flow. Conditional operations can also be used to avoid conditional branches around one or two instructions.

When the pc register is written to by an instruction, the next two instructions, which have already been fetched into the pipeline, are always executed. These two instructions are referred to as delay slot instructions. Generally, the delay slots can be filled with useful operations that are not dependent on program flow or that do no harm if executed when a branch occurs.

The address stage instruction pointer (ipa) and execute stage instruction pointer (ipe) keep track of the program counter as the instruction proceeds through the instruction pipeline. As detailed later, the ipe register can be used for program counter relative calculations (for example, relative branches).

- **Cache controller**

  The cache controller compares the instruction address given by the pc and determines if the instruction is in the PP's cache. If the instruction is already in cache, it translates the address into the location in the PP's cache RAM and issues the translated address over the instruction address port (IAP). If the cache controller determines that the instruction is not in cache, it issues a request to the transfer controller (TC) and stalls the instruction pipeline until the cache has been loaded with the new instruction. Caching is discussed further in Section 3.1, *PP Instruction Cache*.

- **Loop controllers**

  The program flow control unit contains three loop controllers that support up to three simultaneous hardware-controlled loops. Because no hardware execution time is spent on loop counting and branching, the loops are referred to as zero-overhead loops. Furthermore, hardware in the loop controllers operates in the fetch stage so that no delay slot instructions are executed at the bottom of a loop.

For more information about the program flow control unit and its components, see Chapter 5, *Understanding the Program Flow Control Unit*.

1.3.5 The PP Registers

The PP contains many general-purpose registers. Additional registers are used to retain ALU status information and to configure certain features such as hardware loop control. All PP registers are 32 bits wide.

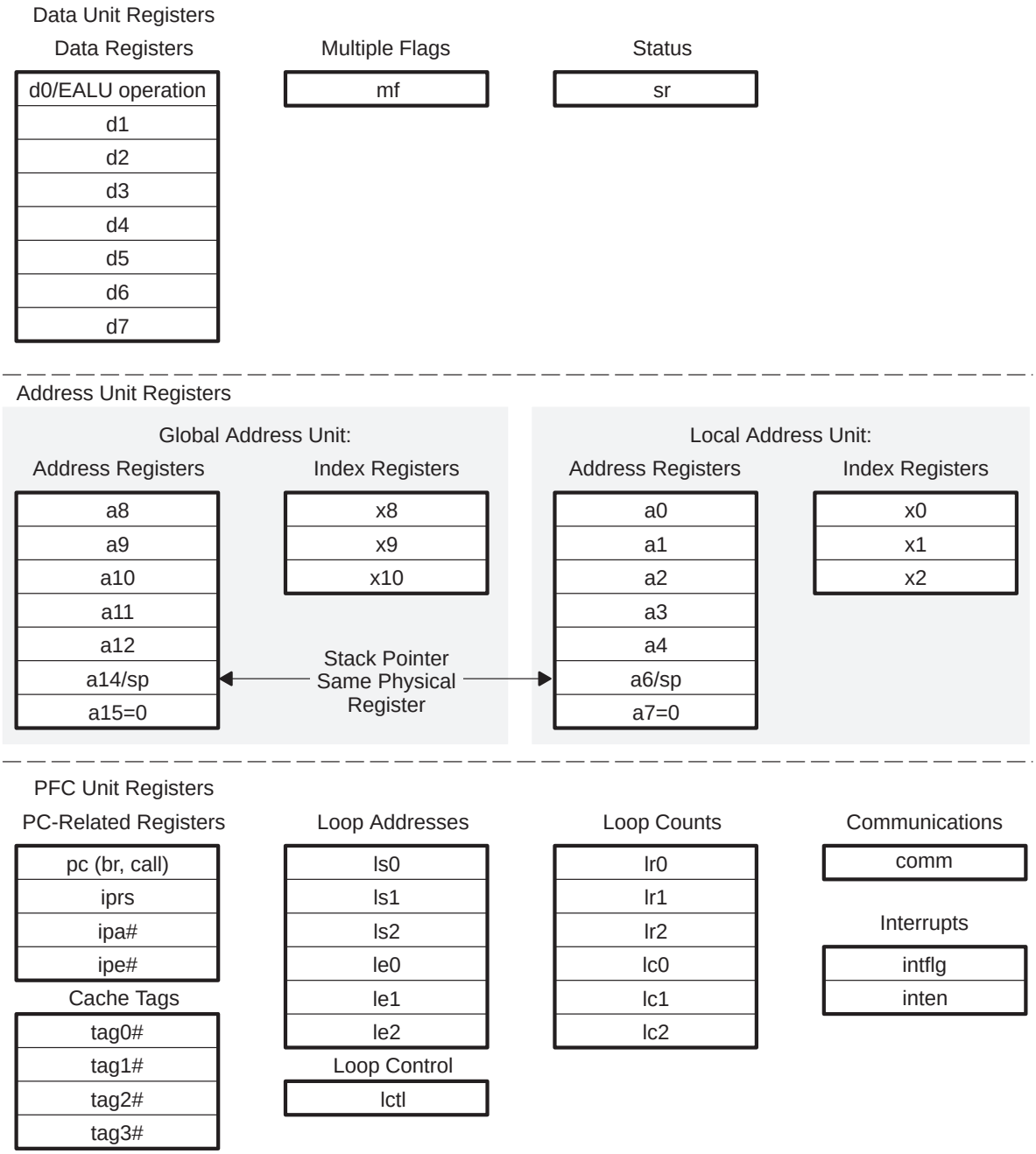
The PP registers are summarized in Table 1–1. Chapter 7, *Summary of PP Registers*, contains a detailed alphabetical description of the PP registers.

As shown in Figure 1–2, each register is associated with one of the functional units. Registers d0–d7 (referred to as **D registers**) are the primary operands for data unit operations. All other registers are referred to collectively as **non-D registers**. Non-D registers can also be operated on by the data unit, but some restrictions apply.

Table 1–1. PP Register Summary

Register Type	Description
Address Registers	Registers used to contain the base address for an address computation. You can also use them for general-purpose data. ❑ Five general-purpose registers used for address computation by the local address unit (a0–a4). ❑ Five general-purpose registers used for address computation by the global address unit (a8–a12). ❑ A system stack pointer shared by both the local and global address units (a6, a14). ❑ Read-as-zero registers (a7, a15).
Comm	Register containing packet transfer handshake bits and the PP identification number.
Data Registers	Eight general-purpose data registers (d0–d7) referred to collectively as **D registers**. These are the primary registers used by the data unit. The d0 register acts as the control register for EALU operations.
Index Registers	Registers used for containing index values which may be added to or subtracted from an address register. You can also use them for general-purpose data. ❑ Three general-purpose registers used for address computation by the local address unit (x0–x2). ❑ Three general-purpose registers used for address computation by the global address unit (x8–x10).
Instruction Pointers	A pointer corresponding to each stage of the PP pipeline: (pc, ipa, ipe) and a return from subroutine call pointer (iprs).
Interrupt	Interrupt flags (intflg) and an interrupt enable register (inten).
Loop Control	PP registers that control three levels of zero-overhead loops including loop counter (lc0, lc1, lc2), loop end (le0, le1, le2), loop start (ls0, ls1, ls2), loop reload (lr0, lr1, lr2), and loop configuration (lctl) registers.
Status	The status register (sr) contains four status bits (negative, carry, overflow, and zero) as well as fields configuring split ALU and mf expander operations. The multiple flags (mf) register is used to record the status of individual results for split ALU operations. You can also use the mf register to provide mask information.
Tag	Cache Tag registers (tag0, tag1, tag2, tag3).

Figure 1–2. Accessible PP Registers



# indicates read-only registers.

1.3.6 The Instruction Set

The PP uses a 64-bit instruction opcode that can specify a number of operations to be performed in parallel by the data unit and address units. The address unit operations are not bound to specific data unit operations. This flexibility greatly enhances the utility of the address units. When programming the PP, it is useful to think of an instruction as consisting of multiple subinstructions (referred to as operations) for the multiplier, ALU data path, global address unit, and local address unit.

- **PP assembly language**

  As described in more detail later, the combination of a three-input ALU and different ALU data path routings into the ALU allows literally thousands of unique operations to be specified. Also, the address units provide 12 addressing modes with several options such as index scaling and PP-relative addressing.

  Rather than naming these thousands of operations, the PP has an algebraic assembler that accepts assignment statements that define the operation. The assembler includes a Boolean expression evaluator so that you do not have to use a specific form for a given operation (the same Boolean operation can be expressed in numerous equivalent ways).

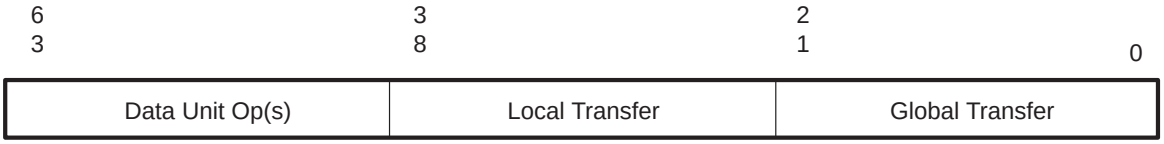
  Chapter 8, *The PP Assembly Language Instruction Set*, describes the PP assembly language in detail.

- **PP opcode**

  While the instruction set is described primarily as an assembly language, a general knowledge of the opcode formats is useful in understanding which operations can be specified in parallel. The figures in this section show a high-level view of the PP opcode formats. Appendix A, *The PP Opcodes*, provides a more detailed description of the PP opcode.

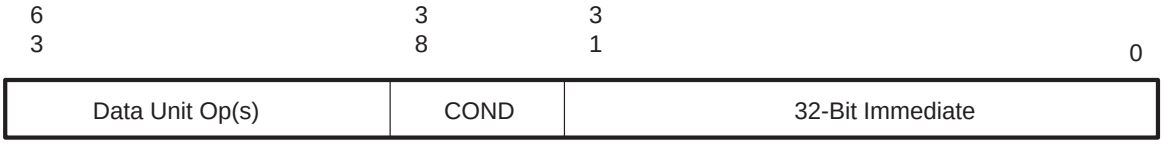
As shown in Figure 1–3, the opcodes for most PP instructions are partitioned into three fields that specify the data unit operation, local address unit operation (referred to as a local transfer), and the global address unit operation (referred to as a global transfer). This allows each unit to specify independent operations. The multiplier and ALU data path operations are both defined by the data unit field.

Figure 1–3. General Opcode Format



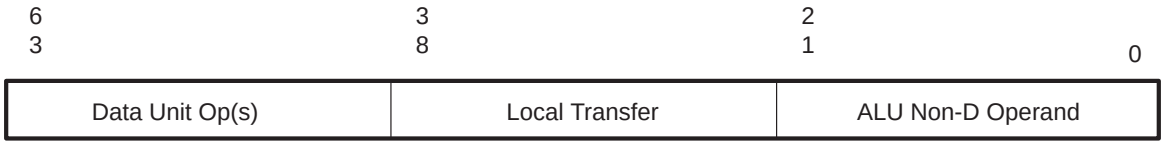
Some operations borrow bits from the opcode fields that are normally used for one or both of the address units and use the bits for other purposes. For example, as shown in Figure 1–4, data unit operations with a 32-bit immediate data source operand require the opcode bits used to specify both the global and local address unit operations. Thus, if a 32-bit immediate is used, no operations can be performed in parallel with the data unit. The data unit operation can optionally be conditional.

Figure 1–4. 32-Bit Immediate Format



Using a non-D register as a source or destination for the data unit's ALU operation requires the opcode bits normally used by the global address unit, as shown in Figure 1–5. Thus, a parallel local transfer can be specified, but no parallel global transfer is allowed.

Figure 1–5. Non-D ALU Operand Format



Conditional operations by the data unit and/or the global address unit require the opcode bits used to specify the local address unit's operation, as shown in Figure 1–6. Thus, conditional operations cannot be specified in parallel with a local transfer, nor can a local transfer be conditional.

Figure 1–6. Conditional Data Unit and/or Global Transfer Format

63	38	21 0
Data Unit Op(s)	COND	Global Transfer

If the data unit ALU operation is conditional and uses a non-D operand, both the global transfer and local transfer fields are required; thus, no operations can be specified in parallel. This instruction format is shown in Figure 1–7.

Figure 1–7. Conditional Data Unit With Non-D Operand Format

63	38	21 0
Data Unit Op(s)	COND	ALU Non-D Operand

To specify a long immediate offset for one address unit requires the opcode bits that would have specified the other address unit's operation. In the case of a local transfer with a long offset, the local transfer may load or store a non-D register. These instruction formats are shown in Figure 1–8.

Figure 1–8. Long Offset Formats

63	38	21 0
Data Unit Op(s)	Local Transfer	Long Offset

63	38	21 0
Data Unit Op(s)	Long Offset	Global Transfer

Chapter 2

Understanding the PP Memory Organization

This chapter describes how the PP accesses on-chip and off-chip memory, the crossbar, and big- and little-endian ordering.

Before you read this chapter, you should review Chapter NO TAG, *Understanding the MVP Memory Organization*, and Chapter NO TAG, *The Crossbar*, in the *MVP System-Level Synopsis* to understand how memory for the entire MVP is organized.

Topics

2.1	**Understanding the PP Memory Map**	**PP: 2-2**
2.2	**Understanding the Crossbar**	**PP: 2-7**
2.3	**Big-Endian and Little-Endian Addressing**	**PP: 2-9**

2.1 Understanding the PP Memory Map

The MVP has sixteen 2K-byte on-chip RAMs that can be accessed by any of the MVP processors in a given cycle. These RAMs are referred to as shared RAMs.

These shared RAMs are a key feature of the MVP. Single-cycle access to the on-chip shared RAMs by any of the MVP processors reduces the traditional bottleneck for multichip parallel processing, such as delays associated with passing data between different memory spaces and wait-stated access to an off-chip shared memory space.

The PPs operate primarily on data stored in the on-chip shared RAMs. Multidimensional data transfers between off-chip and on-chip memory (referred to as packet transfers) are performed by the transfer controller in parallel with PP processing. This allows the PPs to focus their processing power on the manipulation of data, rather than the collection or distribution of data.

Although PPs operate primarily on data that has been brought on-chip via packet transfers, there are a few cases (such as polling a single word in external memory) in which it is desirable to specify an external memory access directly in the PP instruction. For more information about direct external memory accesses (DEAs), see Section NO TAG, *Direct External Memory Access (DEA)*, in the *MVP System-Level Synopsis*.

Each PP can perform two independent data accesses and fetch an instruction in a single cycle. When the PP specifies two data accesses in the same cycle, one access occurs over the PP's global port and the other access is attempted over the local port. The PP's global port can access any of the shared RAMs. The PP's local port can access only the three data RAMs and one parameter RAM that are physically adjacent to the PP.

These RAMs are referred to as the PP's local RAMs. Each PP has its own set of local RAMs, which are a subset of the shared RAMs. If an access is attempted over the local port to a nonlocal RAM, the PP's pipeline stalls and the access is retried on the following cycle over the global port.

The MVP memory map is shown in Figure 2–1. The MVP memory map is divided into two memory spaces: on-chip and off-chip.

2.1.1 On-Chip Memory

The address range from 0x0000 0000 to 0x01FF FFFF is used for MVP on-chip memory. Much of this address space is reserved for future versions of the chip. An MVP has 25 2K-byte memory modules: twelve data RAMs, five parameter RAMs, six instruction cache RAMs, and two data cache RAMs.

- **Data RAMs**

  Single-cycle access RAMs used for general-purpose data storage.

- **Parameter RAM**

  Single-cycle access RAM partially assigned to hardware system control functions, including interrupt vectors, the start of linked-list packet transfer information, and a scratch-pad for temporary information about a suspended packet transfer.

  24 bytes are set aside for use by whatever operating system is running on the MVP. They are intended to provide a data area at a fixed address that is known to the system software on both the PP and MP. This area is useful for communications and operating-system parameters.

  The parameter RAM also has a nondedicated space that can be used for general-purpose data storage. This space is often used for task parameter passing, interprocessor messages, and packet transfer parameters.

  The top of the nondedicated RAM space is used by the PP's system stack. Upon reset, the PP system stack is set to address 0x0100 #7F0 (where # is the PP number) in parameter RAM and grows toward lower addresses. The size of the stack is dependent on the application code; there is no fixed boundary between the general-purpose space in the parameter RAM and the stack.

  Figure 2–2 shows a sample map using parameter RAM.

- **Instruction Cache**

  Each PP has its own single-cycle access instruction cache, as described in Chapter 3, *PP Instruction-Cache Operation and Interprocessor Communications*. This allows the MVP to perform MIMD (multiple-instruction, multiple-data) parallel processing.

The data and PP parameter RAMs are grouped into two categories:

- **Shared RAMs**

  The 12 data RAMs and 4 PP parameter RAMs that can be accessed by any PP (over the global port), the MP, or the TC. Note that only the shared RAMs can be accessed via the crossbar by the PPs. Information exchange between a shared RAM and the MP's data cache or MP parameter RAM must be performed by the MP.

- **Local RAMs**

  A PP-specific subset of the shared RAMs that consists of three local data RAMs and a local parameter RAM that can be accessed by the PP over its local port.

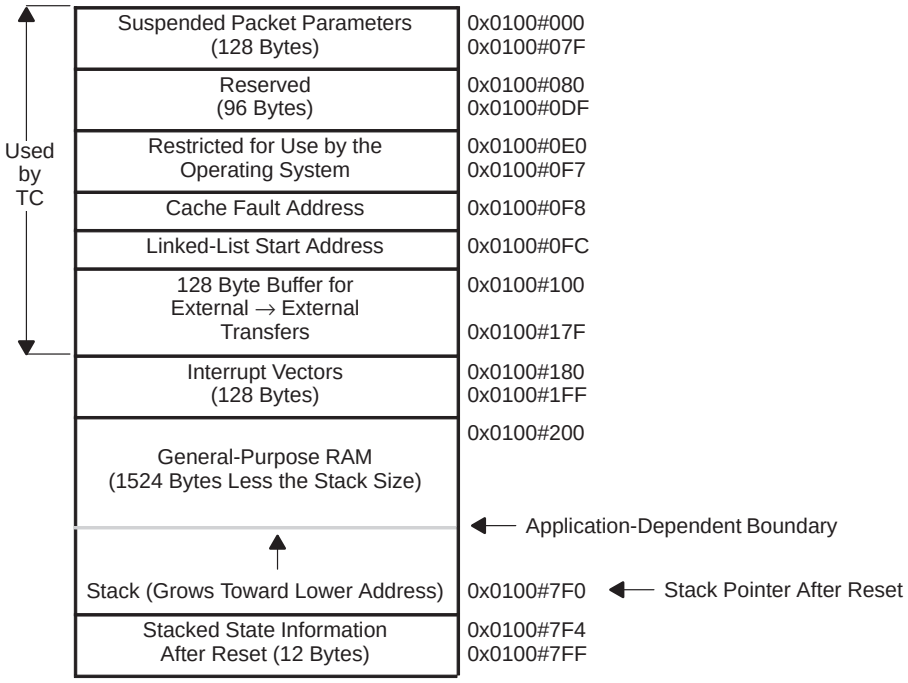
2.1.2 Off-Chip Memory

The address range from 0x0200 0000 to 0xFFFF FFFF is used for off-chip memory. A PP can directly access single words in off-chip memory by using the direct external access mechanism described in Section NO TAG, *Accessing External Memory*, in the *MVP System-Level Synopsis*.

Figure 2–1. MVP Memory Map

LSbyte Address		MSbyte Address
0×0000 0000	PP0 Data RAM 0	0×0000 07FF
0×0000 0800	PP0 Data RAM 1	0×0000 0FFF
0×0000 1000	PP1 Data RAM 0	0×0000 17FF
0×0000 1800	PP1 Data RAM 1	0×0000 1FFF
0×0000 2000	PP2 Data RAM 0	0×0000 27FF
0×0000 2800	PP2 Data RAM 1	0×0000 2FFF
0×0000 3000	PP3 Data RAM 0	0×0000 37FF
0×0000 3800	PP3 Data RAM 1	0×0000 3FFF
0×0000 4000	Reserved	0×0000 7FFF
0×0000 8000	PP0 Data RAM 2	0×0000 87FF
0×0000 8800	Reserved	0×0000 8FFF
0×0000 9000	PP1 Data RAM 2	0×0000 97FF
0×0000 9800	Reserved	0×0000 9FFF
0×0000 A000	PP2 Data RAM 2	0×0000 A7FF
0×0000 A800	Reserved	0×0000 AFFF
0×0000 B000	PP3 Data RAM 2	0×0000 B7FF
0×0000 B800	Reserved	0×00FF FFFF
0×0100 0000	PP0 Parameter RAM	0×0100 07FF
0×0100 0800	Reserved	0×0100 0FFF
0×0100 1000	PP1 Parameter RAM	0×0100 17FF
0×0100 1800	Reserved	0×0100 1FFF
0×0100 2000	PP2 Parameter RAM	0×0100 27FF
0×0100 2800	Reserved	0×0100 2FFF
0×0100 3000	PP3 Parameter RAM	0×0100 37FF
0×0100 3800	Reserved	0×0100 FFFF
0×0101 0000	MP Parameter RAM	0×0101 07FF
0×0101 0800	Reserved	0×0180 17FF
0×0180 1800	PP0 Instruction Cache	0×0180 1FFF
0×0180 2000	Reserved	0×0180 37FF
0×0180 3800	PP1 Instruction Cache	0×0180 3FFF
0×0180 4000	Reserved	0×0180 57FF
0×0180 5800	PP2 Instruction Cache	0×0180 5FFF
0×0180 6000	Reserved	0×0180 77FF
0×0180 7800	PP3 Instruction Cache	0×0180 7FFF
0×0180 8000	Reserved	0×0180 FFFF
0×0181 0000	MP Data Cache 0	0×0181 07FF
0×0181 0800	MP Data Cache 1	0×0181 0FFF
0×0181 1000	Reserved	0×0181 7FFF
0×0181 8000	MP Instruction Cache 0	0×0181 87FF
0×0181 8800	MP Instruction Cache 1	0×0181 8FFF
0×0181 9000	Reserved	0×0181 FFFF
0×0182 0000	Memory-Mapped TC Registers	0×0182 01FF
0×0182 0200	Memory-Mapped FC Registers	0×0182 03FF
0×0182 0400	Reserved	0×01FF FFFF
0×0200 0000	External Memory	0×FFFF FFFF

Figure 2–2. PP Parameter RAM Contents



Note: The # symbol represents the PP number.

2.2 Understanding the Crossbar

The MVP crossbar shown in Figure 2–3 allows each PP to perform two independent parallel data accesses to the on-chip shared RAMs and one instruction fetch every cycle. In the figure, each port connects to a horizontal line that indicates which RAMs the port can connect to. As Figure 2–3 shows, each PP has three crossbar ports:

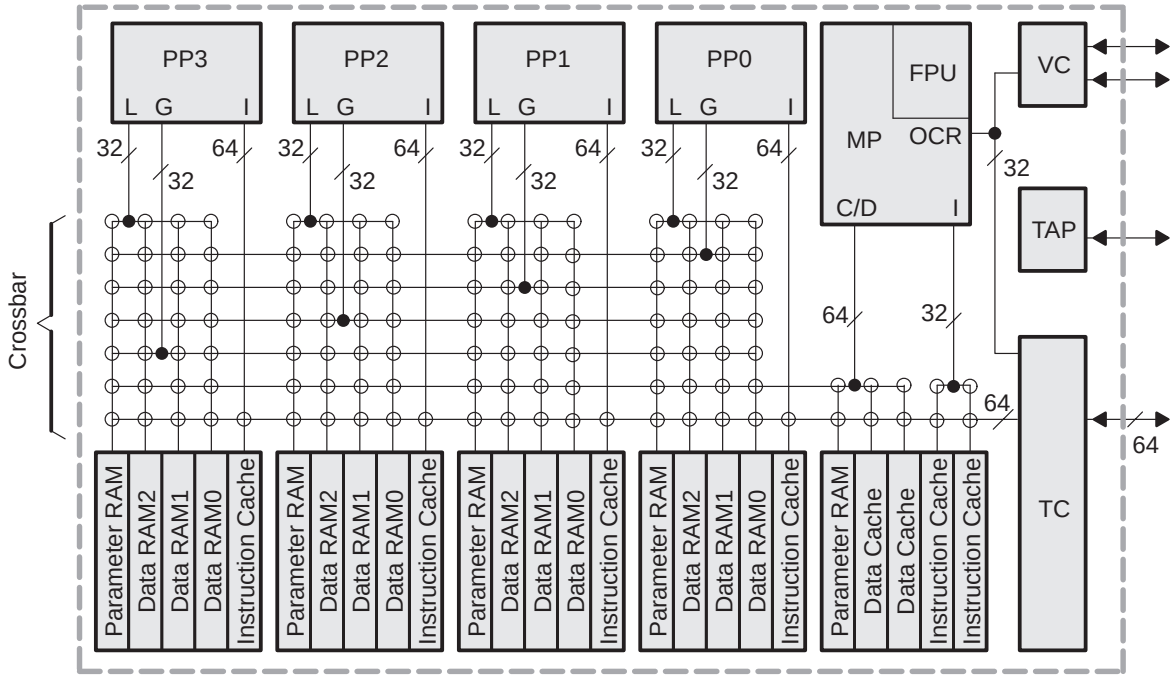
- The **global port** connects to any of the shared RAMs. If an access is attempted over the global port to an address that is not in the shared RAMs, a direct external access request is sent to the transfer controller (see Section NO TAG, *Direct External Memory Access (DEA)*, in the *MVP System-Level Synopsis*).
- The **local port** connects to any of its local RAMs. When a PP attempts a memory access over the local port to an address that is not in its local RAMs, the access is diverted to the global port and tried on the following cycle. This is referred to as local to global diversion.
- The **instruction port** accesses instructions from the PP's instruction cache.

In Figure 2–3, each ⊕ represents a crosspoint between a processor (horizontal) bus and a RAM (vertical) bus, allowing connections between the processors and the RAMs to be made.

These crosspoints switch on a cycle-by-cycle basis, according to the addresses output by the processors, allowing them to share memory dynamically. A processor can access a specific RAM by using an address that is in that RAM. The crossbar automatically connects the processor to the RAM for that cycle.

For information concerning crossbar connections, access decisions, and command words, refer to Chapter NO TAG, *The Crossbar*, in the *MVP System-Level Synopsis.*

Figure 2–3. MVP Crossbar



Note:

L	=	Local port	G	=	Global port
I	=	Instruction port	FPU	=	Floating-point unit
OCR	=	On-chip register port	C/D	=	Cache/data port
TAP	=	Test access port	TC	=	Transfer controller
VC	=	Video controller	MP	=	Master processor
PP0–3	=	Parallel processors 0–3			

2.3 Big-Endian and Little-Endian Addressing

The MVP is designed to interface to external hardware via the TC and can be configured to use either big-endian (for example, 360, 370, 68xxx) or little-endian (for example, VAX, 80x86) byte orderings. The PPs (as well as the MP) support data accesses for both types of byte ordering. Note that the MVP endian is configured at reset.

Note that for diagrams in this user's guide, these conventions are followed:

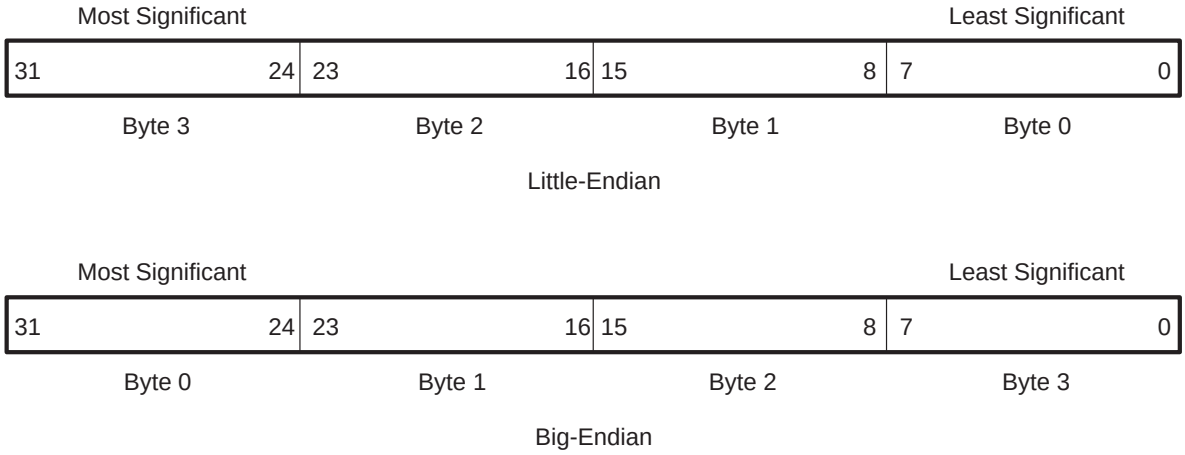
- Register bit 0 is least significant, bit 1 is next most significant, and bit 31 is the most significant, and so on (thus following little-endian bit numbering, regardless of the endian mode).
- The most significant bit of the register is shown on the left.

Coding examples in this user's guide assume that the MVP is operating in big-endian mode unless specifically stated otherwise. Note that in most instances, PP code is endian-independent.

2.3.1 Byte Ordering

With big-endian byte ordering, the most significant byte has an address of 0, while with little-endian byte ordering, the least significant byte is numbered 0. Note that the ordering of bits within a byte is unchanged (this is achieved by suitably wiring the bits of the MVP's data bus to that of the host system). However, the numbering of the bytes within a word is different in the two modes, as shown in Figure 2–4.

Figure 2–4. Byte Ordering—Register View



If data is always loaded from memory with the same data size that it was stored with, PP code can be made endian-independent. However, when data is stored with one data size (such as word) but loaded in a shorter data size (such as halfword), code becomes endian-dependent. Example 2–1 illustrates code that is endian-dependent:

Example 2–1. Endian-Dependent Code

```
d7 = 0xFFFF0000
*a8 = d7
d7 =h *(a8+[1])        ;    d7 = 0x00000000 if big endian
                       ;    or
                       ;    d7 = 0xFFFFFFFF if little en-
                            dian
```

For more information on endian ordering, refer to Section NO TAG, *Endian Ordering*, in the *MVP System-Level Synopsis*.

Chapter 3

PP Instruction-Cache Operation and Interprocessor Communications

This chapter discusses cache management and interprocessor communications.

Topics

3.1	PP Instruction Cache	PP: 3-2
3.2	Interprocessor Commands	PP: 3-8

3.1 PP Instruction Cache

Most program execution time is spent on repeatedly executing a few main procedures or loops. Program execution on the PP is accelerated by keeping these often-used code blocks in the PP's on-chip instruction cache. Each PP has its own 2K-byte instruction cache, thus allowing the MVP to operate in a MIMD (multiple instruction multiple data path) model of parallel processing.

The program flow control unit (PFC) of the PP manages the opcode fetches from the PP's instruction cache. It also performs the instruction-cache control, including handshakes with the transfer controller when cache miss servicing is required. The transfer controller actually services the cache.

3.1.1 Cache Architecture

Figure 3–1 illustrates the PP's instruction cache organization. The instruction cache contains 2K bytes of RAM for storing up to 256 64-bit instructions. As Figure 3–1 shows, the cache is divided into four blocks. Each block has an associated tag register that indicates from where in memory the instructions in that block are cached.

Each cache block is further divided into four subblocks; each subblock contains 16 instructions. Cache servicing is performed on subblocks, so only 16 instructions are loaded into the cache when a cache miss occurs. This reduces the cache servicing overhead when fewer than 64 instructions in a block are used.

Associated with each of the four cache blocks is a cache tag register (see Chapter 7, *Summary of PP Registers*).

Figure 3–2 shows how the bits of the pc register are used in the cache operation.

Figure 3–1. PP Instruction Cache

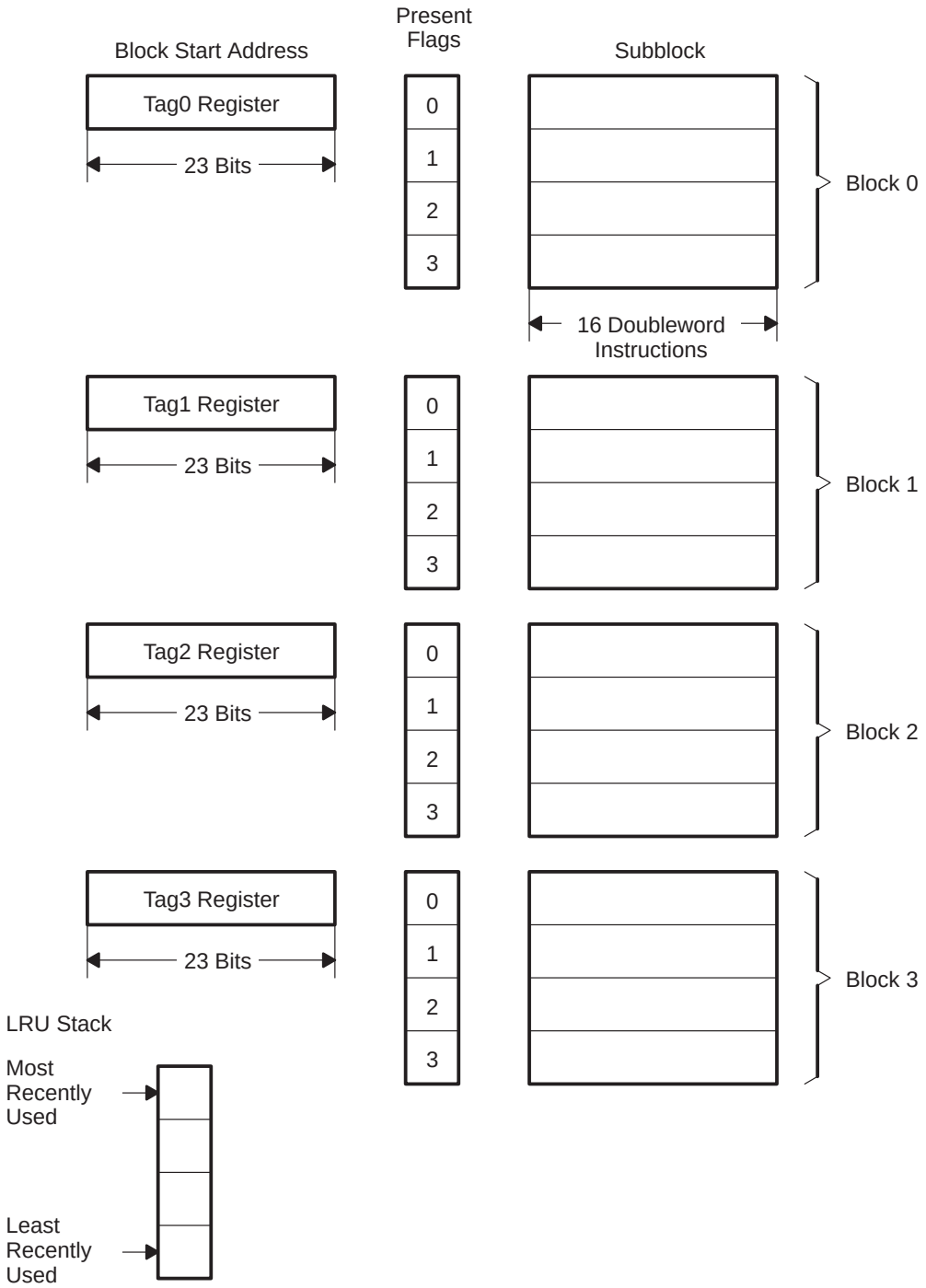


Figure 3–2. pc Register Bit Usage for Cache Operations

31 … 9	8 … 7	6 … 3	2 … 0
23-bit tag value	sub	instruction	ignored

PC Register

Notes: 1) Bits 31–9 correspond to the 23-bit tag register value, which is the address of the block that the instruction is in.

2) Bits 8–7 indicate the subblock within the block that contains the instruction. These bits also correspond to the present bit for the subblock.

3) Bits 6–3 indicate which of the 16 instructions contained in the subblock should be fetched.

4) Bits 2–0 are ignored for instruction fetch and cache operations.

3.1.2 Cache Operation

Each time an instruction fetch is attempted, the PFC checks to see if the cache contains the required instruction. First the processor compares the 23 MSBs of the pc register to the 23-bit tag values. If the PFC doesn't find a match with any tag register, the instruction is not in the cache. This type of cache miss is referred to as a block miss. If the PFC does find a match, it searches for the appropriate subblock. A present (P) bit in the cache tag, associated with each subblock, indicates the presence of a particular subblock within a cache block:

- **P = 1** indicates that the requested instruction word is in the cache. This is called a cache hit.
- **P = 0** indicates that this particular subblock is not in the cache, and, therefore, the instruction is not in the cache. This type of cache miss is called a subblock miss.

A **cache hit** means that the cache contains the requested instruction. In this case, the PFC performs the following actions:

- Fetches the instruction from the cache.
- Moves the block number to the top of the LRU stack, pushing the other three block numbers toward the bottom of the stack.

Because of pipelining, instruction fetches from the cache overlap the address stage and execute stage of the two previous instructions. Thus, the overhead of cache operation (when a cache miss does not occur) is effectively 0.

A **cache miss** means that the cache does not contain the instruction that must be fetched. There are two types of cache misses: block miss and subblock miss.

- **Block Miss**

  The 23 MSBs of the PC do not equal any of the 23-bit tag register values. In this case, the least-recently-used (LRU) replacement algorithm (described in Section 3.1.3) chooses which block to discard and clears all the present bits associated with that block. Then the required subblock is loaded and the corresponding P bit is set to 1.

- **Subblock Miss**

  One of the 23-bit tag register values equals the 23 MSBs of the pc register, but the present bit in the cache tag (corresponding to the cache subblock indicated by bits 8 and 7 of the pc register) is 0. A cache service request is issued to the TC. During the cache service, the PP's pipeline is stalled. After the TC loads the required subblock, the corresponding P bit is set to 1, the instruction fetch is performed, and the PP resumes execution.

If the TC can immediately service the cache miss and off-chip memory is single-cycle, cache servicing requires about 20 cycles. During a cache service, the PP's pipeline is stalled. After the cache is serviced, the instruction fetch is performed and the PP resumes execution. Each servicing of the cache loads one subblock.

3.1.3 Cache Replacement Algorithm

When the PP requests an instruction from a block that is not in the cache (a block miss), the contents of one of the four cache-resident blocks must be discarded to make room for the segment that contains the requested instruction. The PP instruction cache uses a least-recently-used (LRU) replacement algorithm to select the segment to be discarded. The program flow control unit keeps track of which block in the cache is the LRU block. When a new block must be loaded into the cache, the LRU block is discarded.

3.1.4 Cache Reset

The PP's instruction cache is cleared either by the MP or by the PP issuing a cache-reset command as described in subsection 3.2.1, *Issuing an Interprocessor Command*. A cache-reset causes the following:

- The 23-bit tag values are set to the tag's own number (for example, tag3 = 3, tag2 = 2, tag1 = 1, tag0 = 0).
- The program flow control unit clears all the present bits, and sets the LRU bits to the tag's own number, that is:

  tag3(LRU) = 11b

  tag2(LRU) = 10b

  tag1(LRU) = 01b

  tag0(LRU) = 00b

  Thus, tag3 is considered least recently used following a cache reset.

Note:

Reset does not flush the cache.

3.1.5 Code Alignment

To minimize cache misses, it is often critical to ensure that tight-loop code does not cross over cache subblock and especially cache block boundaries. To do this, align sections of code to 128 or 512 byte-aligned addresses, respectively (using the assembler's **.align** directive described in on page NO TAG of the *MVP Code Generation Tools User's Guide*). This can prevent unnecessary cache misses that stall the processor for cache servicing and consequently yield lower performance.

3.1.6 Self-Modifying Code

When a program modifies its own instructions, only the copy of the instruction that resides in memory is affected. Copies of the instructions that reside in cache are not modified, and the internal control logic does not attempt to detect this situation. The cache must be reset in order for the new code to be copied into the cache. You must use caution with self-modifying code because a PP cannot reset other PPs' caches.

3.2 Interprocessor Commands

Before you read this section, you should review Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis* to understand how communication between processors is performed.

You can issue an interprocessor command such as a message interrupt to another PP by setting up a 32-bit command word in a register and then moving the contents in that register to the **cmnd** pseudoregister. This is encoded in the opcode as a move operation and thus can be performed in parallel with a data unit operation and/or a local transfer.

Interprocessor commands allow a PP to send interrupts to on-chip processors (including itself), request that its own operation halt, or reset its instruction cache. Interprocessor commands are also used by the MP to issue the following commands to any or all of the PPs:

- Reset
- Halt
- Unhalt
- Task Interrupt
- Message Interrupt
- Instruction-Cache Reset

3.2.1 Issuing an Interprocessor Command

Before a command is issued, the desired operation(s) and destination processor(s) must be specified in a register according to the 32-bit command word format that is detailed in Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis*. An interprocessor command is issued by a register-to-register move from the register containing the command word to the read-only zero register (a15).

The assembler keyword **cmnd** is used as a pseudoregister to indicate that an interprocessor command is being issued. Use of the cmnd keyword ensures that the performed operation is a register-to-register move. An interprocessor command can be issued in parallel with a data unit operation and a local transfer.

Example 3–1. Assembler Command Keyword

```
cmnd = d1                    ;nop
                             ;||a15 = d1
```

Example 3–2. Interprocessor Command With Parallel Operations

```
d2 = d0 * d1
        ||d5 = d3 + d4
        ||cmnd = x1                          ;a15 = x1
        ||d6 =b *(a4+=[x0])
```

As shown in Example 3–3, two delay-slot instructions are fetched between when an interprocessor command is fetched and when the interprocessor command is executed.

The two delay-slot instructions following a halt command must be legal operations; otherwise, an error will occur. Stores to memory specified by the first delay-slot instruction of a halt command are completed during the halt.

Loads from memory are completed to a temporary latch. The value is written to the destination register only after the PP is unhalted. Likewise, the execution of both delay slot instructions is performed (as if the halt had never occurred), once the PP is unhalted.

Example 3–3. Interprocessor Command Delay Slots (Assuming No Interrupts)

```
cmnd = x1
<Instruction1>
<Instruction2>
; ----------------> cmnd event occurs here
<Instruction3>
```

When a pipeline stall occurs for any reason during the execute stage of an instruction issuing an interprocessor command, the command is not transmitted until after the stall has been resolved. For example, a command on the global port will wait until an access diverted from the local to the global port has completed. This gives consistency to the delay slots of an interprocessor command that is transmitted during the execute pipeline stage.

Example 3–3 assumes that no interrupt occurs between the instruction issuing the command and Instruction2. If an interrupt occurs, the two pipeline-stage delay slots are completed during the interrupt pseudoinstructions, and the command is executed then. If interrupts are enabled and expected, you should ensure that the exact position of the command effect is noncritical.

3.2.2 Using Command Words

The general format for a command word sent by the MVP processors is shown in Figure 3–3. Note that a command word contains some bits indicating the desired operation(s) and other bits indicating the destination processor(s). Only those processor(s) identified by the designator bits will respond to the encoded operation.

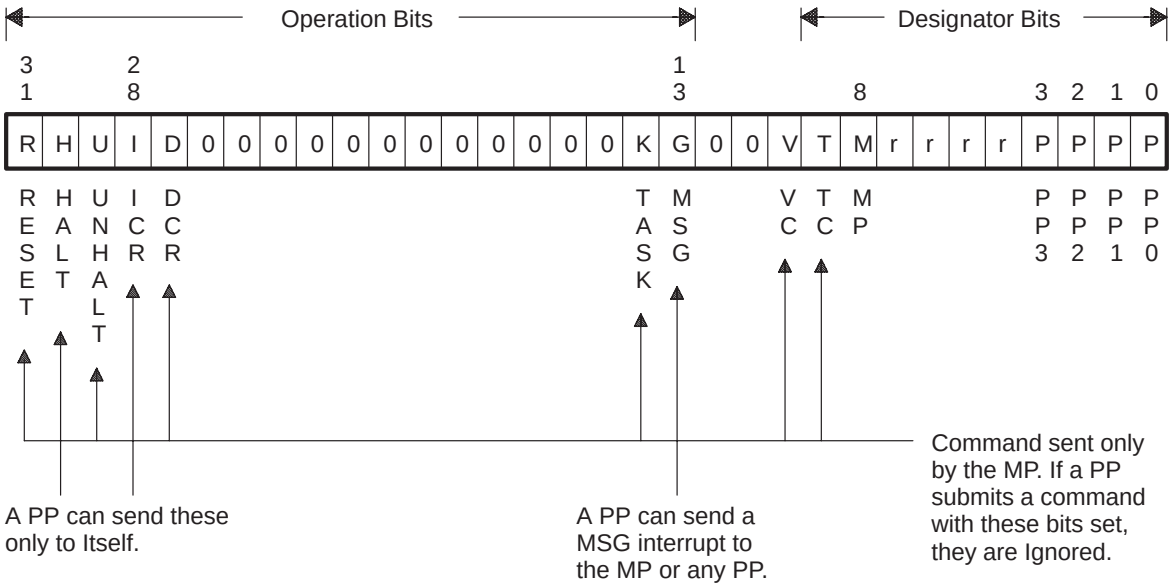
Some command word restrictions apply to the PP. A PP can send halt, or instruction-cache reset commands to itself only. Software reset, unhalt, data-cache reset, and task interrupt commands can be issued only by the MP.

Additionally, VC and TC commands can be issued only by the MP. If a PP submits an interprocessor command with 1s in bits that are valid only from the MP, those bits are ignored.

Note:

Currently, if a PP issues a halt and/or an instruction-cache reset command with the designator bit for other PPs set to one, then these other PPs ignore the command. However, for code portability to future generations of the MVP, it is a good idea to make sure that the designator bits are set to 1 only for the processor issuing the command. All other PP designator bits should be cleared to 0.

Figure 3–3. Command Word Bit Assignment



❑ **Resetting a processor (R bit)**

A software reset can be issued by the MP to any of the PPs. Refer to Section 9.6 for more information on software resets.

❑ Halting a PP (H bit)

Submitting a command with this bit set causes the appropriate PP's halt latch to be set, stalling the PP's pipeline. Halt behaves very much like infinite contention in that nothing is reset, and no program state is modified. During halt, interrupts are not serviced, but they are latched and will be processed in order of priority, if interrupts are enabled when the PP is unhalted. The PP remains halted (with its pipeline stalled) until it is unhalted by the MP. When the unhalt occurs, the PP resumes code execution as though nothing had happened.

A halt command can be issued by the MP or by the PP itself. A PP cannot issue a halt command to another PP or to the MP. If the MP requests an unhalt at the same time a PP requests a halt, the MP unhalt dominates. The PP halt latch value is visible within the PPERROR register of the MP, allowing the halt state to be observed by the MP at any time.

When a PP issues a halt command to itself, it must set its own PP designator bit, or else the command will not be executed. Example 3–4 shows PP-independent code for issuing a halt command and a simultaneous message interrupt to the MP and to itself.

Example 3–4. Set-Up and Submission of Halt Command

```
d1 = comm & 7          ;Extract PP number.
d1 = 0x1\\d1           ;Convert PP number to
                       ;corresponding
                       ;PP designator bit.
d1 = 0x40002100 | d1   ;Merge PP designator bit
                       ;with Halt/MSG Interrupt Command
                       ;to MP.
cmnd = d1              ;Submit the Command.
<Instruction>          ;Delay Slot1 must contain a legal
                       ;instruction.
<Instruction>          ;Delay Slot2 Must contain a legal
                       ;instruction.
```

❑ Unhalting a PP (U bit)

An unhalt command clears a PP's halt latch to 0. An unhalt command can be issued only by the MP. Unhalt commands fall into two categories:

■ Unhalt after Reset

The MP issues an unhalt command to start a PP following a hardware or software (via an interprocessor command) reset. After a reset-induced halt and subsequent unhalt, the PP begins code execution at the address given by the PP's task interrupt vector.

When the MP issues a reset command to a PP, the MP must wait until the PP halts, as specified in the MP's PPERROR register, before it sends an unhalt command. A software reset command (submitted by the MP) will always cause a halt, even if the unhalt bit is set in the command word.

■ Unhalt after Halt

For unhalt commands following a halt command, the PP restarts code execution as though nothing had happened.

An MP unhalt command dominates over a PP halt command if they are simultaneous. Unhalt dominates if the MP submits a command word with both the halt and unhalt bits set.

❑ Resetting the instruction cache (I bit)

An instruction-cache reset command can be issued by the MP or by the PP itself, but not by another PP. A cache reset is performed by clearing all the present bits in the cache tag registers. The MP should issue an instruction-cache reset command before the PP is unhalted the first time after the system is initialized.

❑ Sending a task interrupt (K bit)

The K bit is ignored if set in a command word from any PP (including itself). When set to 1 in command word from the MP, the K bit indicates that the designated PPs should set their task interrupt flags in the intflgs register. If the task interrupt is enabled, the PP will take the task interrupt. Task interrupts are typically used to make the PP switch tasks under MP control.

- **Sending a message interrupt (G bit)**

  This bit, used in conjunction with the processor designator bits, indicates which processor(s) should set their message interrupt flag and take the message interrupt if it is enabled.

  Message interrupts provide a mechanism for on-chip interprocessor communication. For example, if a PP is performing a task for the MP, it can send a message interrupt to the MP when it has completed the task. Note that the meaning of the message interrupt is application-specific. In the multitasking executive, its meaning is qualified by a mailbox value to allow messages to have various meanings.

- **Sending a command to the MP (M bit)**

  The MP designator bit, when written as a 1 by a PP in conjunction with a 1 in the command word's message interrupt (G) bit, indicates to the MP that it should set its message interrupt flag corresponding to that PP.

- **Sending a command to the PPs (P bit)**

  The PP designator bits, when issued as 1s by a PP in conjunction with a 1 in the command word's message interrupt (G) bit, indicate which PP(s) should set their PP message interrupt flag corresponding to the PP sending the word.

  When a PP issues a halt or cache reset command to itself, it must set its own PP designator bit. Refer to Example 3–4 for an example of how to set a PP's own designator bit with PP-independent code for portability.

  When issued by the MP, these bits qualify which PP(s) should respond to the operation bits (reset, halt, unhalt, and/or instruction-cache reset) or interrupt (task or message) bits coded in the rest of the command word.

Chapter 4

Understanding the Data Unit

The PP's data unit supports the massive processing associated with algorithms such as frequency domain transforms, correlation, and filters. It also supports the bit-field and pixel manipulations required by image coding and computer graphics. The combination of special hardware and a flexible data path allows a single-cycle data unit operation to perform the equivalent of many general-purpose processor instructions.

This chapter describes the architecture of the data unit including the three-input ALU and the multiplier data path.

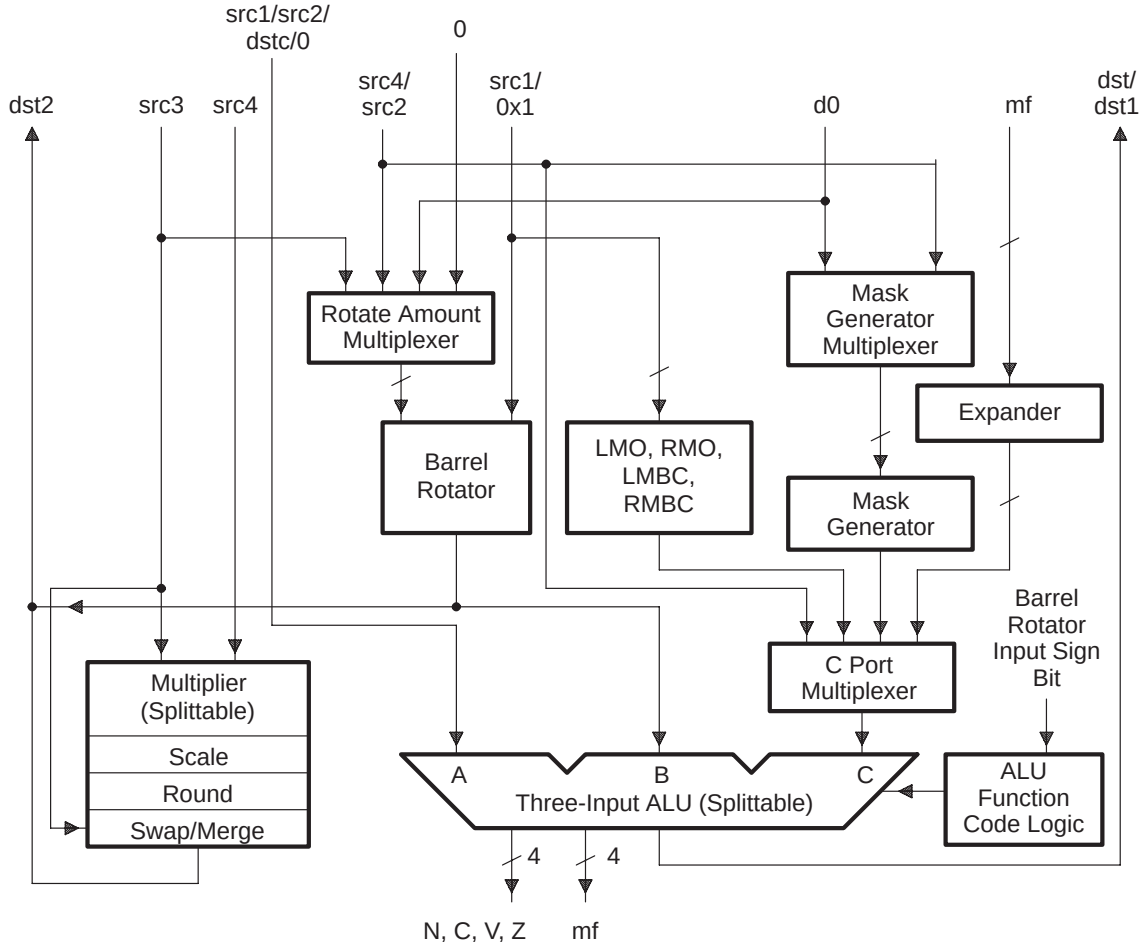
Topics

4.1	Understanding the PP Data Unit	PP: 4-2
4.2	Understanding the ALU Data Path Hardware	PP: 4-6
4.3	Using the Three-Input ALU	PP: 4-11
4.4	Using Data Path Multiplexers	PP: 4-31
4.5	Understanding ALU Data Path Classes	PP: 4-32
4.6	Combining Two or More ALU Data Path Functions	PP: 4-38
4.7	Using the Multiplier Data Path	PP: 4-67

4.1 Understanding the PP Data Unit

Figure 4–1 shows the data unit's data path (not including the data unit registers). The data unit includes two data paths: the multiplier data path and the ALU data path. Each data path has its own set of hardware functions that are independent of the other data path. The multiplier data path includes a 16 × 16-bit multiplier, a halfword swapper, and rounding hardware. The ALU data path includes a 32-bit three-input ALU; a barrel rotator; a mask generator; a multiple flags expander; logic to detect the bit number of the leftmost one (lmo), rightmost one (rmo), leftmost-bit change (lmbc), or rightmost-bit change (rmbc) in a register; and several multiplexers.

Figure 4–1. Data Unit Block Diagram



4.1.1 The Data Unit Registers

The data unit contains the following registers:

- Eight **data registers** (**d0–d7**). You can perform up to seven reads and up to four writes to the data registers in a given cycle. The d0 register is used as a configuration register for the ALU and multiplier data path control for certain instructions. Also, the contents of the d0 register can be used as a default operand for some operations.
- A **status register** (**sr**). The status register holds four bits of status (based on the result of ALU operations) plus bits that configure the split ALU and @mf operations.
- A **multiple flags register** (**mf**). The multiple flags register (mf) holds status information (carry-out(s), zero-compare(s), sign(s), or extended-sign(s)) from split-ALU operations. The mf register is connected to special hardware that expands 4, 2, or 1 bits of the mf register 8, 16, or 32 times, respectively, to fill out a 32-bit word.

4.1.2 Inputs and Outputs

As shown in Figure 4–1, the data unit supports a number of different operands going to and from the multiplier and ALU data paths. The source operands coded in the instruction are routed to different hardware inputs, depending on the data routing class (as described in Section 4.5, *Understanding ALU Data Path Classes*).

Other inputs, including the constant 0x1, the multiple flags (mf) register, and the default rotate amount specified in d0, are associated with certain data routing classes. The list below describes the various inputs and outputs from the data path shown in Figure 4–1:

- **src1**

  In most cases, src1 can be any PP register. src1 is restricted to a D register if it is an input to the lmo, rmo, lmbc, or rmbc hardware. If this operand is a non-D register, there cannot be a global address unit operation in the same instruction. This operand always goes to the ALU data path.

- **src2**

  Can be a D register or sometimes a 5- or 32-bit immediate. This operand always goes to the ALU data path.

- **src3**

  Must be a D register. This operand can specify either the first input the multiplier or the barrel rotate amount for the ALU data path.

- **src4**

  Must be a D register. This operand can specify the second input the multiplier, the input to the C port of the ALU, or the input to the mask generator.

- **dst/dst1**

  ALU result destination register. The ALU result can be written to any PP register. If the destination register is a non-D register, then there cannot be a global address unit operation in the same instruction.

- **dst2**

  Must be a D register. Normally, dst2 is the multiplier data path destination register. There are also two six-operand (four source and two destination operands) ALU data path opcode formats that write the output of the barrel rotator to dst2, as well as to the B input port of the ALU.

- **dstc**

  Destination companion register source. This is always a D register. It is the same register as dst1 if dst1 specifies a D register. If dst1 is a non-D register, then dst1c is the D register with the same three register code LSBs as the non-D register (the corresponding D register is called the companion register). Many instructions support using this as third source to the ALU data path.

- **0x1**

  A constant that can be sent to the barrel rotator. Intended for use in bit manipulation or bit testing.

- **0**

  An input of 0 can be forced into the A port of the ALU. Also, the rotate amount can be set to 0 so that no rotation is performed.

- **d0**

  The five LSBs of d0 serve as the default rotate and/or masking amount for some instructions.

4.2 Understanding the ALU Data Path Hardware

The ALU data path has special hardware not commonly found within other processors which can be very useful for image, graphics, and other processing. The PP's instruction set makes most of this hardware directly accessible. This combination of hardware capability and flexibility allows you to perform in one ALU operation what could require several instructions on most other architectures. Making hardware accessible exposes some aspects of the CPU architecture that are hidden on most other processors.

The PP instruction set supports using elemental functions as building blocks to form useful operations. In the following sections, these elemental functions are described individually. Subsequently, combinations of two or more of these functions that form useful ALU data path operations are described.

The term **function** refers to the individual action of each part of the data path; the term **operation** refers to the operation performed by the combination of the individual data path functions.

4.2.1 Using the Barrel Rotator

The data path into the B port of the ALU contains a barrel rotator capable of rotating the bits in a 32-bit number from 0 (no rotation) to 31 bits in a single pass. The barrel rotator has two inputs:

- The 32-bit number to be rotated
- A 5-bit rotation amount

The rotation amount specifies how many positions the bits are rotated leftward. With rotation, unlike shifting, none of the bits are discarded; bits that rotate out the left-hand side are rotated in on the right-hand side. Since the bits rotate circularly, the net effect of a right rotate by n can be achieved by specifying a left rotate amount of 32–n. As shown later, the barrel rotator can be used in combination with the mask generator and the ALU to perform both left- and right-shift operations.

Example 4–1 demonstrates rotation left by 4 bits (or right by 28 bits) of the number 0x5CFF FFA7. Note how the hex digit 5, which rotates out the left-hand side, is rotated in on the right-hand side of the result.

Example 4–1. Rotate Left by 4

```
Original Number       01011100111111111111111110100111   (0x5cffffa7)

Rotate Number by 4    11001111111111111111101001110101   (0xcffffa75)
```

Use of the rotator can be directly specified in the PP instruction. The PP assembly language defines two back-slashes (****) as the rotation operator. The algebraic assembler notation x\\y specifies a rotation where x is the register to be rotated and y is the rotation amount that can be specified either in a register or by an immediate value.

4.2.2 Using the Mask Generator Function

The data path into the C port of the ALU contains a mask generator. The mask generator takes an unsigned 5-bit input and produces that number of right-justified ones, zero-filled on the left in the corresponding 32-bit output (in other words, $2^n - 1$, where n = 5-bit input). For example, if the number 0x4 is input to the mask generator, the output will be 0x0000 000F (four right-justified 1s).

The mask generator function is specified in PP assembly language by the symbol **%**. % is a unary operator that is applied to the contents of the register or immediate after the symbol. Thus, %5 produces 0x0000 001F. In the special case of an all-zero input, the instruction can specify whether to generate all (32) 1s (%!) or all 0s (%). As shown later, the %! option supports, among other uses, right shifts.

As discussed in more detail later, the 32-bit ALU can be split into either two 16-bit or four 8-bit segments. In these cases, the mask generator can be used to generate replicated masks. With certain instructions, a 32-bit mask can optionally be retained, even when the ALU is split. An example of replicated masks is shown in Example 4–2.

For 16-bit splits of the ALU, the mask generator uses the four LSBs of the input and generates two replicated 16-bit outputs. For 8-bit splits of the ALU, the mask generator uses the three LSBs of input to generate four replicated 8-bit outputs. As with the unreplicated 32-bit case, an input of 0 can optionally be treated as specifying all or no 1s in the output. Replicated masks for 8- and 16-bit split-ALU operations support masking of parallel sets of data.

It should be noted that the mask generator does not itself strip or mask bits off. The mask generator simply outputs a pattern of 1s and 0s that can be used by the PP's ALU to mask data. The ALU can use the mask generator's output in combination with other hardware functions to create a large number of different operations. As shown in Section 4.6, *Combining Two or More ALU Data Path Functions*, the mask generator can be used in conjunction with the barrel rotator and the three-input ALU to perform various shifts (left or right shift with zero or sign extension), bit-field extraction, and other masking operations.

Example 4–2. Mask Generator Functions

```
%3 = 0x00000007       (for 32-bit ALU segment)
%3 = 0x00070007       (for 2 16-bit ALU segments)
%3 = 0x07070707       (for 4 8-bit ALU segments)
```

4.2.3 Using the Expand Operator and the mf Register

The PP has a hardware expander that can replicate 1, 2, or 4 bits 32, 16, or 8 times, respectively, to fill out a 32-bit word. The replicated bits are packed together into 32-bit words as shown in Example 4–3.

The expander can be used as a part of many different operations. The two most common uses are to translate from one-bit to pixel-size quantities or to use the expanded value as a control mask for the operations performed by the ALU. When acting as a control mask for multiple arithmetic (split-ALU) operations, the expand amount will usually correspond to the split amount for the ALU.

The multiple flags register (mf) is the only register that connects to the input of the PP's expander. The output of the expander is an input to the C port of the ALU. The assembler uses the symbol @mf to specify sending the expanded mf register to the C port of the ALU. The amount of expansion is controlled by the Msize field in the sr register.

In addition to connecting to the expansion hardware, the mf register has rotation and clear options associated with it that help in operating on multiple byte, halfword, or word quantities. The mf can be optionally rotated by the number of bits used by the expander or set by the ALU to facilitate processing up to 32 bits through the expander for each load of the mf register. The register can be cleared before bits are set when a split-ALU operation is specified. The mf register also can have 1, 2, or 4 bits set according to split-ALU results as will be discussed later.

Example 4–3. Expand Functions

```
Binary    1  expands 1–to–32 to   11111111111111111111111111111111
Binary   01  expands 1–to–16 to   00000000000000001111111111111111
Binary   10  eXpands 1–to–16 to   11111111111111110000000000000000
Binary 1010  expands 1–to–8  to   11111111000000001111111100000000
Binary 1101  expands 1–to–8  to   11111111111111110000000011111111
```

4.2.4 Using Bit-Detection Logic

A dedicated set of hardware detects the leftmost one (lmo()), rightmost one (rmo()), leftmost bit change (lmbc()), or rightmost bit change (rmbc()) within a 32-bit word. The result is 31 minus the bit position (that is, the count starting from the MSB) of the bit detected.

- The leftmost one (lmo()) is defined as the MSB in the source register that is not zero. Similarly, the rightmost one (rmo()) is the LSB in the source register that is not zero.
- The leftmost bit change (lmbc()) is the MSB that is not the same as bit 31. Likewise, the rightmost bit change (rmbc()) is the LSB that is not the same as bit 0.

If no bit is detected, then the result is set to 32 (100000b) and the overflow status bit (V) is set. The output of the bit detection logic is fed to the C port of the ALU; this allows the result to be adjusted or compared.

Whenever an ALU operation involves the bit detection logic, the ALU is split at 8-bit boundaries. For details on the operations supported by the bit-detection logic, refer to subsection 8.3.4.2, *The Bit-Detection Modifiers*.

4.3 Using the Three-Input ALU

The PP's 32-bit three-input ALU can perform in one pass an operation that could take several passes through a two-input ALU normally used in other processors. The functions commonly available on processors with two-input ALUs are a subset of the PP's three-input ALU set of functions.

The ALU's three 32-bit inputs are referred to as the A port, B port, and C port; the inputs to the A, B, and C ports are described as the A input, B input, and C input, respectively. Because different hardware functions precede each input and the ALU can combine those inputs in numerous ways, a large number of different operations can be performed.

The three-input ALU has a control bit that selects either an arithmetic or Boolean function. For Boolean functions, no carries are generated out of or between bits of the ALU; thus, the 32-bit ALU combines the three inputs to each bit of the ALU, independently generating 32 individual bitwise logical results.

The ALU's three-input Boolean capability, combined with other features of the ALU data path, supports better bit-field processing and pixel manipulation. The ALU also improves the flexibility of the PP's overall data path by being able to selectively combine results from other data path hardware.

In arithmetic mode, the ALU can perform arithmetic operations on two Boolean combinations of the ALU inputs in a single cycle. For example, the A input ANDed with the C input, can be added to the B input ANDed with the C input. For this function, (A&C) + (B&C), the C input is used to mask the other two inputs prior to the add.

For arithmetic functions, each bit location can have a carry-in input from the adjacent LSB and can cause a carry-out to the next MSB location. In arithmetic mode, the carry path of the ALU can be split so that the 32-bit ALU behaves like two parallel 16-bit ALUs or four parallel 8-bit ALUs.

The term **multiple arithmetic** refers to split-ALU operations coupled with status setting in the mf register according to the carry-out, zero-compare, sign, or extended-sign for each split-ALU segment.

4.3.1 Understanding the Base Set and Extended Set of Operations

The three-input ALU is controlled by an 8-bit function code and an arithmetic or Boolean mode select signal. This allows the ALU to support 256 (2^8) arithmetic and 256 Boolean functions. The PP's assembler selects the appropriate ALU control signals based on the algebraically specified operation.

Although the ALU supports 256 different arithmetic functions, only a subset of these are needed for most programming. Some of the desired operations, such as sign-extended shifts and absolute value, require conditional carry-in or function control based on sign. The most generally useful set of arithmetic functions, carry-in, and sign-extension functions can be specified directly in the instruction word in what is referred to as the base set of operations or base set ALUs. This base set includes a rich set of functions that combine all three inputs. Base set ALUs associate the carry-in and, in some instances, selection between two different sign-based functions directly with the specified function code.

All 256 arithmetic functions along with more explicit carry-in and sign-extension control are available via the extended set of ALU operations (EALUs). EALU operations specify the ALU function, carry-in, and conditional sign control in D register d0 instead of directly in the opcode. Generally, you should use EALU operations only when you are coding speed-critical loops.

4.3.2 Setting the ALU Status Register

Generally, four result status bits (negative, carry, overflow, and zero) are set in the PP status register as a result of each ALU operation. Status bits have different meanings with multiple arithmetic, bit detection, and divide iteration operations.

Normally, all four status bits are set based on the result of each ALU operation, but there are cases in which some or all of the bits are not modified.

- The status bits are not updated when the destination of the ALU operation is the pc register. This feature allows a series of conditional branches on the same status register value.
- If the instruction makes the saving of the ALU result conditional, the status bits are set only if the result is written to the destination (that is, the condition is true).
- Additionally, the instruction set supports the protection of individual status bits from being modified by a given instruction.

The status bits are useful for performing conditional operations based on previous ALU results. For more details on setting ALU status, refer to the description of the status register. Conditional operations and/or status protection are described in Chapter 8, *The PP Assembly Language Instruction Set*.

4.3.3 Using Natural Arithmetic Functions

The next two sections describe the ALU functions that are controlled by an 8-bit function code and an arithmetic or Boolean mode select bit (as described in subsection 4.3.1, *Understanding the Base Set and Extended Set of Operations*). These functions are referred to as the natural functions. As discussed in subsection 4.3.5, *Using Sign-Extend and Absolute Value Functions*, additional functions such as shift right with sign extension are performed by choosing between two different natural functions, according to input-sign information.

4.3.3.1 Using Three-Input Boolean Functions

In Boolean (bitwise logical) mode, the three-input ALU is capable of performing 256 functions that combine the three inputs to each bit of the 32-bit ALU, independently generating 32 bitwise-logical results. The 256 functions are controlled by an 8-bit ALU function code.

The PP assembler contains a Boolean expression evaluator that allows a selected Boolean function to be expressed in any order and/or any equivalent form. For example, (A&B)|(A&C), A&(B|C), and (B|C)&A are equivalent expressions; thus, they are reduced to the same ALU function code by the assembler.

Note that although the ALU is capable of combining all three inputs, many functions don't involve some of the inputs. For example, the expression A&B ignores the C input; the expression A|C ignores the B input. The ability to use or ignore any of the ALU inputs essentially allows you to choose to use or ignore the corresponding hardware along the data path into the ALU.

The most common uses of three-input functions involve masking and merging. Often, one input acts as a mask that selects between the other two inputs or can mask either one or both of the inputs.

The following examples show three-input functions in which the C input is treated as a mask:

- **(A&C)|(B&~C)**

  Multiplex (merge) A and B bit by bit based on C. (A is chosen if the corresponding bit in C is 1; B is chosen otherwise.)

- **(A&~C)|(B&C)**

  Multiplex (merge) A and B bit by bit based on C. (B is chosen if the corresponding bit in C is 1; A is chosen otherwise.)

- **(A|B)&~C**

  Bitwise OR A and B and then force the result to 0 everywhere that C is a 1.

- **(A&B)&~C**

  Bitwise AND A and B and then force the result to 0 everywhere that C is a 1.

- **A|(B&C)**

  If C is 0, force the B input to 0 before bitwise ORing with A.

- **A|(B|~C)**

  If C is 0, force the B input to 1 before bitwise ORing with A.

While all of the examples above treat the C port as the mask or merging control, the ALU itself is fully symmetrical in terms of Boolean functions between A, B, and C. However, other hardware features of the PP's data path facilitate the generation of masks that feed the C port of the ALU. For example, the C input could be a mask of replicated 1s or 0s, each the width of a byte or half-word, created by the expander; this allows multiplexing or masking of bytes or halfwords.

4.3.3.2 Using Carry-In and Carry-Out Generation

Before discussing the three-input ALU arithmetic functions, it may be helpful to review the basics of the binary number system and 2s-complement arithmetic. These concepts are described briefly in Appendix C, *Understanding the Binary Number System*. The following paragraphs explain briefly the differences in carry-in for addition and subtraction.

For **addition**, the carry-in function is straightforward. A carry-in of 0 causes two addends to be added together ($x + y$); a carry-in of 1 causes an additional 1 to be added to the sum ($x + y + 1$). For extended precision arithmetic—for example, add with carry—the carry-out from the lower order addition is used as the carry-in for the higher order addition.

The ALU is explained below in terms of always performing an addition; subtraction is equivalent to adding a negative number. The following equalities are used throughout this discussion:

- The 1s complement of a number is equal to a logical inversion of that number (~x).
- The 2s complement of a number is equal to the arithmetic negative of that number (–x).
- The 1s complement of a number is equal to the 2s complement of that number minus 1 (~x = –x – 1).
- Thus, the 2s complement of a number can be generated by inverting the number and adding 1 (–x = –x–1+1 = ~x + 1).

Using the above equalities, the ALU negates a number by logically inverting (or 1s complementing) the number and injecting a carry-in of 1 into the LSB. The ALU performs subtraction (minuend minus subtrahend) by adding the minuend to the 1s complement (logical inversion) of the subtrahend and injecting a carry into the LSB of the addition.

When a **subtraction** is performed, the natural function of the ALU is to generate a carry-out of 0 for a borrow and a carry-out of 1 for a not-borrow ($\overline{\text{borrow}}$). The PP sets the carry-out status bit on the basis of the natural sense of the carry-out. Note that most other processors invert the natural sense of carry when setting the status register carry bit for subtractions; then they invert the carry bit in the status register when performing extended-precision subtractions. The PP does not perform either of these inversions.

In the base set of operations, carry-in is 0 for addition and 1 for subtraction. The PP's instruction decoder automatically associates the carry-in correctly for addition or subtraction in the case of 32-bit arithmetic.

The instruction set also supports carry-in based on the status register's carry-out bit; this makes extended-precision add-with-carry or subtract-with-borrow operations possible. For extended-precision subtraction, the absence of a carry-in to the LSB of the higher order addition is equivalent to borrowing 1 from the computation; thus, for subtraction, carry-in means not-borrow ($\overline{\text{borrow}}$).

As explained in more detail in subsection 4.6.9, *Using Multiple Arithmetic With Conditional Function Control*, some of the ALU functions allow the C input to control whether the B input is added or subtracted from the A input. Because these ALU functions can combine with a split ALU, the C input can be used to control whether each segment of the ALU performs an add or subtract.

Accordingly, in the base set of operations, the carry-in to each split of the ALU is controlled so that a carry-in of 0 is injected if addition is being performed and a carry-in of 1 is injected if subtraction is being performed.

4.3.4 Using Three-Input Arithmetic Functions

The three-input ALU is capable of 256 different mixed Boolean and arithmetic functions in a single pass. These functions support performing a Boolean function prior to an arithmetic function. For brevity, the mixed Boolean and arithmetic functions are referred to simply as arithmetic functions. Various compound functions such as shift and add (or subtract) as well as field masking prior to addition or subtraction can be performed by the appropriate ALU function in combination with other ALU data path hardware.

Table 4–1 lists many of the more useful arithmetic functions and their corresponding ALU function code (in hex). Note that Table 4–1 gives the natural function of the ALU, assuming a carry-in of 0 for addition and a carry-in of 1 (injected into each split of the ALU) for subtractions. As discussed in subsection 4.3.5, additional functions such as shift right with sign extension are implemented by choosing between two different natural functions, according to input-sign information.

For most algorithms, the subset of arithmetic operations listed in Table 4–1 is more than adequate. The remainder of this section includes information on the full range of arithmetic functions supported by the PP's three-input ALU.

A description of how to derive the ALU function code for a particular operation is included solely to aid in the understanding of the ALU; the PP's algebraic assembler automatically determines the function code from each algebraic description.

Table 4–1. Some Useful Three-Input ALU Arithmetic Functions

ALU Code	Function	Default Carry-In	Common Use
66	A + B	0	Add A to B and ignore C.
99	A – B	1	Subtract B from A and ignore C.
5A	A + C	0	Add A to C and ignore B.
A5	A – C	1	Subtract C from A and ignore B
6A	A + (B&C)	0	Shift right and add (C acts as shift mask).
95	A – (B&C)	1	Shift right and sub (C acts as shift mask).
56	A + (B|C)	0	Force bits in B to 1 and add to A.
A9	A – (B|C)	1	Force bits in B to 1 and sub from A.
A6	A + (B&~C)	0	Shift left and add (C acts as shift mask).
59	A – (B&~C)	1	Shift left and sub (C acts as shift mask).
65	A + (B|~C)	0	Signed right shift and add.†
9A	A – (B|~C)	1	Signed right shift and sub.†
60	(A&C) + (B&C)	0	Mask A and B by C and then add.
96	A + ((–B&C) | (B&~C))	LSB of C	Add B or subtract B based on ~C.
69	A + ((B&C) | (–B&~C))	LSB of ~C	Add B or subtract B based on C.
AA	A	0	Return A and ignore B and C.
55	–A	1	Negative A and ignore B and C.
CC	B	0	Return B and ignore A and C.
33	–B	1	Negative B and ignore A and C.
F0	C	0	Return C and ignore A and B.
0F	–C	1	Negative C and ignore A and B.
C0	(B&C)	0	Shift right (C acts as shift mask).
3F	– (B&C)	1	Shift right (C acts as shift mask).
FC	(B | C)	0	Force bits in B to 1.
03	– (B | C)	1	Force bits in B to 1 and negate.
0C	(B&~C)	0	Shift left (C acts as shift mask).
F3	–(B&~C)	1	Shift left and negate.
CF	(B | ~C)	0	Signed right shift.†
30	–(B | ~C)	1	Signed right shift.†
3C	(–B&C) | (B&~C)	LSB of C	Return B or –B based on ~C.
C3	(B&C) | (–B&~C)	LSB of ~C	Return B or –B based on C

† The Boolean operation B|~C forces B to 1 everywhere that C is 0; this can be used to one-extend right-rotated numbers, but it is most often used to perform sign extension of negative numbers by operations that are controlled on the basis of the sign bit going to the barrel rotator (see sections on sign extension below).

4.3.4.1 Function Code Considerations

The following observations about the ALU function codes shown in Table 4–1 help explain ALU and EALU operations.

- For the base set of operations, the carry-in is equal to the LSB of the ALU function code (with an exception for the cases in which the C input controls addition or subtraction between A and B).

- Inverting all the ALU function code bits changes the sign of the operation. For example, function codes 0x66 (A+B) and 0x99 (A–B) are bitwise inverses; similarly, 0x65 (A+(B|~C)) and 0x9A (A–(B|~C)) are bitwise inverses. MPY||EALU operations support either ealu(), which performs the ALU function specified in d0, or ealuf(), which performs the inverse of the ALU function specified in d0. This allows both addition and subtraction to be performed using the same d0 setup as discussed in subsection 8.5.2.4, *ealu and ealuf*.

- Conditional inversion of the four LSBs of the ALU function code based on the sign bit of the input to the barrel rotator can be used to support sign-extended shift right. For example, if the sign of the input to the barrel rotator is not negative, the four LSBs of the function A + (B|~C) (65) are automatically inverted to form the function A + (B & C) (6A).

  - When the sign bit of the barrel rotator input is 1 (the number is negative), the four LSBs of the function are used as is, yielding the function A + (B|~C). The (B|~C) combination ORs in a one everywhere that the C input is 0. This can be used to sign-extend negative numbers before the addition is performed.

  - When the sign bit of the barrel rotator input is 0 (the number is positive), the four LSBs of the function A + (B|~C) (65) are inverted to form A + (B & C) (6A). The (B&C) combination zeros out bits in B where the C input is 0. This can be used to zero-extend positive numbers before addition is performed.

  Sign-extended shift right is described in more detail in subsection 4.6.1.7, *Signed (Arithmetic) Shift Right: >> or >>s*.

4.3.4.2 Arithmetic Function Equation

The following general equation describes all of the 256 arithmetic functions supported by the PP's three-input ALU:

output = A & f1(B,C) + f2(B,C) [+ 1]

f1(B,C) and f2(B,C), referred to as subfunctions, can be any of the 16 possible functions of B and C shown in Table 4–2. f1(B,C) and f2(B,C) are independent of each other (they can be, and usually are, different). The [+1] term indicates the appropriate injection of a carry-in on the basis of addition/subtraction or addition-with-carry/subtraction-with-borrow.

The subfunctions include negative terms of B and/or C so that when a negative number is added, subtraction is performed.

Note from the equation above that the A input cannot be negated in an arithmetic expression. Either B or C can be negated/inverted because the f1 and f2 subfunctions support negation/inversion.

Table 4–2. The 16 Possible Functions of B and C (Assuming Carry-In = 0)

f1 Code	f2 Code	Subfunction	Common Use
00	00	0	Zeros one of the terms.
AA	FF	All 1s = –1	All 1s or –1.
88	CC	B	Return B
22	33	–B–1	Negate B.
A0	F0	C	Return C
0A	0F	–C–1	Negate C.
80	C0	B&C	Force bits in B to 0 where C is 0.
2A	3F	–(B&C) –1	Force bits in B to 0 where C is 0 and negate.
A8	FC	B | C	Force bits in B to 1 where C is 1.
02	03	–(B | C) –1	Force bits in B to 1 where C is 1 and negate.
08	0C	B&~C	Force bits in B to 0 where C is 1.
A2	F3	–(B&~C) –1	Force bits in B to 0 where C is 1 and negate.
8A	CF	B | ~C	Force bits in B to 1 where C is 0.
20	30	–(B | ~C) –1	Force bits in B to 1 where C is 0 and negate.
28	3C	(B&~C) | ((–B–1) &C)	Choose B if C = all 0s and –B if C = all 1s
82	C3	(B&C) | ((–B–1) &~C)	Choose B if C = all 1s and –B if C = all 0s.

4.3.4.3 Representing and Factoring Functions

The subfunctions given in Table 4–2 are the most generally useful ways to express each of the various functions of B and C; there are other ways to represent or factor each function.

For example, if you apply DeMorgan's Law, the function B|~C is equivalent to ~(~B&C). Because ~X = –X–1, ~(~B&C) is equivalent to –(~B&C)–1, and B|~C is equivalent to B|(–C–1). Similar conversions can be made on many of the functions in Table 4–1 and Table 4–2.

Note:

The negative forms in Table 4–2 have a trailing –1 term because the ALU performs a logical inverse, which forms the 1s complement. Generally, a carry-in of 1 will be injected into the LSB of the ALU (or each split) in order to cancel out the –1 and finish the 2s complement. The optional +1 in the equation is used to specify a carry-in of 1.

Often, it is convenient to think of the Boolean functions in Table 4–2 as performing a masking operation. The function B&C can be interpreted as forcing the B input value to zero where the corresponding bit in C is a 0. The function B|~C can be interpreted as forcing the B input value to 1 everywhere the C input is 0.

Because the C port of the ALU is fed by both the mask generator and expander (via a multiplexer), the C port is usually used as a mask in functions that involve both B and C terms. Table 4–2 factors each function accordingly.

Table 4–2 includes codes for the subfunctions f1 and f2 that can be used to derive the ALU function code. Note that the same function of B and C has different codes depending on whether it plugs into the f1 or f2 part of the three-input ALU arithmetic equation.

The ALU function code can be derived by bitwise XORing the f1 and f2 codes for the subfunctions plugged into the three-input ALU equation. The contents of the table can be plugged into the equation to get any of the 256 arithmetic functions that the ALU can perform.

Occasionally, you can reduce an expression further and still keep it equivalent to the original. Some important considerations are listed below:

- The default for base set operations is to have a carry-in of 0 for addition and a carry-in of 1 when f2(B,C) has a negative B or C term factored as in Table 4–2. This carry-in injection has the effect of turning the 1s complement (logical inversion) into 2s complement.

  Thus with a carry-in of 1 injected into the functions in Table 4–2 with a preceding minus sign, the –1 on the right-hand side of these functions is effectively canceled out. Note that this canceling process does inject a carry that can ripple (for example, 0 – 0 is actually performed by 0+(–0–1)+1, which results in 0 with a carry-out of 1).

- The bitwise AND of A and all 1s equals A. It is common to specify f1(B,C) as all 1s in order to place A on the left side of the equation.

- All 1s is the 2s complement form of minus 1 (–1).

4.3.4.4 Deriving ALU Functions

This section shows how you can use the general arithmetic function equation and Table 4–2 to derive any of the possible ALU functions and their corresponding ALU functions codes. Note that for most uses, the assembler performs all of these transformations and generates the ALU function code.

The following paragraphs explain some of the ALU arithmetic functions (as shown in Table 4–1). For information on operator precedence, refer to subsection 8.2.1, *Operator Precedence*.

- **A + B**

  This can be arrived at by specifying that f1(B,C) = all 1s and f2(B,C) = B. Thus, A&(1s) + B = A + B. The f1 code for all 1s is AA, and the f2 code for B is CC. Bitwise XORing AA and CC gives 66, which is the ALU function code for A + B. Note in this ALU function, the C input has no effect.

- **A–B**

  To derive A–B, use f1(B,C) = all 1s (f1 code AA) and f2(B,C) = –B–1 (f2 code 33) in the arithmetic function equation. This results in the function A&(1s) + (–B–1). Injecting a carry-in of 1 and simplifying A&(1s) to A results in A – B. Bitwise XORing the f1 code AA with the f2 code 33 gives 99, which is the ALU function code for A–B, assuming a carry-in of 1.

- **A+C**

  A+C can be derived like A+B: by choosing f1(B,C) = all 1s and f2(B,C) = C so that the result is A& (all 1s) + C = A+C. With this function the B input has no effect.

- **A–C**

  Choosing f1(B,C) = all 1s and f2(B,C)= –C–1 yields A–C if a carry-in of 1 is injected into the LSB of the ALU.

A major benefit of the three-input ALU is the masking and/or conditional functions between two of the inputs on the basis of the third input. The data path of the PP has been set up so that the C port will generally be most useful as a mask (often using the mask generator) or conditional control input (often using the expander).

Remember when looking at the following examples that the ALU always performs the Boolean function before addition (or net subtraction). Therefore, even though bits may be cleared by the Boolean function, a carry could ripple out of the unmasked bits into a bit or bits that were zeroed or set by the Boolean function.

- **A + (B&C)**

  Choosing f1(B,C) = all 1s (f1 code of AA) and f2(B,C) = B&C (f2 code of C0) yields A+(B&C). Bitwise XORing AA and C0 gives the ALU function code 6A.

  This function can be used to strip off bits from unsigned numbers or as part of sign-extending 0s for positive signed numbers. This function, when used with barrel rotation and the mask generator, can perform unsigned shift right and add operations.

  You can also use this function to conditionally add B to A. For example, if the C port is fed by the output of the expander, the addition of B to A is controlled by the bits in the mf register. If the ALU is split, then corresponding bits in the mf register control each split of the ALU.

- **A+(B|~C)**

  Choosing f1(B,C) = all 1s (f1 code of AA) and f2(B,C) = B|~C (f2 code of CF) gives A+(B|~C). Bitwise XORing AA and CF gives the ALU function code of 65. This function can be used to force bits in B to 1, before the addition, everywhere that C is 0.

  This function is most often used as a part of sign extension. The C input acts as a mask for setting sign bits. You can use hardware controlled by the sign bit of B to choose this operation if the sign is 1, or to choose A+(B&C) if the sign bit is 0.

- **A – (B&C)**

  Choosing f1(B,C) = all 1s and f2(B,C) = –(B&C)–1 and injecting a carry-in of 1 gives A – (B&C). The ALU function code is given by XORing the f1 code of AA with the f2 code of 3F to get the code of 95.

  Like A + (B&C), the C input of this function can be used to strip off or mask bits in the B input before the subtraction from A.

When f1(B,C) = 0, then the ALU function is the f2(B,C) subfunction. These functions are commonly used in conjunction with other hardware to perform negation, absolute value, bit masking, and/or sign extension of the B input by the C input.

- **–(B&C)**

  Choosing f1(B,C) = 0, choosing f2(B,C) = –(B&C)–1, and injecting a carry-in of 1 results in –(B&C).

  This function uses the C input to mask bits in B and then negates the quantity. This function can be used as part of a shift right and negate operation.

The examples below show some functions that support masking both sides of the plus sign in the three-input ALU equation.

- **(A&C) + (B&C)**

  Choosing f1(B,C) = C and f2(B,C) = B&C results in this function. This function, before the addition, effectively zeroes the corresponding bits of the A and B inputs where C is a 0. Note that the Boolean function is applied before the addition and that carry(s) can ripple into the bits that have been zeroed unless the ALU is split at that point.

  You can use this function to sum together multiple smaller quantities that are in one register. The B port gets a rotated version of the number going to the A port, and the C port masks off bits that overlap. In this manner, four bytes can be summed into two halfwords, or two halfwords can be summed into a one-word result.

As described in Section 8.4, *The Extended Set of ALU Operations (EALU)*, other functions are supported that are not shown in Table 4–1 using EALU operations. The following are examples of functions available with EALU:

- **(A&C)–(B&C)**

  Choosing f1(B,C) = C, choosing f2(B,C) = –(B&C)–1, and injecting a carry-in of 1 results in the function (A&C)–(B&C). This function can be used to produce negative sums with the C port value acting as a mask of the A and B inputs.

- **(A&B)+B**

  Choosing f1(B,C) = B and f2(B,C) = B results in the function (A&B)+B. This function can be used to conditionally double B depending on whether A is all 1s or all 0s.

4.3.5 Using Sign-Extend and Absolute Value Functions

In order to perform sign-extended shifts or absolute value, the hardware must select between two different natural functions on the basis of whether the input is positive or negative.

The PP has hardware that tests the MSB (sign bit) of the input to the barrel rotator to determine whether to invert the four LSBs of the ALU function code. This hardware can produce the net effect of sign-extending or taking the absolute value of the B input.

The PP's algebraic assembler recognizes the shift right symbol (>>) (signed shift right is the default) and absolute value symbol (|*value*|) and translates them into the appropriate data path and ALU control combinations.

4.3.6 Using Multiple Arithmetic Functions

The PP's ALU can be split (referred to as multiple arithmetic) into one 32-bit ALU, two 16-bit ALUs, or four 8-bit ALUs. When multiple arithmetic is performed, one bit of status information per split ALU segment can be set in the mf register. For examples of this, see subsections 4.6.5 through 4.6.9.

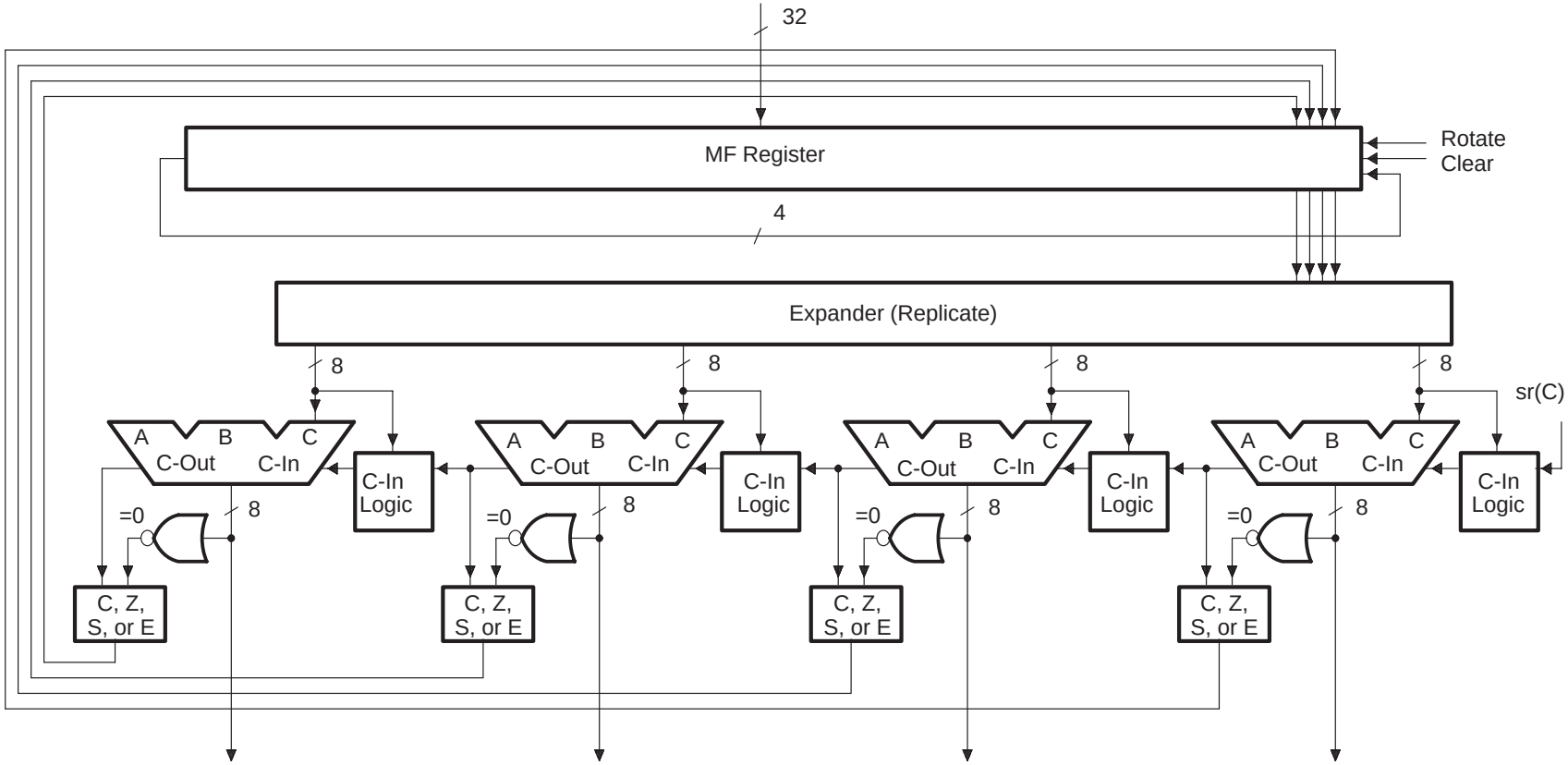
The bits in the mf register are optionally set according to a status result of each split of the ALU. In the case of 32-bit multiple arithmetic, the ALU is not split but can set one mf register bit. Thus 1, 2, or 4 bits can be set in the mf register with multiple arithmetic, as desired.

Multiple arithmetic is specified directly in the instruction by an ALU function modifier (refer to subsection 8.3.4, *ALU Function Modifiers*, for more information on function modifiers). Thus, the ALU can switch between single 32-bit arithmetic and multiple arithmetic on an instruction-by-instruction basis. The size of the split ALU for multiple arithmetic is configured by the Asize field in the sr register (refer to page PP: 7-36 for a detailed description of the sr register).

Figure 4–2 diagrams how data flows in multiple arithmetic operation for the case of four 8-bit splits of the ALU (referred to as multiple-byte arithmetic). In multiple arithmetic, the ALU effectively operates as independent parallel ALUs where each ALU receives the same function code.

There is logic for each split of the ALU to control the carry-in. Each split in the ALU has zero detection, and bits in the mf can be set according to this zero detection, the sign bit, the extended sign bit, or carry out. The extended sign bit acts as an extra bit of precision on signed add or subtract operations. Often, in multiple arithmetic, the C input of the ALU is driven by the output of the expander, as shown in Figure 4–2, and the C port of each split of the ALU gets a replicated copy of one bit in the mf register.

Figure 4–2. Multiple-Byte Arithmetic Data Flow



Notes: 1) C-In stands for carry-in
2) C-Out stands for carry-out
3) Z stands for zero detect
4) S stands for sign of the result
5) E stands for extended sign of the result

For multiple arithmetic, certain ALU functions allow the input to the C port to control how A and B are combined. For example, you can use the ALU function A + ((B&C)|(–B&~C)) to add B to or subtract B from A, as determined by C. This is useful when a prior multiple arithmetic operation has set bits in the mf register; each split of the ALU can be conditionally controlled by the prior status results using @mf.

The mf register has special control and rotating functions that facilitate consuming successive bits from the mf register or agglomerating bits into the mf register. The register can be cleared, or it can be rotated left by the number of bits to be set before it is written with status. Note that unless the register is cleared or rotated, only the bits that are set by the split-ALU status are modified in the mf; this supports the selective accumulation of results by successive operations.

The ALU's ability to set one bit for each split quantity can be used to perform n-bit to 1-bit conversions and to maintain precision when split-ALU quantities roll over their split size.

4.4 Using Data Path Multiplexers

As shown in Figure 4–1, the ALU data path has three multiplexers: the rotate amount multiplexer, the mask generator input multiplexer, and the ALU C port input multiplexer. These multiplexers are controlled by the instruction so that the data path required by a given operation is selected.

Note that the ALU can functionally ignore any of its inputs. The ALU's ability to select inputs combines with the multiplexer selections to give a large number of data path options.

❑ Rotate Amount Multiplexer

The rotate amount multiplexer chooses between a D register that is specified explicitly in the opcode, d0, 5-bit immediate data from the opcode, and the constant 0. Only the five LSBs are sent out of the rotate amount multiplexer to control the rotation performed by the barrel rotator.

❑ Mask Generator Multiplexer

The mask generator input multiplexer chooses between opcode-specified source registers, immediate data, and d0. The five LSBs of the selected source are sent out of the multiplexer to the mask generator.

❑ C Port Multiplexer

The ALU's C port multiplexer selects between inputs to the C port. These inputs include source 2 (src2), source 4 (src4), the output from the bit-detection logic (LMO, RMO, LMBC, RMBC), the mask generator, and the expander. The C port multiplexer always sends 32 bits of data to the ALU's C port.

4.5 Understanding ALU Data Path Classes

The multiplexers support many different ALU data path combinations (see Section 4.4). For the base set of ALU operations, a 3-bit code in the instruction, referred to as the **operation class**, specifies one of eight supported combinations of multiplexer settings. Figure 4–3 shows a summary of the ALU data path for each of the eight base set ALU operation classes.

Each figure leaves out any hardware that is not used in that class. Because of the flexibility offered by the operation classes, the same ALU function can perform different ALU data path operations with different classes. The effect of the operation classes is essentially to allow eight different ALU data path configurations to be used by the same ALU function. The eight operation classes and the large number of base set ALU functions combine to support over 2000 different single-cycle ALU data path operations.

The operation class also determines the connections of the source operands to the various hardware units. For example, for operation class 1, src1 is used as the input to the barrel rotator; for operation class 7, src1 is used as the input to the A port of the ALU.

Note that although each operation class has some connection to all three ALU input ports, the function codes that control the ALU can be programmed to use or ignore any of the three inputs. For example, if the ALU function is A+B, then the input to the C port is ignored.

Each operation class shown in Figure 4–3 provides a useful set of ALU operations. For example, shift operations described in subsection 4.6.1, *Using Shifts With Addition and Subtraction*, are supported by operation classes 3 and 4. Operation classes 0 and 5, which route the output of the expander to the C port of the ALU, support many different operations, including color expansion (described in subsection 4.6.4) and transparency (described in subsection 4.6.7.1).

Eight additional data path combinations are supported by the data unit instructions that use six operands. The MPY||ADD data path executes a 16×16-bit multiply (described later in Section 4.7, *Using the Multiplier Data Path*) in parallel with an add or subtract performed by the ALU. The MPY||SADD supports a 16×16 multiply in parallel with a right-shift with an add or subtract.

The MPY||EALU data path supports either one 16 × 16-bit multiply or two 8 x 8-bit multiplies performed in parallel with the extended ALU operation specified in d0. This allows any of the 256 Boolean or 256 arithmetic ALU functions to be performed in parallel with a multiply. A data path option specified by bit 13 in the d0 register supports using the expanded mf (@mf) as the ALU input to the C port.

The instruction set also supports two ALU data paths for extended ALU operations, referred to as EALU||ROTATE, that save the output of the barrel rotator to a destination register and feed it to the B port input of the ALU. The EALU||ROTATE data paths provide flexibility by allowing six operands for the ALU operation to be specified in the instruction. A data path option specified by bit 13 in the d0 register supports using the expanded mf (@mf) as the ALU input to the C port.

As demonstrated in subsection 4.6.3, the EALU||ROTATE (1) data path shown in Figure 4–4 can support an arbitrary field extract from a register.

In addition to the data path combinations shown in Figure 4–4, the ALU data path supports unsigned division through an iterative approach (one cycle per bit of precision) specified by a six-operand opcode format with conditional execution. The divide iteration (divi) operation is detailed in Section 8.6, *The Divide Operation*.

The different data path classes enable the ALU to perform a wide range of powerful operations in a single cycle; essentially, the ALU has a cycle-by-cycle configurable data path. As you gain experience in programming the device, you will become familiar with what can and cannot be done in a single ALU data path operation. Since the assembler handles the actual breakdown of the operation class, you don't have to consider what the class is for the desired operation.

Figure 4–3. Base Set ALU Operation Classes

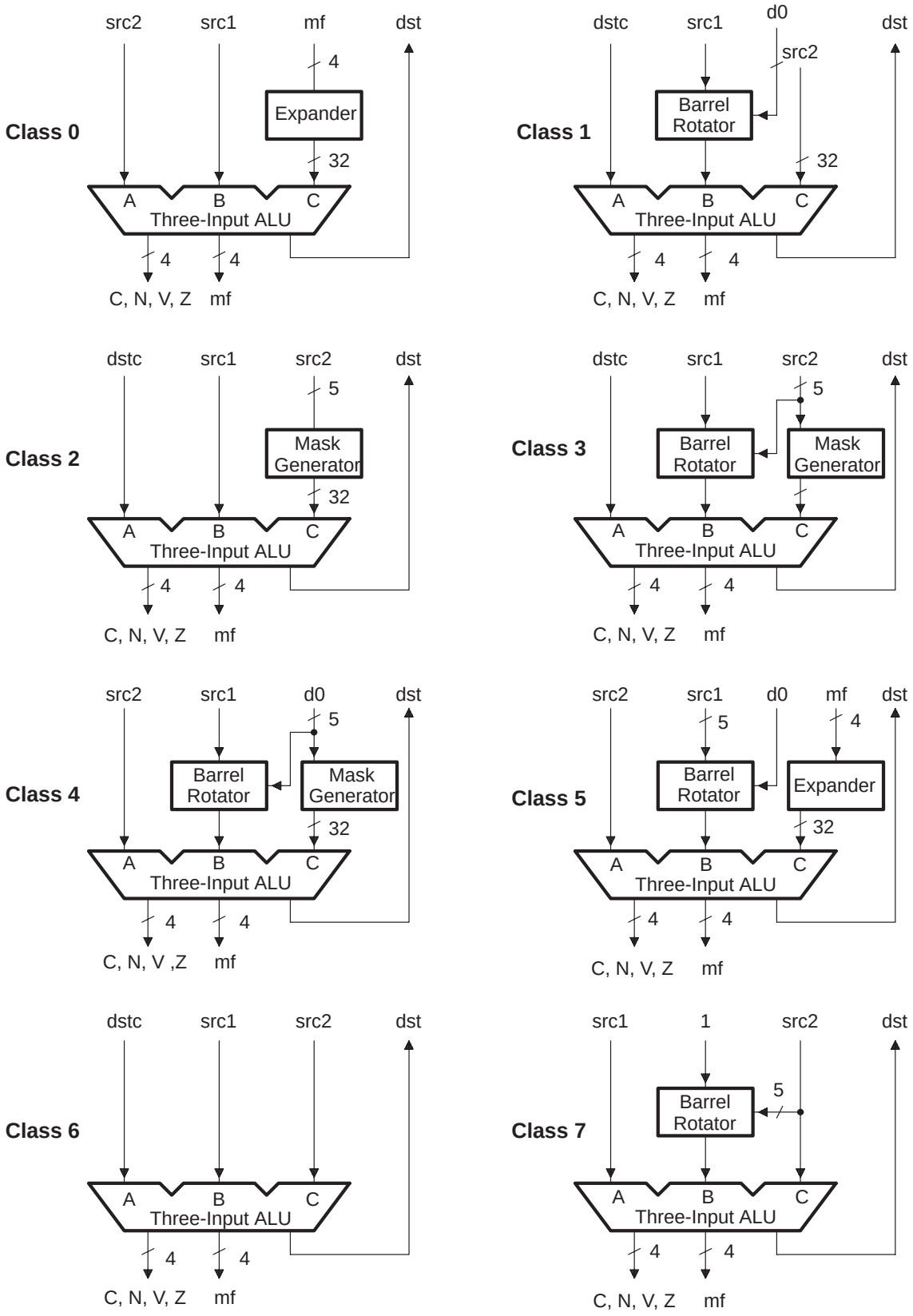


Figure 4–4. Six-Operand Data Path Classes

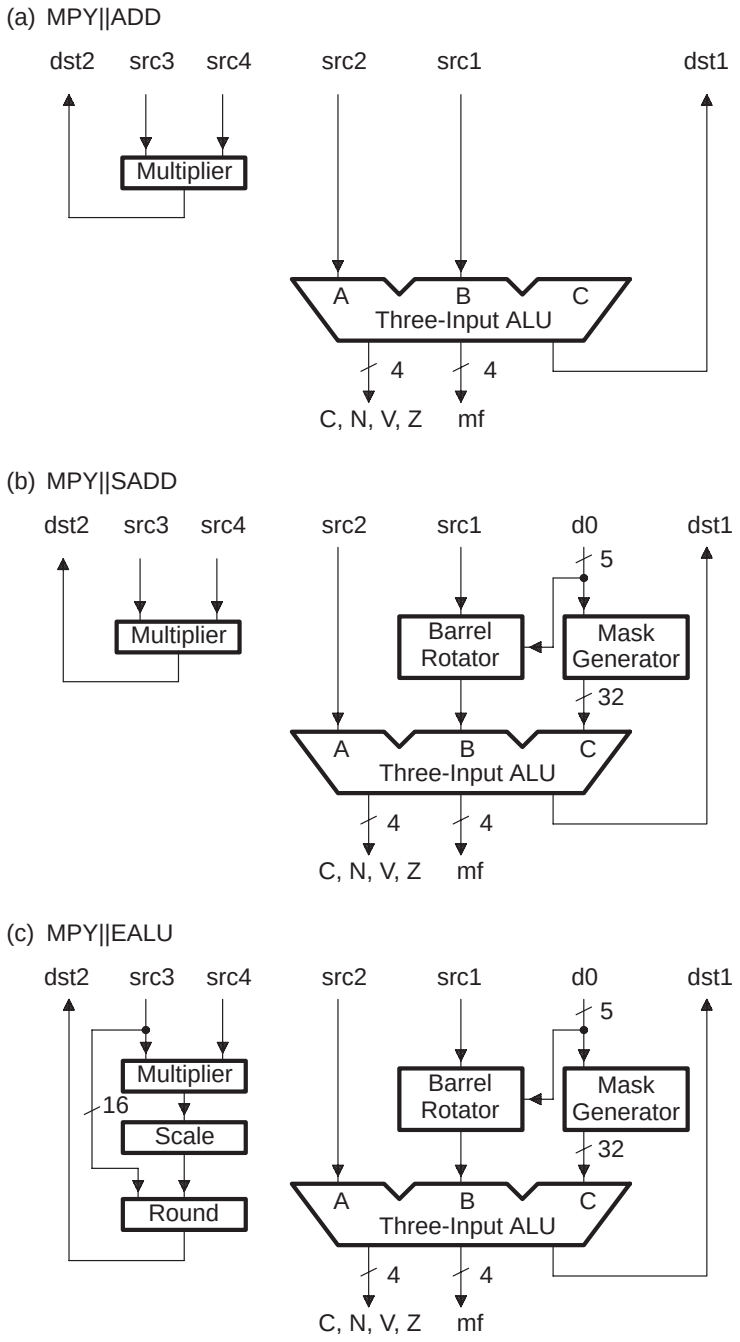
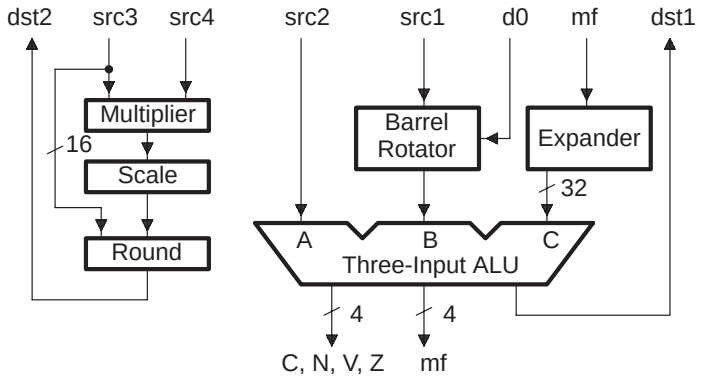
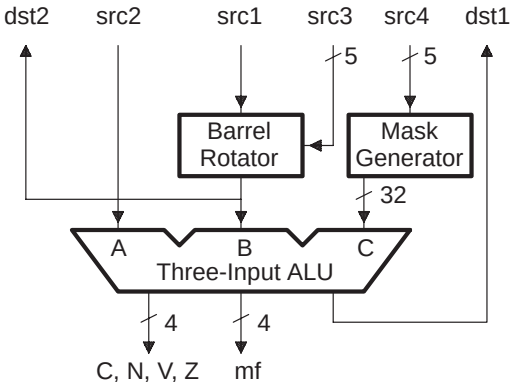


Figure 4–4. Six-Operand Data Path Classes (Continued)

(d) MPY||EALU with @mf



(e) EALU||ROTATE (1)



(f) EALU||ROTATE (2)

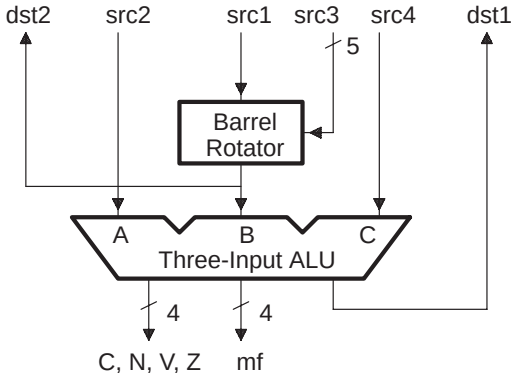
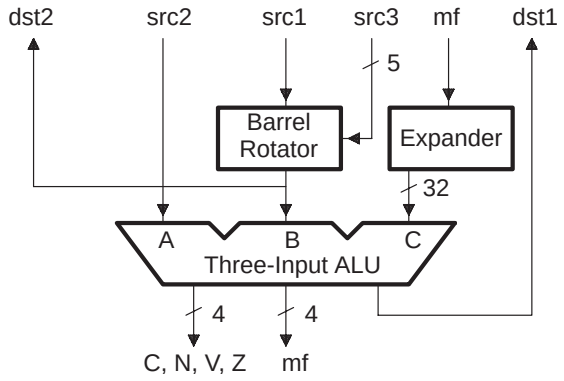


Figure 4–4. Six-Operand Data Path Classes (Continued)

(g) EALU with @mf||ROTATE



4.6 Combining Two or More ALU Data Path Functions

The previous sections described the individual workings of each functional block in the ALU data path. This section demonstrates how to combine these functions to perform even more powerful operations.

Throughout this section, the symbol b (lowercase b) indicates the input to the barrel rotator. Symbols in square brackets indicate either the ALU data path input that the adjacent data quantity is fed into or the function performed by the three-input ALU. For instance, in Example 4–4, [b] indicates that the original number is used as the input to the barrel rotator. [B&~C] in the last line of the example indicates that the function B&~C is used to produce the final result by combining the B and C inputs.

4.6.1 Using Shifts With Addition and Subtraction

This subsection describes how the barrel rotator, mask generator, and three-input ALU can work together to perform shift left, unsigned (logical) shift right, and signed (arithmetic) shift right operations. These operations can be performed solely as shifts or in combination with an addition or subtraction in a single cycle.

The assembler supports the symbols >>u for unsigned shift right, >> or >>s for signed shift right, and << for a shift left. These shift notations act as macro functions that select the appropriate rotate, mask generation, and ALU functions.

The PP assembly language also allows you to explicitly specify shifts in terms of barrel rotation (\\), mask generation (%), and the ALU function. The explicit notations given in the following shift examples demonstrate the operation of the ALU's data path. When you program the PP, it is easier to write and read programs that use the shift symbols.

4.6.1.1 Shift Left: <<

A shift left operation can be performed by the PP in a single cycle. This operation can be broken down into the following elemental functions:

1) The input to the barrel rotator is rotated left by the number of bits to be shifted left. Bits that rotate out of the MSBs (the left) rotate back into the LSBs (on the right) and must be stripped off for a shift left. The rotated output from the barrel rotator is sent to the B port of the ALU.

2) The shift amount is input to the mask generator to form a mask with the number of right-justified 1s equal to the shift amount. The same amount is input to the barrel rotator control input and the mask generator input. The mask generator output is sent to the C port of the ALU.

3) The ALU uses the function B&~C to combine the barrel rotator and mask generator outputs.

The following example shows how a shift left is expressed in the PP's algebraic notation:

```
dst = b<<n
```

dst: Destination register for shift left result.

b: Register input to the barrel rotator.

n: Either a 5-bit constant or the five LSBs contained in a register.

This operation is equivalent to the explicit notation below:

```
dst = b\\n & ~%n
```

Example 4–4 demonstrates a shift left of 0x53FF FFA7 by four bits. The example shows several steps, but the operation is performed in a single pass through the ALU data path.

Example 4–4. Shift Left by Four Bits

```
Original Number          [b]  01010011111111111111111110100111  (0x53ffffa7)
Rotate Number by 4       [B]  00111111111111111111101001110101  (0x3ffffa75)
Mask                     [C]  00000000000000000000000000001111  (0x0000000f)
Shift Left Result     [B&~C]  00111111111111111111101001110000  (0x3ffffa70)
```

In the first line of Example 4–4, [b] indicates that the original number is used as the input to the barrel rotator. In the next line, [B] indicates that the barrel rotator output becomes the input to the B port of the ALU. Likewise, [C] indicates that a mask is the input to the C port of the ALU.

The ALU function B&~C is used to combine the C port and B port inputs to mask off the wrapped around bits in the rotated value; this produces the desired shifted left result. This format is followed throughout the subsequent examples in this section.

In a shift left, arithmetic overflow occurs if any of the bits shifted out or the resulting sign bit is different than the original sign bit. In the Example 4–4, notice that the shift left has overflowed (the example has data in these bits to demonstrate the masking process).

Arithmetic overflow is not automatically detected for shift lefts, but it can be detected by subtracting the leftmost bit change from the shift amount.

4.6.1.2 Shift Left and Add: A + b<<n

In PP assembly code, the operation A + b<<n is translated by the assembler into an instruction that uses the barrel rotator, mask generator, and three-input ALU function to performed the desired operation. The shift left and add works like the previous example of a simple shift left, except for the ALU's function. Instead of performing B&~C as in a simple shift, the ALU performs the function A+(B&~C).

Shift left and add is expressed in the PP's algebraic notation below:

```
dst = A + b<<n
```

dst: Destination register for shift left and add result.

A: Register input to the A port.

b: Register input to the barrel rotator.

n: Either a 5-bit constant or the five LSBs contained in a register.

This operation is equivalent to the explicit notation below:

```
dst = A + (b\\n &~%n)
```

Example 4–5 shows 0x53FF FFA7 shifted left by four bits with an add to 0x0000 00AA. While this is shown as several steps, it can all be done by the PP in a single pass through the ALU data path.

Example 4–5. Shift Left and Add

```
Original Number            [b] 01010011111111111111111110100111  (0x53ffffa7)
Rotate Number by 4         [B] 00111111111111111111101001110101  (0x3ffffa75)
Mask                       [C] 00000000000000000000000000001111  (0x0000000f)
Shift Left Result       [B&~C] 00111111111111111111101001110000  (0x3ffffa70)
A Port Input               [A] 00000000000000000000000010101010  (0x000000aa)
Final Result        [A + b<<4] 00111111111111111111101100011010  (0x3ffffb1a)
```

4.6.1.3 Shift Left and Subtract: A – b<<n

The shift left and subtract operation is performed like the shift left and add in Example 4–5, except for the ALU's function. The ALU function is A–(B&~C) for a shift left and subtract. Because the operation involves a subtraction, the default carry-in is 1.

Shift left and subtract is expressed in the PP's algebraic notation below:

```
dst = A – b<<n
```

dst:	Destination register for shift left and subtract result.
A:	Register input to the A port.
b:	Register input to the barrel rotator.
n:	Either a 5-bit constant or the five LSBs contained in a register.

This operation is equivalent to the explicit notation below:

```
dst = A – (b\\n & ~%n)
```

Example 4–6 shows 0x53FF FFA7 shifted left by four bits and the result subtracted from 0x0000 00AA.

Example 4–6. Shift Left and Subtract

```
Original Number            [b]  01010011111111111111111110100111  (0x53ffffa7)
Rotate Number by 4         [B]  00111111111111111111101001110101  (0x3ffffa75)
Mask                       [C]  00000000000000000000000000001111  (0x0000000f)
Shift Left Result       [B&~C]  00111111111111111111101001110000  (0x3ffffa70)
A port input               [A]  00000000000000000000000010101010  (0x000000aa)
Final result        [A – b<<4]  11000000000000000000011000111010  (0xC000063a)
```

4.6.1.4 Unsigned (Logical) Shift Right: >>u

Unsigned shift right (0s shift in from the left) operations can be performed by the PP in a single cycle. Because the barrel rotator performs a left rotate, a net right rotate requires the specified shift amount to be 32–n, where n is the number of bits to rotate right. Only the five LSBs of the rotation/shift amount are used by the rotator and mask generator so that the amounts 32 (0x20) and 0 are equivalent in terms of controlling the shift operation. For an immediate shift right amount, the assembler automatically performs the 32–n computation for shifts; for register-based shifts, you must take this into account when you load the desired rotate amount into the register.

Once the accommodation for right rotation is made, an unsigned shift right works like a shift left, except that a different ALU function is used. This operation can be broken down into the following elemental functions:

1) The barrel rotator performs a net rotation right with a rotation left by 32 minus the number of bits (n) to be rotated right. Bits rotated out of the LSBs rotate into the MSBs of the word.

2) The same amount sent to the rotator (32–n) is sent to the mask generator, and the mask generator generates 32–n right-justified 1s. The 0s in this mask (note that the 1s were used in the shift left case) indicate the corresponding bit locations that have rotated in. The right shift mask modifier (%!) causes a shift amount of 0 to generate a mask of all 1s so that no bits will be stripped off in step 3 for the zero-shift case.

3) The ALU function B&C combines the outputs of the rotator and mask generator.

An unsigned shift right operation is shown below in algebraic notation. To indicate an unsigned right shift, you must use a u after the right shift symbol >>. Note that a signed right shift is the default.

The assembler syntax for shift right operations depends on whether the shift amount is specified by a constant or in a register.

Since the input to the barrel rotator specifies the number of bits to rotate left, the register that specifies the shift right amount actually contains the 2s complement of the shift right amount that will result. As a reminder, the assembler requires a minus (–) before the shift amount register source. For a constant shift amount, the assembler assumes that the specified shift amount is the amount to shift right; therefore, it encodes the 2s complement of the specified constant as the immediate in the instruction. The two forms of specifying a right shift are shown below:

```
dst = b>>u constant  ; Constant shift amount.
dst = b>>u -dj       ; Register-sourced shift amount.
```

dst:	Destination register for the unsigned shift right result.
b:	Register input to the barrel rotator.
constant:	Constant defining the desired shift right amount.
dj:	Register specifying the 2s complement of the desired shift right amount.

Below are the equivalent operations that are actually performed by the ALU data path for a shift right by a constant and a register-sourced amount, respectively.

```
dst = b\\(32-constant) & %!(32-constant)  ;   Constant shift amount.
dst = b\\dj & %!dj                        ;   Register-sourced shift
                                          ;   amount.
```

Note that the shift-right mask modifier (%!) is used. This causes a shift amount of 0 to generate a mask of all 1s so that no bits are stripped off. If % was used without the modifier, a shift amount of 0 would generate a mask of all 0s. This would cause the undesired effect of stripping off all 32 bits of the result.

Example 4–7 shows an unsigned shift right of the number 0x5CFF FFA7 by four bit positions. To accomplish this, the barrel rotator input number specifies a left rotation amount of 32–4 (28).

Example 4–7. Unsigned Shift Right

```
Original Number        [b]  01011100111111111111111110100111  (0x5cffffa7)
Rotate by 32–4=28      [B]  01110101110011111111111111111010  (0x75cffffa)
Mask (32–4=28)         [C]  00001111111111111111111111111111  (0x0fffffff)
Unsigned Shift       [B&C]  00000101110011111111111111111010  (0x05cffffa)
Right
```

4.6.1.5 Unsigned (Logical) Shift Right and Add: A + b>>u n

In PP assembly code, the function A + b>>u is translated by the assembler into an instruction that uses the barrel rotator, mask generator, and three-input ALU function to perform the desired operation. The shift right and add works like the previous example of a simple shift right, except for the ALU's function. Instead of performing B&C as in a simple shift, the ALU performs the function A+(B&C).

Unsigned shift right and add is expressed in the PP's algebraic notation below. The syntax for a shift amount specified in a constant is different than a shift amount specified in a register. The assembly language default for a shift right, is a signed shift right so the u notation after the shift right symbol, >>u, designates an unsigned shift:

```
dst = A + b>>u constant   ; Constant shift amount.
dst = A + b>>u –dj        ; Register-sourced shift amount.
```

dst:	Destination register for the unsigned shift right and add result.
A:	Register input to the A port of the ALU.
b:	Register input to the barrel rotator.
constant:	Constant defining the desired shift right amount.
dj:	Register specifying the 2s complement of the desired shiftright amount.

These operations are equivalent to the explicit notation below:

```
dst = A + b\\(32-constant) & %!(32-constant)
                                          ;   Constant shift amount.
dst = A + b\\dj & %!dj                    ;   Register-sourced shift amount.
```

Remember that because the barrel rotator performs a left rotate, the amount specified in a register for the shift amount must be 32–n, where n is the number of bits to perform a net rotate right; you must take this into account with register-based shift amounts.

For shifts specified by a constant in the source code, the assembler automatically performs the 32–n computation so that you can specify the desired right shift amount. The rotate amount is also fed to the mask generator to form the mask needed to convert the rotate into a shift.

Example 4–8 shows the unsigned shift right by four and addition of the number 0x5CFF FFA7 to 0x0000 00AA. Since the barrel rotate amount specifies a left rotation amount, the number 32–4 (28) must be specified in order to shift right by four.

Example 4–8. Unsigned Shift Right and Add

```
Original Number              [b]  01011100111111111111111110100111  (0x5cffffa7)
Rotate by 32–4=28            [B]  01110101110011111111101001110101  (0x75cffffa)
Mask (32–4=28)               [C]  00001111111111111111111111111111  (0x0fffffff)
Unsigned Shift Right       [B&C]  00000101110011111111111111111010  (0x05cffffa)
A Port Input                 [A]  00000000000000000000000010101010  (0x000000aa)
Result                [A+b<<u 4]  00000101110100000000000010100100  (0x05d000a4)
```

4.6.1.6 Unsigned (Logical) Shift Right and subtract: A – b>>u n

The unsigned shift and subtract works like the previous two examples, except that the ALU function is changed to A – (B&C). As in all shift operations, the algebraic assembler automatically selects the ALU function and data path connections for the specified shift operation.

4.6.1.7 Signed (Arithmetic) Shift Right: >> or >>s

A signed shift right (sign bit shifts in from the left) can be performed by the PP in a single cycle. This operation uses sign detection hardware in the ALU's data path to change the function the ALU performs, according to the sign of the input to the barrel rotator.

As with unsigned shift right, the amount specified for the shift amount must be 32–n where n is the number of bits to rotate right. This operation can be broken down into the following elemental functions:

1) Perform a net rotate right by rotating left by 32 minus the number of bits (n) to be rotated right.

2) The same amount sent to the rotator (32–n) is sent to the mask generator, and the mask generator generates 32–n right-justified 1s. The 0s in this mask are then used to force bits to the sign of the MSB input to the barrel rotator.

3) The outputs of the rotator and mask generator are then combined by the ALU. If the sign bit at the input of the barrel rotator is 0, the ALU performs B & C which forces to 0 any bits that are 0 in the mask. If the sign bit is 1, the ALU performs B|~C, which forces to 1 any bits that are 0 in the mask.

In step 3 above, the sign bit of the input to the barrel rotator controls the ALU function; hardware logic inverts the four LSBs of the ALU function code (see Table 4–1) according to the sign bit. The ALU function B|~C is specified, and the sign control hardware inverts the four LSBs of the function code if the sign bit is 0.

The sign-extend function is implied by the ALU functions A + (B|~C) and A – (B|~C), but they are not directly accepted by the assembler in the base set of operations. Therefore, there is no explicit way to specify the sign-extended shift right other than by using the algebraic symbols >>s or simply >>.

The default for a shift right is a signed shift right so that the s for signed (shown in the second equation below) is optional. The assembler automatically selects the opcode combination that indicates that the ALU function control is dependent on the sign of the operand.

In the case of immediate shift amounts, the assembler automatically makes the 32–n adjustment to the specified shift right amount.

For a register-sourced shift amount, the assembler requires a minus (–) before the register to remind you that the contents of the register actually specify the 2s complement of the shift right amount that will occur. A sign-extended shift right can be expressed in either of the following ways:

```
dst = b>> constant     ; Constant shift amount.
dst = b>>s constant    ; Constant shift amount.
dst = b>> –dj          ; Register sourced shift amount.
dst = b>>s –dj         ; Register sourced shift amount.
```

dst:	Destination register for the signed shift right result.
b:	Register input to the barrel rotator.
constant:	Constant defining the desired shift right amount.
dj:	Register specifying the 2s complement of the desired signed shift right amount.

Shifting 0x5CFF FFA7, a number with the sign bit of 0, arithmetically right by four bits is identical to shifting it logically right by four bits, as shown earlier in Example 4–7. Example 4–9 shows what happens if the negative number 0xECFF FFA7 is arithmetically shifted right by four bits.

Example 4–9. Signed Shift Right

```
Original Number          [b]  11101100111111111111111110100111  (0xecffffa7)
Rotate by 32–4=28        [B]  11001111111111111111101001110101  (0x7ecffffa)
Mask (32–4=28)           [C]  00001111111111111111111111111111  (0x0fffffff)
Signed Shift Right    [B|~C]  11111110110011111111111111111010  (0xfecffffa)
```

Note that in this case, the ALU function chosen is A + (B | ~C), but no A input is specified. In this example, the A input can be forced to zero using a function modifier (see subsection 8.3.4, *ALU Function Modifiers,* for more information about forcing the A input to zero).

4.6.1.8 Signed (Arithmetic) Shift Right and Add: A + b>>n

Signed shift right and add works like the simple signed shift right, except for selection between the pair of ALU functions according to the sign bit of the value to be shifted. The sign control hardware looks at the MSB (sign bit) of the input to the barrel rotator and selects the function A+(B&C) if the sign bit is 0 or the function A + (B|~C) if the sign bit is 1. Note that the ALU function code differs only in the bitwise inversion of the four LSBs.

The assembly language default for a shift right is a signed shift, so the shift operation can be expressed with either >> or >>s. The assembler automatically uses the A + (B|~C) opcode combination that selects the ALU function control that is dependent on the sign of the operand (switches to A + (B & C) if the sign is zero).

```
dst = A + b>> constant    ; Constant shift amount.
dst = A + b>>s constant   ; Constant shift amount.
dst = A + b>> –dj         ; Register sourced shift amount.
dst = A + b>>s –dj        ; Register sourced shift amount.
```

dst:	Destination register for the signed shift right and add result.
A:	Register input to the A port of the ALU.
b:	Register input to the barrel rotator.
constant:	Constant defining the desired shift right amount.
dj:	Register specifying the 2s complement of the desired signed shift right amount.

4.6.1.9 Signed (Arithmetic) Shift Right and Subtract: A – b>> n

Signed shift right and subtraction is performed like the shift right and add, except that the selection of ALU function code is either A – (B | ~C) or A – (B & C) instead of A + (B |~C) or A + (B & C).

If the sign bit is 0, A–(B&C) is used; if the sign bit is 1, A–(B|~C) is used. Both these functions have an implied carry-in of 1. The equations below show how to express the signed shift and subtract operation in PP assembly code.

```
dst = A – b>> constant   ; Constant shift amount.
dst = A – b>>s constant  ; Constant shift amount.
dst = A – b>> –dj        ; Register sourced shift amount.
dst = A – b>>s –dj       ; Register sourced shift amount.
```

dst:	Destination register for the signed shift right and subtract result.
A:	Register input to the A port of the ALU.
b:	Register input to the barrel rotator.
constant:	Constant defining the desired shift right amount.
dj:	Register specifying the 2s complement of the desired signed shift right amount.

4.6.2 Absolute Value: |value|

Functions involving absolute value are supported on the PP by sign-detection hardware that modifies the ALU function according to the sign bit of the input to the barrel rotator. This hardware is selected in assembly language by using the absolute value operator (|*value*|).

To perform the ALU function A + |B|, the sign-detection hardware chooses between the function A + (B&~C | –B&C) (function code 96) and the function A – B (function code 99).

- If input to the barrel rotator is positive (sign-bit is 0), then the function performed by the ALU is A + (B&~C | –B&C). The assembler encodes the input to the C port as 0 so that the result of the function is essentially A + B.
- If the input to the barrel rotator is negative (sign-bit is 1), then the four LSBs of the ALU function are inverted so that the performed function is A – B.

By adding the B port input if it is positive or subtracting the B port input if is is negative, the ALU essentially performs A + |B|.

Example 4–10 shows the absolute value of 0xFFFF FFF1 (–15) added to 0x0000 0010 (16) and vice versa.

The function |B| can be performed using A + |B| as described above but with the input to the A port set to 0.

Example 4–10. Absolute Value

```
A+|B| (for a positive input to the barrel rotator)

input 1                 [A]  11111111111111111111111111110001  (0xFFFFFFF1)
input 2                 [B]  00000000000000000000000000010000  (0x00000010)
forced 0                [C]  00000000000000000000000000000000  (0x00000000)
result  [A + (B&~C | –B&C)]  00000000000000000000000000000001  (0x00000001)
```

```
A+|B| (for a negative input to the barrel rotator)

input 1                 [A]  00000000000000000000000000010000  (0x00000010)
input 2                 [B]  11111111111111111111111111110001  (0xFFFFFFF1)

result               [A – B]  00000000000000000000000000011111  (0x0000001F)
```

4.6.3 Using Field Extraction to Right-Justify Fields

The barrel rotator, mask generator, and ALU can be used to perform field extraction in a single cycle. A field extraction takes a field of bits in a word starting at an arbitrary bit position, strips off the bits outside the field, and right-justifies the field.

Field extraction is performed by rotating the word left by the number of bits necessary to right-justify the field and then masking the result of the rotation by the number of bits in the size of the field. Thus, unlike shifts, the rotation amount (based on the bit position) and the mask input (based on the field size) are not necessarily the same amount.

The instruction set supports field extraction via an extended ALU operation. The syntax and setup details for this operation are discussed in subsection 8.4.6, *Using EALU Operations Without a Parallel Multiply*. For sake of this discussion, the following expression defines the operation performed by the ALU data path.

Field_Extract = (Value \\ (32-starting_bit)) & %!Field_size

The %! operator uses the mask generator to form a mask that has the number of right-justified 1s equal to the field size, except for an input of 0, in which case, all bits are 1 so that no bits are masked by the bitwise AND operation. There is no hardware check to ensure that the field size is not greater than the starting bit position (in which case bits that have rotated out will wrap into the result).

Example 4–11 demonstrates field extraction of a 4-bit field starting at bit 24 (the eighth bit from the left). The number must be rotated left by 32–24 (8) bits in order to right-justify the field. The field size—in this example, four bits—goes to the mask generator. A bitwise AND of the mask and the rotated number is used to extract the desired field.

Example 4–11. Field Extraction

```
Original Number        [b]  01011100111111111111111110100111  (0x5cffffa7)
Rotate left by 8       [B]  11111111111111111010011101011100  (0xffffa75c)
Mask 4                 [C]  00000000000000000000000000001111  (0x0000000f)
Extracted Field      [B&C]  00000000000000000000000000001100  (0x0000000c)
```

4.6.4 Color Expansion

A basic graphics operation is the conversion of 1-bit-per-pixel shape descriptors into pixel size quantities. For example, the pattern template of bitmapped text fonts are often stored at 1-bit-per-pixel and are then expanded into the desired color(s) as they are drawn into the display memory. Generally, the 1s in the shape select a 1 color, and the 0s in the shape select a 0 color or act as a place saver (transparent) pixel.

Color expansion is commonly used with BitBlt (bit-aligned block transfer) algorithms. To perform a complete BitBlt, the data must be rotated and merged with prior data to align the bits in the data to be expanded with the pixel alignment of words. The barrel rotator and ALU can be used to move aligned words into the mf register. The examples below assume that the pixel data is already aligned properly in the mf register.

Color expansion can be performed by the PP via the expander and the ALU as shown in Example 4–12. Example 4–12 shows a 4-bit number in the mf register expanded into four 8-bit fields of all 1s or all 0s.

The ALU performs the function (A&~C)|(B&C), where the A input to the ALU has the contents of a register holding the 0 color, the B input has the contents of a register holding the 1 color, and the C input has the expanded mf register.

The ALU function effectively selects the A or B input as the output according to the C input. Since the mf is expanded by the pixel size (in this example eight bits), the bits in the mf register are effectively selecting between pixels for the result.

For the 0 color and 1 color to be solid values, they must be replicated in the two source registers (a special option of register-to-register moves supports replication of an amount throughout a register).

Example 4–12. Color Expansion

```
mf Register                  xxxxxxxx   xxxxxxxx   xxxxxxxx   xxxx0110
Expanded mf              [C] 00000000   11111111   11111111   00000000
0 Color                  [A] 10101010   10101010   10101010   10101010
1 Color                  [B] 11110000   11110000   11110000   11110000
Result         [(A&~C)|B&C)] 10101010   11110000   11110000   10101010
```

Simple color expansion forces the result to be one of two solid colors. Often, particularly with kerned text letters that can overlap, it is desirable to expand 1s in the fonts to the 1 color but have 0s in the font act as place saver or transparent pixels.

This transparent color expansion can be performed by using a register containing the original contents of the destination, as shown in Example 4–13. The ALU performs the same function as in Example 4–12, except that the original destination is used as one of the inputs to the ALU.

Example 4–13. Color Expansion With Transparency

```
mf register                    xxxxxxxx  xxxxxxxx  xxxxxxxx xxxx0110

expanded mf                  [C]  00000000 11111111 11111111 00000000

Original Destination         [A]  11001100 10101010 11101110 11111111

1 color                      [B]  11110000 11110000 11110000 11110000

New Destination[(A&~C)|(B&C)]     11001100 11110000 11110000 11111111
```

4.6.5 Using the mf Rotator and Split ALU to Match Pixel Values

Example 4–14 shows the generation of a 1-bit-per-pixel mask on the basis of an exact match to a fixed quantity for a series of 8-bit quantities. In this case, mf register bits are set if the corresponding split-ALU result is 0. This example executes two instructions in a row to demonstrate how results can be accumulated by rotating the mf register.

Example 4–14 generates an 8-bit quantity representing eight adjacent values. The routine could be extended to create up to 32 bits before saving the mf register.

Example 4–14. Using the mf Rotator to Accumulate the Results From Two Instructions

```
original mf register            xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx

first quantity            [A]   00000011 00001111 00000001 00000011

compare value             [B]   00000011 00000011 00000011 00000011

subtraction result      [A–B]   00000000 00001100 11111110 00000000

zero compares                   1        0        0        1

mf after 1st instruction        00000000 00000000 00000000 00001001
  (with pre-clear of mf)

-----------------------------------------------------------------

second quantity           [A]   00000111 11111100 00000011 00000000

2nd subtraction result  [A–B]   00000100 11111001 00000000 11111101
  (same compare value)

zero compares                   0        0        1        0

mf set with rotation            00000000 00000000 00000000 10010010
```

4.6.6 Threshold Detection Using Multiple Arithmetic

Threshold detection involves comparing pixel values to a threshold level and producing a 1-bit value for each pixel that indicates whether the pixel is greater than (bit set to 1) or less than (bit set to 0) the threshold. Depending on the use, the equal case can be grouped with either greater or less than.

The carry-out result from a subtraction can be used to indicate greater than or equal to between two unsigned numbers. If B is subtracted from A, a carry-out occurs only if A is greater than or equal to B.

To understand how the carry-outs are generated, note that the ALU performs subtraction (minuend –- subtrahend), in effect, by adding the minuend to the 1s complement plus one of the subtrahend. For unsigned numbers, this results in a carry-out whenever the result of the subtraction is greater than or equal to 0.

The PP sets carry-out status strictly on the basis of the natural carry-out. Thus a carry-out from a subtraction is equivalent to not-borrow. When most other microprocessors set the status for a subtraction, they invert the sense of natural carry-out in order to represent a borrow.

Example 4–15 uses a multiple-byte subtract to compare four 8-bit pixels to the threshold level (which is replicated into each byte of a 32-bit word). The carryout for each segment of the split ALU is one for input pixels greater than or equal to the threshold, and is zero for input pixels less than the threshold. Thus, in a single cycle, four 8-bit pixels are thresholded, producing four 1-bit values.

The carry-out from each segment of the split ALU is written to the mf register. The mf register can optionally be rotated by four before the multiple status is written. This allows a full 32-bit word of 1-bit values to be accumulated in eight cycles and then stored to memory. This type of threshold detection can be used as part of object identification.

Example 4–15. Threshold Based Conversion From Eight-Bit to One-Bit Space

```
Input, I              [A]  00001100  00000001  00000110  00000111
Threshold, T          [B]  00000111  00000111  00000111  00000111
Compare, I – T      [A–B]  00000101  11111010  11111111  00000000
Carry outs                 1         0         0         1
mf result                  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxx1001
```

4.6.7 Using Multiple Arithmetic and the Expander

Multiple arithmetic status information in the mf register can be expanded and used to control multiple parallel-byte or parallel-halfword operations. Some of the pixel-processing operations you can perform by combining multiple arithmetic with the expander are:

- Transparency
- MAX and MIN
- Add-with-saturate

4.6.7.1 Transparency

You can combine multiple-byte arithmetic with expansion to perform a transparency operation. Transparency is commonly used when performing rectangular BitBlts of shapes that are not rectangular.

The transparent pixels are used as place saver pixels that do not affect the destination and thus allow the original destination to show through. With transparency, only the pixels in the source that are not equal to the transparent code are replaced in the destination.

Example 4–16 shows a transparency operation. In the first instruction, the transparent color code is subtracted from the source, and the mf register is set on the basis of the zero compares for the split ALU. If a given 8-bit quantity matches the transparent code, it will set the corresponding bit in the mf register to 1. The 8-bit transparent code of 0x03 has been replicated four times in the example so that all pixels are compared to the same value.

The second instruction shown in Example 4–16 uses the expanded mf register to control the selection, pixel by pixel, of the source or destination. The PP's ALU supports three-input Boolean (and mixed Boolean and arithmetic) functions that can be used to combine the source and destination.

The ALU performs the function (A&~C)|(B&C). While the Boolean function is performed bit-by-bit, the mf register bits are expanded to the pixel size of eight bits; therefore, the mask created by the mf expander selects between pixels.

Example 4–16. Transparency Operation on Four 8-Bit Pixels in Two Cycles

```
First instruction

   Source (SRC)                  [A]  00000011   01110011   00000011   00000001
   Transparent Code (TC)         [B]  00000011   00000011   00000011   00000011
   SRC – TC                    [A–B]  00000000   01110000   00000000   11111110
   Zero Compares                      1          0          1          0
   mf Register                        xxxxxxxx xxxxxxxx xxxxxxxx xxxx1010
   ------------------------------------------------------------------

Second Instruction

   Expanded mf (@mf)             [C]  11111111   00000000   11111111   00000000
   Original Dest. (DEST)         [B]  11110001   00110011   01110111   11111111
   (SRC & ~@mf)|(DEST & @mf)          11110001   01110011   01110111   00000001
```

4.6.7.2 MAX and MIN

The MAX function outputs the greater of the two unsigned inputs. Example 4–17 shows how the MAX operation can be performed on four 8-bit quantities in two instructions.

The first instruction performs multiple-byte subtraction setting the mf register bits based on the carry-out of the corresponding segment of the ALU. A carry-out of one indicates that the OP1 is greater than or equal to OP2; a carry-out of zero indicates that OP1 is less than OP2 (and thus, OP2 is greater than or equal to OP1).

The second instruction performs the operation (OP1 & @mf)|(OP2 & ~@mf), in which the bits set in the previous instruction are used to control the merging between OP1 and OP2.

Example 4–17. The MAX Function on Four 8-Bit Numbers

```
First Instruction

  First Operand (OP1)         [A]  00000001 11111110 00000011 00000100

  Second Operand (OP2)        [B]  00000011 00000111 00000111 00000011

  OP1 – OP2                 [A–B]  11111110 11110111 11111100 00000001

  Carry Outs (Not-Borrows)         0        1        0        1

  mf Register                      xxxxxxxx xxxxxxxx xxxxxxxx xxxx0101

  +-------------------------------------------------------------------

Second Instruction

  Expanded mf (@mf)           [C]  00000000 11111111 00000000 11111111

  (OP1 & @mf)|(OP2 & ~@mf)         00000011 11111110 00000111 00000100
```

The MIN function returns the lesser of two unsigned inputs. The MIN function works like the MAX function above, except that the ALU function in the second instruction is changed to (OP1 & ~@mf)|(OP2 & @mf) to select the lesser rather than greater of the two numbers.

4.6.7.3 Add-With-Saturate

When two unsigned numbers are added, a carry-out indicates that the result is greater than can be expressed in the output precision. This carry-out represents the MSB of precision of the result. Thus, the carry-outs that are saved in mf can be used to maintain precision. Often, in image processing and graphics where it is desirable to keep quantities to small integers, this overflow can be used to saturate or clamp the result to a maximum value.

The add-with-saturate function is commonly used to keep small integer results from rolling over the highest number back to a low number. Example 4–18 performs an add-with-saturate on four 8-bit numbers in two instructions. Results that overflow are clamped to 0xFF.

The first instruction in the example performs a multiple-byte addition, setting bits in the mf register on the basis of carry-outs. A carry-out of one indicates that the result was too big to fit into an 8-bit value. The second instruction selects either the result from the addition if the corresponding carry-out was zero, or 0xFF if the corresponding carry-out was one.

Example 4–18. The Add-With-Saturate Function on Four 8-Bit Numbers

```
First Instruction

  First Operand (OP1)        [A]  00000001 11111001 00000011 00111111

  Second Operand (OP2)       [B]  11111111 00001011 00000111 01111111

  RESULT = OP1 + OP2       [A+B]  00000000 00000100 00001010 10111110

  Carry Outs                      1        1        0        0

  mf Register                     xxxxxxxx xxxxxxxx xxxxxxxx xxxx1100

  ----------------------------------------------------------------

Second Instruction

  Expanded mf (@mf)          [C]  11111111 11111111 00000000 00000000

  (RESULT | @mf)           [A|C]  11111111 11111111 00001010 10111110
```

4.6.8 Adding Two Masked Inputs

Example 4–19 shows how four 8-bit numbers can be totaled into one 32-bit sum in just two passes through the ALU. The first instruction sums two pairs of 8-bit numbers into two 16 bit numbers, and the second instruction then sums the two 16-bit numbers.

Example 4–19. Totaling Four 8-Bit Numbers in Two Instructions

```
First instruction

 Original Number    [A,b]   00010010101010111000100111001101 (0x12ab89cd)

 Rotate by 8          [B]   10101011100010011100110100010010 (0xab89cd12)

 Mask                 [C]   00000000111111110000000011111111 (0x00ff00ff)

 Masked Add   [A&C + B&C]   00000001001101000000000010111111 (0x013400df)

-------------------------------------------------------------------------

Second instruction

 Prior Result       [A,b]   00000001001101000000000010111111 (0x013400df)

 Rotate by 16         [B]   00000000000000000000000100110100 (0x00000134)

 Mask                 [C]   00000000000000001111111111111111 (0x0000ffff)

 Final Result [A&C + B&C]   00000000000000000000001000010011 (0x00000213)
```

4.6.9 Using Multiple Arithmetic With Conditional Function Control

The PP's 32-bit ALU can be split into two 16-bit ALUs or four 8-bit ALUs to perform multiple arithmetic operations. The mixed Boolean and arithmetic capability of the ALU gives added flexibility. A mask of all 1s or all 0s for each split of the ALU can be used to control the ALU function for each split ALU section.

The expanded mf, which goes to the ALU's C port, facilitates the conditional control of each section of the split ALU. Generally, Msize (which determines the number of times each mf register bit is replicated) and Asize (which determines the size of the split ALU) are set to the same amount so that each mf register bit controls the function of the corresponding split-ALU segment. Because the bits in the mf register can be set on the basis of status results from prior split ALU functions, each split can be conditionally controlled.

4.6.9.1 Conditionally Adding or Subtracting B Based on C

The ALU functions A+((–B&C)|(B&~C)) and A+((B&C)|(–B&~C)) are available in the base set operations when the expander output goes to the C-port input. These combinations of ALU expander functions enable each bit in the mf register to control whether addition or subtraction is performed for each split ALU. The carry-in to each split of the ALU is controlled by the LSB of the C input, causing a carry-in of 0 for an add and 1 for a subtract.

Example 4–20 shows how the A + ((B&C)|(–B&~C)) function can be used in combination with the split ALU and expander to complete the absolute value of the difference for four pairs of 8-bit numbers in two cycles.

The first instruction uses the split ALU to subtract four pairs of unsigned 8-bit numbers and sets carries in the mf register. Note that when subtracting unsigned numbers, a carry will be generated only if the result is greater than or equal to zero; therefore, the bits set in the mf register at the end of the first instruction record whether the intermediate result was positive or negative.

In the second instruction, the set bits in the mf control whether to add or subtract (add if a positive result and subtract if a negative result). This essentially performs a multiple sum of the absolute values function. In the example, the split-ALU size and the expansion size are eight bits.

Example 4–20. Taking Absolute Value of the Difference of Four 8-bit Numbers

```
First instruction

 1st number                [A]   00010010101010111000100111001101 (0x12ab89cd)

 2nd number                [B]   00110100101010110101011011111110 (0x34ab56fe)

 Split ALU subtract      [A–B]   11011110000000000011001111001111 (0xde0033cf)

 mf result        [carry outs]   xxxxxxxxxxxxxxxxxxxxxxxxxxxx0110 (0xxxxxxxx6)

-------------------------------------------------------------------------------

Second instruction

 Zero                      [A]   00000000000000000000000000000000 (0x00000000)

 Prior Result              [B]   11011110000000000011001111001111 (0xde0033cf)

 Expanded mf               [C]   00000000111111111111111100000000 (0x00ffff00)

 Final Result[0–(B&C)|(–B&~C)]   00100010000000000011001100110001 (0x22003331)
```

4.7 Using the Multiplier Data Path

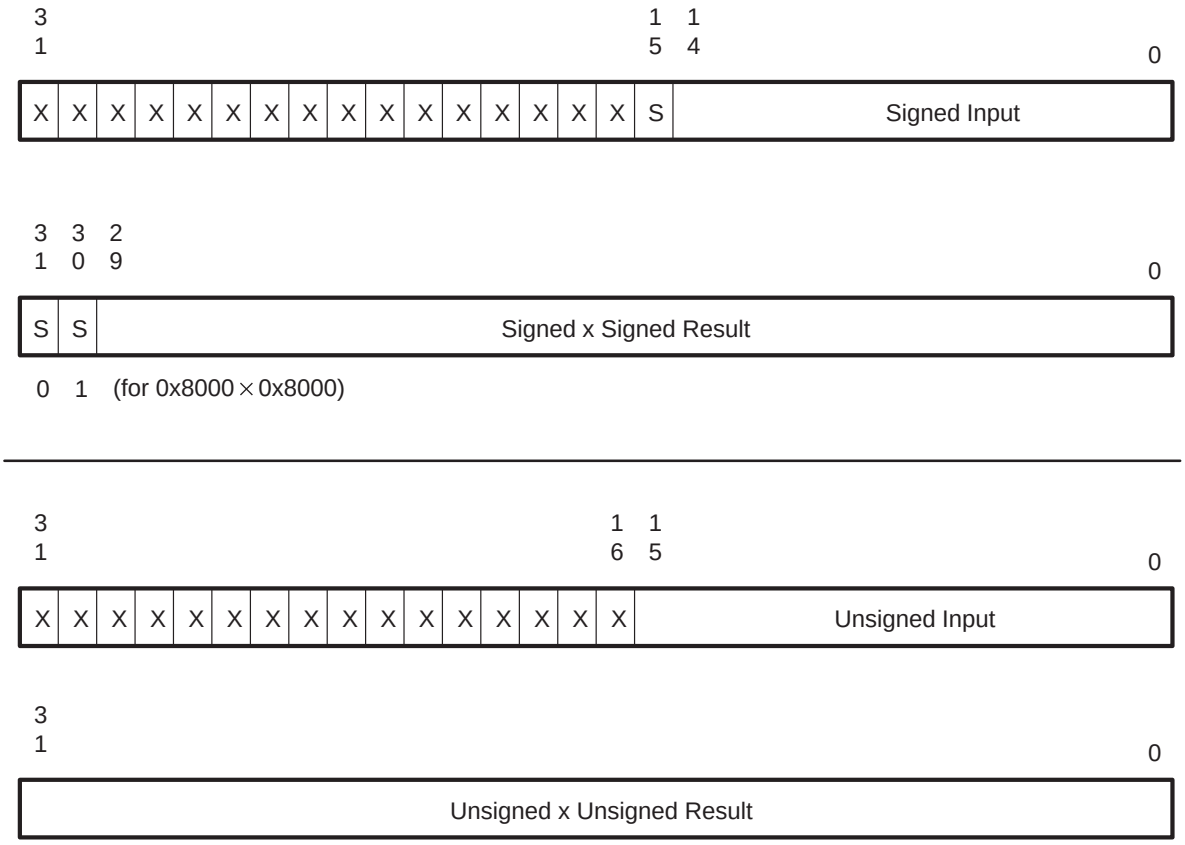
The PP's hardware multiplier can perform one 16×16 multiply with a 32-bit result ($16 \times 16 = 32$) or two 8×8 multiplies with two 16-bit results (two $8 \times 8 = 16$) in a single cycle. A 16×16 multiply can be signed or unsigned. For two 8×8 multiplies, one of the inputs must be unsigned; the other input can be signed or unsigned. The output of the multiplier can be shifted left by up to three bits. Special rounding hardware, available on 16×16 multiplies, can be used to help maintain precision.

All multiplies are executed in a single cycle. The result of the multiply from one instruction can therefore be used by the very next instruction.

4.7.1 Performing 16x16 Multiply Operations

The multiplier can perform 16 × 16 = 32 integer multiplies. Both signed and unsigned multiplies are supported. Signed multiply is the default assumed by the assembler; unsigned multiply is indicated by a **u** attached to the assignment operator for the multiply (=u). All multiplier operands (2 source, 1 destination) must be D registers, (d0–d7). The input data format and corresponding outputs are shown in Figure 4–5. Since only the 16 LSBs in the input registers are used by the multiplier, it is not necessary to zero- or sign-extend the inputs (that is, the 16 MSBs are ignored).

Figure 4–5. 16 x 16 Multiplier Data Formats

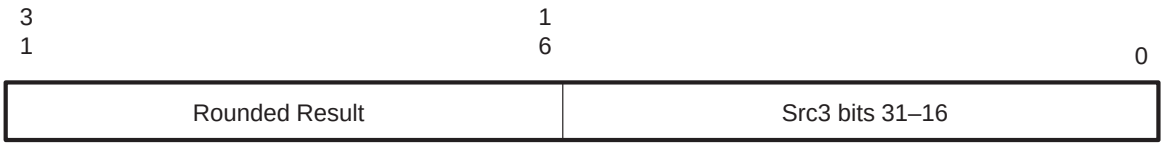


No status setting is associated with multiplies for the PP.

4.7.2 Rounding and Scaling 16x16 Multiply Operations

When you use the multiplier in parallel with an EALU, the single-cycle multiply can be configured to shift its output left by 0, 1, 2, or 3 bits and/or to round bit 15 into the 16 MSBs (of the optionally shifted result). When rounding is performed, the result of the rounded multiply occupies bits 31–16 of the destination value, and bits 15–0 are filled with bits 31–16 of the first multiply source operand (src3), as shown in Figure 4–6.

Figure 4–6. Rounded Result Format



When rounding is performed, the output of the multiplier is shifted left (scaled) by 0, 1-, 2-, or 3-bit positions according to the default multiply shift specified in d0. After scaling, bit 15 of the result is then added to bits 31–16 of the result to perform half-bit rounding into the result's most significant halfword. The result after rounding then becomes bits 31–16 of the destination.

Bits 15–0 of the destination are filled with bits 31–16 of src3. Copying the MSBs of src3 facilitates efficient processing of a series of 16-bit quantities packed into 32-bit words.

Example 4–21 shows how copying the src3 MSBs to the LSBs of the destination can be used. Two 16-bit values A and B, stored in d1, are multiplied in two successive instructions (with rounding) by values C1 and C2, stored in two other registers. After the second instruction, the two 16-bit results are concatenated in d4.

There is no mechanism to detect any significant bits that are shifted out by the scaling hardware in the multiplier data path. You must ensure that no significant bits are shifted out in the design of your algorithm.

Rounded multiplies can be specified only with the extended ALU (EALU) opcode format. Subsection 8.5.2, *Multiplying in Parallel With an EALU Operation*, provides details on how to specify a rounded multiply operation to the assembler and also shows more detailed examples of usage. Subsection 15.2.4, *Discrete Cosine Transform (DCT),* provides another example of rounding and shifting with multiplies.

Example 4–21. Rounded Multiply

31 … 16	15 … 0	
A	B	d1

31 … 16	15 … 0	
Don’t Care	C1	d2

31 … 16	15 … 0	
Don’t Care	C2	d3

First Instruction

d4 =r d1 * d2

Result

31 … 16	15 … 0	
Rounded B*C1	A	d4

Second Instruction

d4 =r d4 * d3

Result

31 … 16	15 … 0	
Rounded A*C2	Rounded B*C1	d4

4.7.3 Performing Split Multiply

The PP multiplier can perform two simultaneous 8-bit by 8-bit multiplies referred to as a **split multiply**. Byte 0 in the first input word is multiplied by byte 0 in the second input word and produces a 16-bit result in the least significant halfword of the destination register.

Likewise, byte 1 in the first input word is multiplied by byte 1 in the second input word and produces a 16-bit result in the most significant halfword of the destination register. For split multiplies, the first input word always contains unsigned byte multiplicands. The second input word can contain either signed or unsigned byte multiplicands, as determined by the instruction.

When the second input word is signed, this is referred to as a **signed split multiply**. A signed split multiply can be used to multiply unsigned pixel data by signed coefficients to produce a signed result.

The input and output data formats for signed and unsigned split multiplies are shown in Figure 4–7 and Figure 4–8, respectively. It is not necessary to sign-extend or zero-extend the input data; the 16 MSBs of data are always ignored.

Figure 4–7. Signed Split Multiply Data Formats

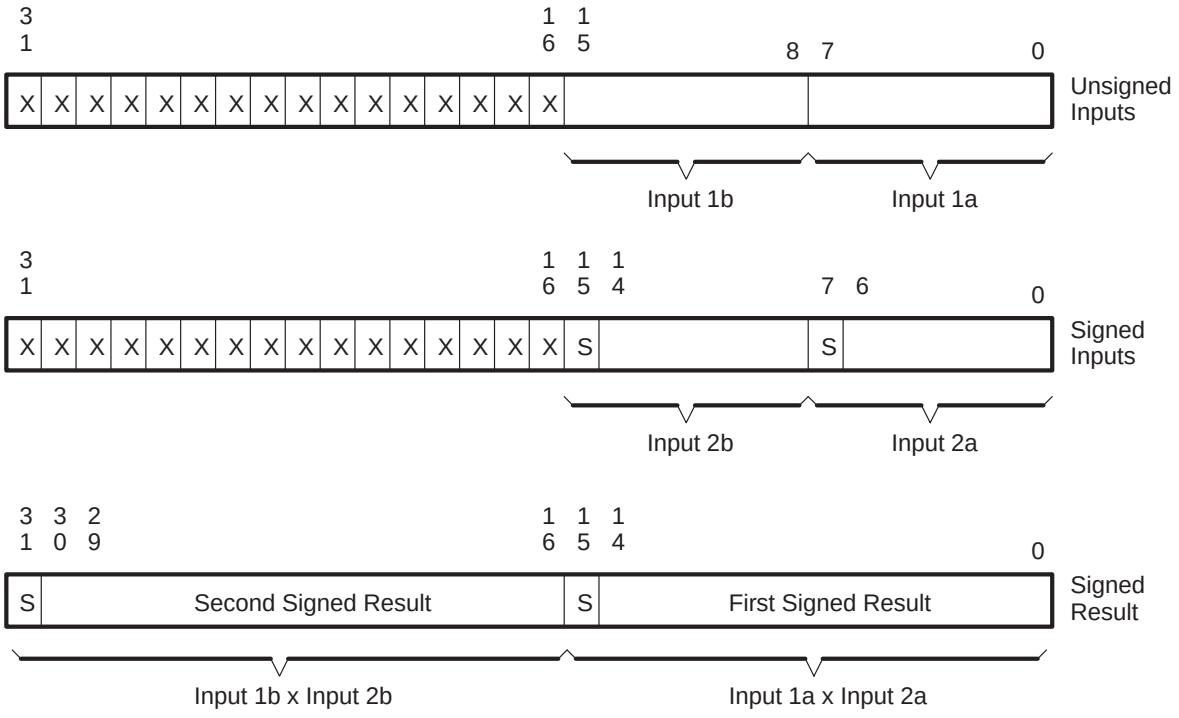
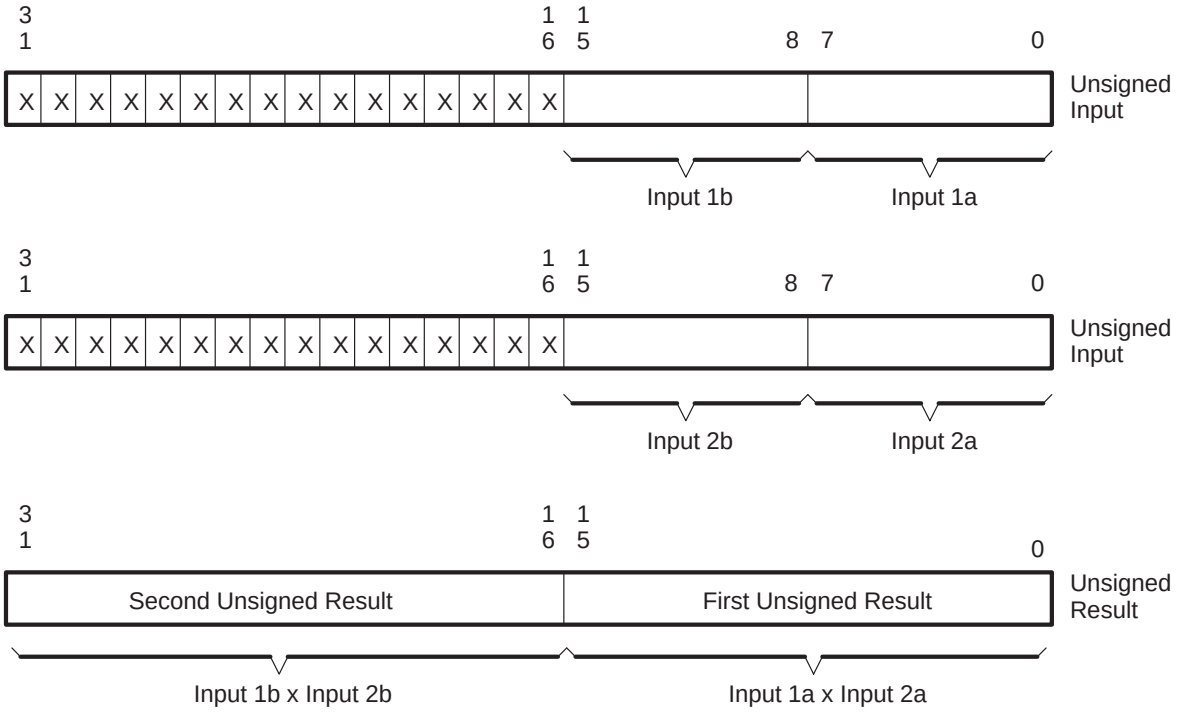


Figure 4–8. Unsigned Split Multiply Data Formats



Split multiplies can be specified only with the extended ALU (EALU) opcode format. Section 8.5, *Multiply Operations*, describes the assembly language syntax for a split multiply and also shows an example of split multiply usage.

While its applicability is sometimes limited because of precision considerations, split multiplies can be used to perform small kernels or the first stage of a series of multiplies in which the latter stages use a 16×16 multiply.

Chapter 5

Understanding the Program Flow Control Unit

This chapter describes the PP's program flow control unit and its components. Chapter 9, *Interrupts and Reset*, Chapter 10, *Understanding Branches and Calls*, and Chapter 11, *Understanding the Loop Control Logic*, provide a software-oriented description of interrupts, branching, calls, and zero-overhead looping program flow control elements.

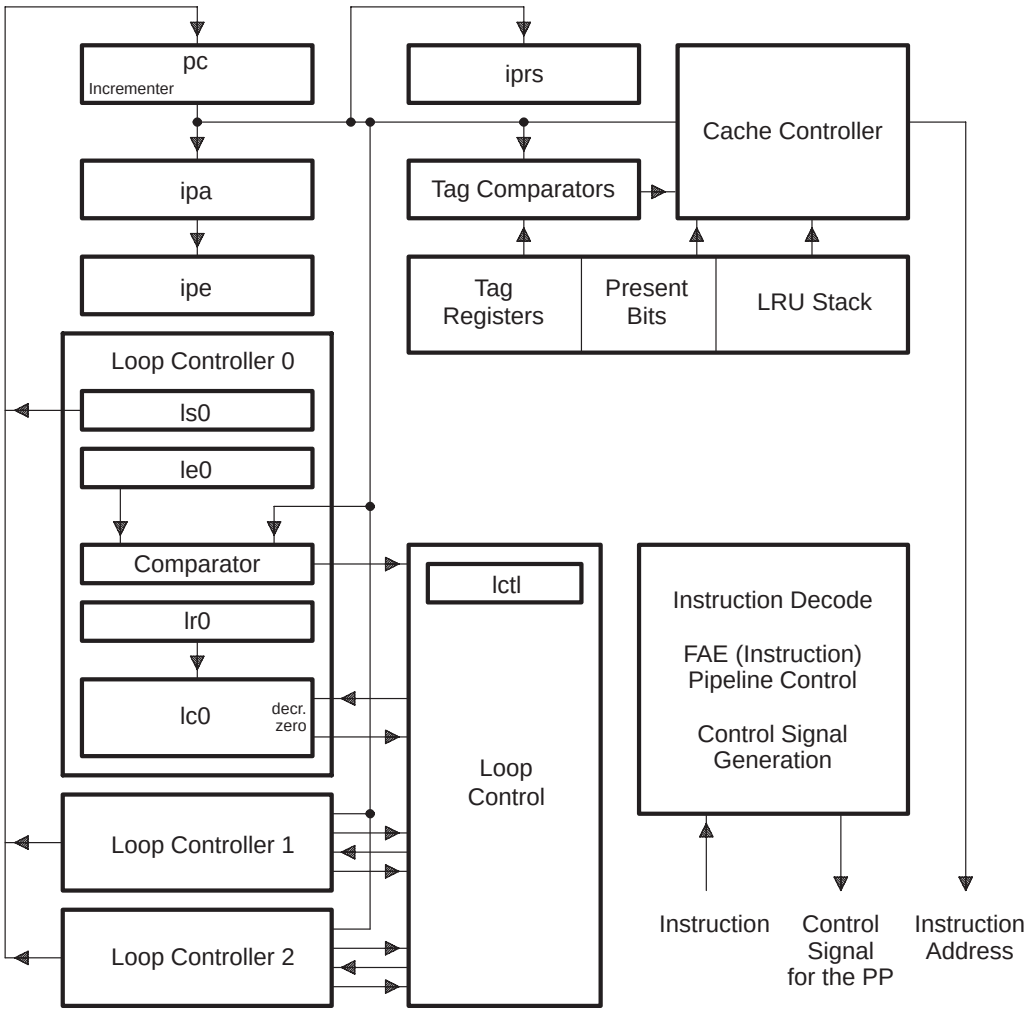
Topics

5.1	Overview of the Program Flow Control Unit	PP: 5-2
5.2	FAE Instruction Pipeline	PP: 5-3
5.3	Loop Controllers	PP: 5-12
5.4	Cache Controller	PP: 5-15

5.1 Overview of the Program Flow Control Unit

The program flow control unit performs instruction fetching and decoding, handles any necessary handshaking with the transfer controller, and facilitates interrupt response and prioritization. Figure 5–1 shows a block diagram of the program flow control unit.

Figure 5–1. Program Flow Control Unit Block Diagram



5.2 FAE Instruction Pipeline

The PP has a three-stage instruction pipeline that the program flow control unit controls. This pipeline is different from the MP's three-stage pipeline (the FEA pipeline). The PP's pipeline stages are summarized as follows:

- **F**etch instruction stage

  The address contained in the program counter (pc) is compared to cache-tag register and present flags.

  - If the instruction is in cache, it is fetched, and instruction decoding is started.
  - If the instruction is not in cache, the program flow control unit issues a cache service request to the TC and stalls the PP until the cache is loaded with the instruction.

  Once the instruction has been fetched, the pc is incremented, unless the pc is written to by a software branch or reloaded with a loop start address by the hardware loop control.

- **A**ddress unit computation stage

  All address unit computations specified by the previous fetched instruction are performed. The computed addresses are sent to the crossbar at the end of this stage for contention detection/prioritization. The actual memory accesses do not occur until the execute stage.

- **E**xecute data unit operations and memory transfers stage

  In this stage, operands from the register file are fed to the data unit, the data unit operations are performed, and the results are written back to the register file. Also in this stage, data is transferred between registers or between registers and memory, as specified by the global and local transfers. Conflicts in accessing memory may stall the pipeline at this point.

Since the order of the stages is **f**etch, **a**ddress, and **e**xecute, the operation is referred to as an **FAE pipeline**. Pipeline stages of three instructions overlap, as shown in Figure 5–2. At any given time, one instruction is at the fetch stage, another is at the address stage, and the third is at the execute stage. Thus, when no stall conditions occur, the net throughput is one instruction every cycle.

Figure 5–2. FAE (Instruction) Pipeline

Instruction	T1	T2	T3	T4	T5
One	Fetch	Address	Execute		
Two		Fetch	Address	Execute	
Three			Fetch	Address	Execute

5.2.1 Program Counter-Related Registers

The PP contains program-counter-related registers, including an instruction pointer for each stage of the FAE pipeline and an instruction pointer for returns from subroutine calls. These registers are summarized below:

- **pc** (program counter)

  The pc register points to the instruction currently being fetched. The value in the pc register is incremented by one instruction (eight bytes) each time the pipeline advances, unless the pc is written to by software or by one of the loop controllers, or if an interrupt is taken. When a PP operation writes to the pc register, the program flow changes. You can specify branches and calls by software writes to the pc register. When writing to the pc, you can use one of two different register codes:

  - The **br** register code (used for branches) modifies the pc register.
  - The **call** register code (used for subroutine calls) saves the return address in the iprs register (described below), in addition to modifying the pc register.

- **ipa** (instruction pointer address stage)

  This read-only pipeline storage register tracks the address of the instruction that is at the address stage of the pipeline.

- **ipe** (instruction pointer execute stage)

  The ipe register is read-only and is used for program counter relative addressing because it tracks the address of the instruction being executed. **Do not** use the pc register as a source operand for program counter relative addressing: the pc may bear no relation to the instruction being executed, because of a branch or loop.

- **iprs** (instruction pointer return from subroutine)

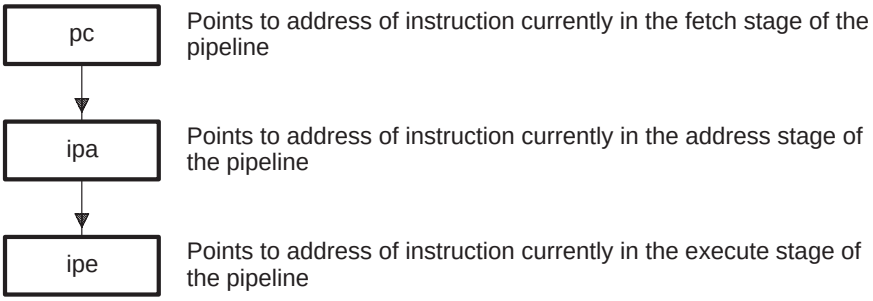
  This register contains an instruction pointer for a return from a subroutine call. Whenever a write is performed to the call register code, the iprs register is loaded with the instruction sequentially following the second delay slot instruction of the call (the two delay-slot instructions are executed before the branch to the subroutine address occurs). When nesting multiple calls, you must save the iprs register on the stack in order to preserve the return address of the prior call. To restore a previously saved value, you can write to the iprs register. To return from a call, move the contents of the iprs register back into the pc register, or load directly off the top of the stack the iprs value that was saved earlier.

In the FAE pipeline, the ipa and ipe registers record the program counter history, as shown in Figure 5–3.

On each cycle in which the pipeline advances (that is, there is no stalling condition), the contents of the ipa register are moved to the ipe register, and the contents of the pc register are moved to the ipa register.

As described in Section 10.1, *Software Branching*, many data unit and address unit operations can specify the pc register (using either the call or br register code) as the destination. This is the way that subroutine calls and branches are performed. Since both data unit and global address unit operations can be made conditional, either the data unit or a global address unit operation can perform a conditional branch.

Figure 5–3. Program Counter History



5.2.2 Pipeline Implications

The PP's FAE pipeline is designed for fast execution of streams of data that are processed in a tight loop. Data results from one instruction can be used or stored by the next instruction. One instruction can load data from memory to a register, the next instruction can operate on the data, and the following instruction can store the result from a register to memory. Because each PP instruction can define parallel operations, and because zero-overhead looping is supported, three operations like these can be performed in one instruction.

For example, multiply-accumulate operations flow well through the pipeline. With a set of data stored in each of two different RAMs, in a single instruction cycle, the PP can fetch two pieces of data to be multiplied in the next instruction, multiply the data fetched in the previous instruction, and add/accumulate results from the previous instruction's multiply. The pipeline makes it possible for this same instruction to be executed repetitively in a zero-overhead loop with a net throughput of one multiply accumulate every cycle.

Unlike the MVP's master processor, no register scoreboarding is performed on the PP.

The major pipeline effects and characteristics are discussed in the following subsections. These conditions hold true, regardless of any stalls in the pipeline due to events such as instruction cache misses or memory access contention. Stalls simply freeze the pipeline until the stall condition is removed.

5.2.2.1 Modifying an Address or Index Register

One of the consequences of the FAE pipeline is that whenever an address or index register is modified by the data unit or loaded by the address unit (both occur in the execute pipeline stage), that register **cannot** be used in the next instruction for address computations (which occur in the address stage). **If this rule is violated, the result of the address computation will be unpredictable**; either the old or new value may be used in the address computation, depending on whether an interrupt occurred or you are single-stepping or running full speed.

- If an interrupt occurs, the execute stage of the instruction modifying the address or index register occurs during the interrupt routine. When the return from interrupt occurs, the address computation of the next sequential instruction is performed using the **new** value of the address or index.
- If no interrupt occurs, the execute stage of the instruction writing to the address register occurs in parallel with the address stage of the next instruction. Thus, the address computation is based on the **old** value of the address or index.

You might think that if you disable interrupts to ensure that the old value of the register is used, you would be safe. However, if you use the debugger and single-step through your code, the new value is used. In summary, simply allow one delay-slot instruction between an instruction that modifies an address or index register in the execute stage and an instruction that uses that register for an address computation.

5.2.2.2 Branch Delay-Slot Instructions

Another consequence of the three-stage FAE pipeline is that two delay-slot instructions are associated with a branch or subroutine call. After an instruction that specifies a branch or call is fetched, two more instructions are fetched before the branch or call is executed. Unlike the master processor, the PP does not support an annul option. Due to possible prior branches or hardware looping, the two delay-slot instructions for a branch are not necessarily the next two instructions in memory.

In most cases, you can structure your code to fill the delay slots with two useful instructions. For example, for subroutine calls, you can use the delay-slot instructions to set up arguments and/or push registers onto the stack. For subroutine returns, you can use the delay-slot instructions to restore registers by popping the registers' previous contents off the stack. For circumstances in which no useful work can be done, you can fill a delay-slot instruction with an instruction that does nothing (nop).

For two primary reasons, PP code tends to require fewer software branches (and their associated delay slots) than most other processors.

- First, the PP supports up to three levels of hardware-controlled looping. Hardware looping operates in the fetch stage of in the pipeline so it does not have any associated delay slots for the hardware loop. (Note that the looping-related registers must be loaded at least two instructions before a corresponding loop end address is encountered.)
- Secondly, the PP's support of conditionally executed instructions can be used to avoid conditionally branching around a few instructions.

5.2.2.3 Summary of Pipeline Implications

The major pipeline effects and characteristics discussed in this subsection are summarized below. These conditions hold true, regardless of any stalls in the pipeline caused by events such as instruction cache misses or memory access contention. Stalls simply freeze the pipeline until the stall condition is removed.

- Registers loaded in one instruction can be used as data in the next instruction.
- Data results from one instruction can be stored by the next instruction.
- Because the address computations occur ahead of the execute stage, you must allow one delay slot instruction before using an address or index register that is changed by the data unit or by a load from memory. Unpredictable behavior will result if this condition is not met. The assembler cannot ensure that this condition is met.
- The two instructions fetched following a branch or call (the delay-slot instructions) are always executed, regardless of whether the branch is taken or not. The delay-slot instructions are not necessarily the next two sequential instructions if there are other branches or loops.
- Zero overhead loops have no delay-slot instructions. The loop controllers work in the fetch pipeline stage to eliminate delay slots.
- Because the loop-controlling registers work in the fetch stage of the pipeline, loop control registers must be loaded at least two instructions before the corresponding loop end address is encountered.

5.2.3 Pipeline Stall Conditions

Several events can cause the normal instruction fetching operations to cease temporarily. These events, referred to as pipeline stall conditions, are discussed briefly in this subsection. When any of these stall conditions occur, the program flow control unit instructs the address units and data unit to stall. Therefore, the program flow control unit is considered to be master over the other two units. Program flow resumes, once the stall condition is cleared.

These are examples of pipeline stall conditions:

- Cache miss

  Cache misses are detected by the cache control logic, and the transfer controller is automatically signaled when one occurs. The information that the TC needs is supplied by the program flow control unit to allow the TC to service the cache. The PP stalls until its cache request has been serviced.

- Illegal operation detection

  An unimplemented opcode, fetched during the previous fetch cycle, is decoded, causing the pipeline to stall and an interrupt request to be sent to the MP. The PP remains stalled until the MP sends a reset command.

- Crossbar contention

  If one or both of the local and global data ports experience contention with another processor when requesting a memory access, the program flow control unit and data unit stall. The address units also stall, although they continue to request access to the memory. Once contention is resolved, the program flow control unit releases all units to normal operation.

- Local port address miss—diversion to global port

  When a local transfer is performed in parallel with a global transfer, the local transfer access is attempted over the local port. If the local transfer specifies an access to an address that is not in the PP's local RAM, the transfer cannot occur over the local port, and the pipeline stalls. Once the global transfer has completed (thus freeing up the global port), the local transfer is diverted automatically to the global port, and the access is completed.

- Global port address miss—DEA (direct external access)

  If the address unit detects that it has generated an address that is not in the shared RAMs, then the pipeline stalls and a DEA request is sent to the TC. This is serviced in a manner similar to a cache miss. It finally completes like a normal memory access.

- Halt request

  The MP or the PP itself can specifically halt the operation of a PP under software control. This causes the pipeline to stall until the MP unhalts the PP.

5.3 Loop Controllers

The program flow control unit has three sets of zero-overhead loop controllers. Each controller has a register for the loop start address (ls0 – ls2), loop end address (le0 – le2), current loop count (lc0 – lc2), and loop reload count (lr0 – lr2). A loop control register (lctl) specifies which (if any) loop controllers are active.

5.3.1 Hardware-Looping Mechanism

For a hardware loop, a comparator detects when the pc register is fetching the instruction at the loop end address.

- If the pc register points to the loop end and the loop count is nonzero, then the loop controller loads the pc register with the address in the loop start register to cause the next instruction to be fetched from the top of the loop; also, the loop count is decremented by 1.
- If the pc register points to the loop end address and the loop count is 0, then the pc register is allowed to increment to point at the next sequential instruction (thus exiting the loop). Also, the loop reload count is copied into the loop count register to set up the next pass through the loop (for example, in the case of a hardware loop nested inside another loop).

5.3.2 Multiple Nested Loops

Multiple nested hardware loops with a common end address are supported by prioritizing the three loop controllers. Loop controller 2 has the highest priority (and thus controls the innermost loop), and loop controller 0 has the lowest priority (thus controlling the outermost loop).

When the pc register points to the common end of loop, the highest priority loop start address register with a nonzero loop count is copied into the pc register. If, at the common end address, a higher priority loop counter is 0, that loop counter is reloaded by its corresponding loop reload register. If all loop counters are 0 at the loop end address, then the pc register increments and all the corresponding loop counters are reloaded.

5.3.3 Hardware Branching

In addition to specifying which, if any, loop controllers are active, the lctl register associates a loop counter register with each loop end. Using a loop controller with no associated loop counter essentially sets up a hardware-controlled branch. The loop end register (le*n*) specifies where the branch will occur, and the loop start register (ls*n*) specifies where to branch to. Since hardware detects the branch conditions, these branches occur with no overhead, once the loop start and end registers are set up.

Hardware branching can be particularly useful for supporting run-time options. The address that is loaded into the loop start register can point to the option. For example, graphics has different pixel-processing options for algorithms such as line drawing and BitBLT that are often selected at runtime. With the three sets of loop controllers, hardware branches can be combined with hardware looping; one controller can be used to control the loop, a second controller the branch out (to the option), and the third controller the branch back. This example is described more thoroughly in subsection 15.1.3, *Pixel Block Transfers (PIXBLTs)*.

You can use the loop controllers to perform zero-overhead conditional branches by loading the loop counter (lc*n*) register with either a zero or nonzero value before the corresponding loop end address is reached. If the value loaded into the loop counter is nonzero, then the program will branch to the loop start address. Furthermore, this technique of zero-overhead conditional branching can be combined with zero-overhead looping; a higher priority controller is used for the conditional branch, and a lower priority controller is used to control the loop.

5.3.4 Software Branching Within a Loop

Software branches or calls (that is, software writes to the pc register) take precedence over the hardware loop controllers when the branch or call instruction is executed in the same cycle in which an instruction at a loop end address is fetched (that is, the instruction at the loop end address is the second delay-slot instruction of a branch or call). When the pc register is written to by software, loop ends are ignored and loop counts are not updated. This allows loops to be exited conditionally (regardless of the loop count).

When you place loop ends in conditional branch delay slots, take special care in order to get predictable behavior. Section 11.1, *Looping*, discusses this, gives several examples, and contains a more detailed description of the loop controllers operation and capabilities.

5.4 Cache Controller

The instruction cache controller inside the program flow control unit contains the address tag registers, address tag comparators, present flags, least recently used (LRU) stack, and control hardware for managing the PP's instruction cache.

As discussed in subsection 3.1.1, *Cache Architecture*, the instruction cache RAM contains 2K bytes or 256 64-bit doubleword instructions, broken into four sets of 64 doubleword blocks. Each block in turn is broken into four sub-blocks of 16 doublewords each.

Each block has a tag register that contains the 23 MSBs of the address for the instructions held in that block of the cache. The tag comparators associated with each block compare the address of the instruction being fetched with the tag register to see if the address is in the range of the block. Each time an instruction is fetched from a block, that block becomes the most recently used and is put at the back of the LRU stack.

Each of the subblocks has a present flag to indicate whether the instructions in that subblock have actually been transferred from off-chip memory into the cache. Whenever there is a cache miss (the instruction being fetched is not in the cache), a cache service request is submitted to the transfer controller (TC) to transfer the entire subblock (16 instructions). Thus, the PP caching method always pulls in 16 instructions when the next instruction needed is not already present in the cache. This reduces the number of cache misses and makes better use of the available off-chip bandwidth. After the subblock has been brought into the cache, the subblock's present flag is set and the PP resumes instruction execution.

A more detailed explanation of the PP's instruction cache operation is given in Chapter 3, *PP Instruction-Cache Operation and Interprocessor Communications*.

Chapter 6

Understanding the Address Units

The global and local address units compute addresses for memory accesses and control all register movement on the global source, global destination, and local destination/source buses for the PP. This chapter describes the address unit hardware, the differences between the two address units, and data flow and control by the address units.

Topics

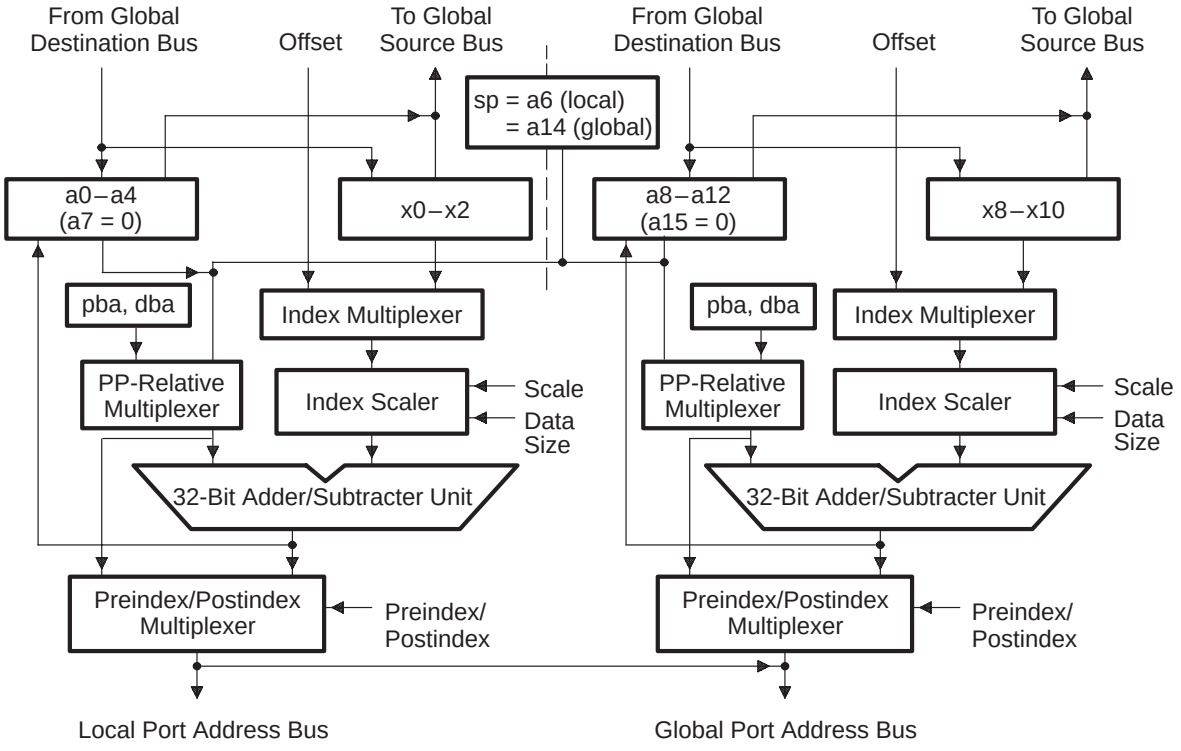
6.1	Overview of the Address Units	PP: 6-2
6.2	Bus Structure	PP: 6-6
6.3	Differences Between the Global and Local Address Units	PP: 6-11
6.4	Data Flow and Control	PP: 6-13

6.1 Overview of the Address Units

The PP has two address units: the **global address unit** and the **local address unit**. These units operate independently of each other. Each unit is responsible for both computing addresses and accessing memory. Section 8.8, *Parallel Transfers*, provides a software-oriented discussion of the operations supported by the address units.

Figure 6–1 shows the two address units. These address units support 12 different addressing modes. An addressing mode for each address unit is specified directly in the opcode. The addressing mode controls the operations performed by the address unit hardware.

Figure 6–1. Address Units Block Diagram



6.1.1 Registers

Each address unit has five address registers and three index registers.

- The local address unit contains registers identified by numbers in the range 0 through 7:
  - Five address registers (a0–a4)
  - Three index registers (x0–x2)
  - One zero-valued address register (a7)
  - A data path for computing addresses
- The global address unit contains registers identified by numbers in the range 8 through 15:
  - Five address registers (a8–a12)
  - Three index registers (x8–x10)
  - One zero-valued address register (a15)
  - A data path for computing addresses

A single stack pointer (sp) is mapped into both register files (a6 for the local address unit and a14 for the global address unit). Local address unit register a7 and global address unit register a15 always read as the constant 0; writing to these two registers has no effect.

6.1.2 Index Multiplexer

All 12 addressing modes use indexed addressing. The index can be specified either in an index register or directly in the instruction as an immediate offset. As shown in Figure 6–1, the index multiplexer, controlled by the addressing mode, selects between an index register and immediate offset. The immediate offset can be either a short offset or long offset, depending on the operations being specified in parallel in the instruction.

- A **short offset** is 3 bits for halfword or word transfers and 4 bits for byte transfers.
- A **long offset** is 15 bits for halfword or word transfers and 16 bits for byte transfers.

6.1.3 Index Scaler

You can scale an index by the data size before combining it with the address register value. For a halfword access, the index can be shifted left by one bit. For a word access, the index can be shifted left by two bits. Index scaling is frequently used to perform a table look-up. When you use an immediate offset, index scaling allows a wider access range for both short and long offsets.

6.1.4 Adder/Subtracter Unit

The adder/subtracter unit performs a 32-bit addition or subtraction between the base address contained in an address register and an index. The operation performed by the adder/subtracter unit is specified by the addressing mode in the instruction. The addressing mode determines whether or not the address register is updated with the result of the adder/subtracter unit.

6.1.5 Preindex/Postindex Multiplexer

The addressing mode controls a preindex/postindex multiplexer that selects either the base address (the address register's original contents) or the result of the addition or subtraction to send out over the address bus. This allows either preindexed or postindexed addressing to be performed.

6.1.6 PP-Relative Multiplexer

The PP-relative multiplexer replaces the value read from an address register with a PP-relative base address. The PP-relative base address is hardwired into each PP and is unique for each PP. There are two PP-relative base addresses: pba and dba.

- pba is the base address of a PP's local parameter RAM.
- dba is the base address of a PP's local data RAM0.

You can tell the PP-relative multiplexer to use the PP-relative base address by specifying pba or dba in the assembly instruction. Like a normal addressing operation, you can specify pba or dba with any of the twelve addressing modes. If you specify pba or dba with a pre/post modify of the address register, the address register will be initialized with a PP-relative address. Subsequent normal memory accesses with that register will be PP-relative without the need to specify pba or dba.

6.1.7 Right-Align/Sign Extend Hardware

Both the local and global ports are associated with hardware that right-justifies byte (8-bit) and halfword (16-bit) quantities on loads and replicates byte and halfword quantities on stores. On byte or halfword loads, this hardware will either zero- or sign-extend the quantity to 32 bits after it has been right-justified. On stores, byte or halfword data will be replicated to fill 32 bits, and the PP will indicate which copy of the data is to be written.

6.1.8 Associated Bus Structure

As discussed in detail in Section 6.2, each PP has three ports to memory: the instruction port, global port, and local port.

- The instruction port is 64 bits wide and is dedicated to the instructions being fetched by the program flow control unit.
- Both the global and local ports are 32 bits wide for loading and storing PP data.
  - The global port has crossbar connections to all of the shared RAMs on the chip.
  - The local port has crossbar connections to only the four RAMs that are local to the given PP.

6.2 Bus Structure

Several buses are closely associated with the two address units. These buses transfer data both to and from memory and between the various PP functional units. These buses and their associated hardware are integral to the function of the address units but are distributed within the PP. Additionally, buses associated with the program flow control unit provide instruction addresses and receive instructions.

The PP's major bus structure is shown in Figure 1–1, *The PP Block Diagram*. The major buses associated with the address units are:

- **Local address port**

  On this bus, the local address port can send addresses to the PP's associated local RAMs but not to nonlocal RAMs. Local address unit accesses go to the crossbar via the local address port if there is a parallel global address unit operation. Attempted accesses to nonlocal RAMs over the local address port stall and are automatically diverted to the global address port when it becomes available.

- **Global address port**

  On this bus, the addresses generated by either the local address unit or global address unit can reach any crossbar RAM. If there is no parallel global address unit transfer, a local address unit transfer will occur automatically over the global port instead of the local port; no stall will occur due to this crossover.

- **Local data port**

  This data port bus is used to transfer data in from or out to the address specified by the local address port.

- **Global data port**

  This data port bus is used to transfer data in from or out to the address specified by the global address port. Also, this bus can be used to connect the global source data bus to the global destination data bus for register-to-register moves. When a local transfer takes place over the global address port, the global data port is used instead of the local data port.

- **Local destination/source**

  This bus is used to transfer data in either direction between the local or global data port and its associated hardware and the D registers (d0–d7) in the data unit.

  Note that in Figure 1–1, Lds represents the local destination/source bus.

- **Global source**

  This bus is used to transfer data from a non-D register to the data unit's data path or from any PP register to the hardware associated with the global data port.

  Note that in Figure 1–1, Gsrc represents the global source bus.

- **Global destination**

  This bus transfers data either from the output of the data unit's ALU to a non-D register or from the hardware associated with the global data port to any PP register.

  Note that in Figure 1–1, Gdst represents the global destination bus.

- **Instruction address port**

  This bus sends addresses that have been translated by the instruction cache controller to the instruction cache RAM.

- **Instruction port**

  This bus receives instructions from the instruction cache.

Note:

The global address unit operation always uses the global address port bus, the global source bus, and global destination bus; the local address unit can also use these buses when there is not a parallel global address unit operation. Consequently, these buses are not as dedicated as their names may imply.

Because the global address port and global data port bus can be used to access any of the shared RAMs, they are used by the local address unit whenever it needs to access memories that are not local to the PP.

When an instruction specifies both a global address unit operation and a local address unit operation, the global address unit is given the global data port and the global address port, while the local address unit must first try to access the address via the local port. If the address specified by the local address unit is not in the local RAMs, the instruction pipeline stalls until the global address unit's access is complete; then the local address unit's access is made over the global address port and global data port.

If there is no global transfer operation specified, the local transfer will be attempted first over the global port, without a stall.

6.2.1 Hardware for Operations on Data Less Than 32 Bits

Both the local and global ports are associated with hardware that allows operations on data that is less than 32 bits.

- **The replicate hardware (Repl)** replicates bytes or halfwords four or two times, respectively, to fill out a 32-bit word. This hardware is used in combination with write strobes to write to the desired byte or halfword within a 32-bit word in memory.

  The replicate hardware connected to the global source bus can also be used to replicate bytes or halfwords for register-to-register moves.

- **The right-align and sign-extend or zero-extend hardware (A/S)** right-justifies and either sign- or zero-extends bytes or halfwords to fill a 32-bit word. The A/S hardware can be used by either loads from memory or register-to-register moves.

6.2.2 Buses Associated With D Registers

The local destination/source bus is used to load and store D registers via the local data port and its associated hardware under control of the local address unit. Alternately, it can be used to write the local address unit arithmetic result to a D register.

- As shown in Figure 1–1, the local destination/source bus can store D register data via the hardware associated with the global data port. The local destination/source to global data port connection is used when an instruction with no global address unit operation specifies a store by the local address unit.
- Similarly, the global data port to local destination/source bus connection is used when you use an instruction to load data to a D register and no parallel global address unit operation is specified.

6.2.3 Buses Associated With All Registers

The global destination and source buses connect to PP registers. Single-cycle register-to-register moves read the source register via the global source bus through the replicate hardware onto the global data port, and write to the destination register via the right-align and sign-extend (A/S) hardware and the global destination bus.

Instead of addressing memory, you can write the outputs of the global and/or local address ports as data to any PP register (referred to as address unit arithmetic). As shown in Figure 1–1, the global address port bus can be routed to the global destination bus, and the local address port bus can be routed to the local destination/source bus. The global address unit computation can be written to any PP register.

If there is no parallel global address unit operation and the data unit operation does not use a non-D register operand, the result of the local address unit computation can be written to any PP register via the global address port bus.

If a parallel global address unit operation is performed or the data unit uses non-D operands, the local address unit source register for stores or destination register for loads or address unit arithmetic must be a D register because the transfer of data occurs via the local address port and local destination/source buses.

6.3 Differences Between the Global and Local Address Units

Figure 6–1 shows the two address units. For the most part, the address units are identical. They each have the same data path and support the same addressing modes. The key differences between the two units are summarized in the following subsections.

6.3.1 Address and Index Registers

Local address unit addressing operations use local address and index registers (a0–a4, a6–a7, x0–x2), while global address unit addressing operations use global address and index registers (a8–a12, a14–a15, x8–x10).

6.3.2 Accessing Memory in the Same Instruction

In spite of its name, the local address unit can access any of the on-chip shared RAMs. However, when two parallel transfers are specified in the same instruction (that is, both address units are used), it is more efficient if the local address unit addresses a local RAM.

When there are two parallel transfers, if the local address unit specifies a nonlocal address, then the instruction requires two cycles to complete (assuming no other stall conditions exist). Even if the global address unit is accessing a local RAM while the local address unit is accessing a global RAM, the instruction execution will be delayed.

Since two parallel transfers are frequently specified in a single instruction, to improve efficiency, you can adopt programming conventions that keep data that is needed by the local address unit in the local RAMs.

6.3.3 Performing Parallel Operations

Only the local address unit can operate in parallel with data unit operations that use non-D register operands. When non-D registers are needed by the data unit's ALU operation, the instruction opcode bits and buses needed for the global address unit operation are used to support non-D registers going to and coming from the data unit.

Additionally, when a data unit and/or global address unit operation is performed conditionally, no local address unit operation can be specified in parallel.

6.4 Data Flow and Control

The address units and their associated instruction opcodes control the loading and storing of data and the movement of data within the PP. This section describes the flow of data controlled by the address units and includes the following topics:

- Memory accesses (loads and stores)
- Register-to-register moves
- Address unit arithmetic

6.4.1 Memory Accesses (Loads and Stores)

The address units support two types of memory accesses:

- Loads (memory-to-register transfers)
- Stores (register-to-memory transfers)

For both types of memory accesses, the address units use 1 of 12 addressing modes to compute the address in memory that is to be accessed. Addresses are computed during the address stage of the FAE (instruction) pipeline (see Section 5.2, *FAE Instruction Pipeline*, for more information about pipeline stages). Then, the actual memory access (that is, the fetching or storing of data) occurs during the execute stage of the instruction.

6.4.1.1 Generating Addresses

Referring to Figure 6–1, during the address stage of the FAE pipeline, the address in the specified address register or the pba/dba value is added to an index specified in an index register or to a short (3- or 4-bit) or long (15- or 16-bit) immediate offset in the instruction. The index scaler can optionally scale the index according to the data size. A full 32-bit result is computed by adding the output of the index scaler to the specified address register. In this regard, both address units operate the same, except that they use different sets of registers.

The output of the adder/subtracter unit is routed both to a multiplexer and back to the address registers (including the stack pointer). The multiplexer selects either the adder/subtracter unit's result (in the case of preindexing) or the original address register value (in the case of postindexing) to send out over the associated address bus. Optionally, the result from the adder/subtracter unit can be written back to the address register in order to step through memory on successive operations.

In the same cycle that addresses are computed, the addresses are sent out over either the global address port or local address port. During this cycle, crossbar connections are determined, and any contention issues are resolved. Note that data is not actually transferred at this time; the actual transfer of data does not occur until the execute stage of the pipeline.

When the global address unit generates an address, it always uses the global address port. The local address unit also accesses memory via the global address port, as long as there is not a parallel global address unit operation.

If the local address unit cannot use the global address port because it is being used by the global address unit, then the local address unit will try to address memory via the local address port. The local address port is, however, restricted to accessing only the PP's local RAMs. If it is detected that the address specified is not to one of the local RAMs, the FAE pipeline stalls until the global address port, global destination bus, and global source bus are free (generally due to the global address unit finishing its operation). Once the needed buses are free, the local address unit access is diverted to the global address port, and the memory access occurs.

6.4.1.2 Transferring Data

During the execute stage of the FAE pipeline, data is transferred between memory and the register (or vice versa). Loads that use the global address port for the address bring in data that is read from the RAM via the global data port and global destination bus. Stores that use the global address port for the address will send out data from a register via the global source bus and through the global data port to memory. Loads and stores that use the local address port for the address use the local data port and local destination/source bus to transmit the data in either direction.

To the PP, the RAMs on the MVP appear to be 32 bits wide with write enable per byte. The PP has hardware that creates the effect of byte and halfword transfers.

Data going from a register to memory via either the local destination/source or global source bus passes through replicate hardware. On byte and halfword stores, the rightmost byte or halfword quantity is replicated to fill out 32 bits by the replicate hardware. The replicated value is then sent to the memory via the local data port or global data port with write strobes enabling only the specific byte or halfword specified by the address to be written.

Byte and halfword data being loaded from memory pass though the align and sign-extend hardware.

- The align hardware right-shifts the data specified by the address.
- The sign-extend hardware can then, as specified by the instruction, either zero-extend or sign-extend the right-aligned quantity to fill 32 bits.

Note that while not explicitly shown, both the global source and global destination buses can access any of the PP's registers, while the local destination/source bus can access only the D registers. Thus, data can be loaded or stored between any register and memory when the global source and global destination buses are used.

For instructions that perform operations in both the global address unit and local address unit, the local address unit source (for stores) or destination (for loads) must be a D register. This is both an instruction opcode limitation and a bus structure limitation because the local destination/source bus connects the local data port only to the D registers.

You can make loads and stores by the global address unit conditional by omitting the parallel local address unit operation. To do so, the instruction opcode bits normally associated with the local address unit specify the conditional nature of the load or store operation. The conditional operation can select between a pair of source registers on the basis of the negative status register bit for a store operation. For loads, the actual latching of the data accessed from memory can occur conditionally according to any of the 15 condition codes.

6.4.1.3 Resolving Contention

Contention occurs when more than one address unit from either the same processor or different processors try to access the same RAM in a cycle. In normal operation, a hardware-controlled round-robin method keeps rotating the priority of the all the ports of all the processors. Only the current highest priority access is allowed, and the other processors with contending accesses must stall. On each cycle, the contention for each RAM is re-evaluated so that as a processor completes its access, another processor can have access to the RAM on the next cycle. Because of the round robin prioritization, all accesses are eventually allowed.

The crossbar's round-robin prioritization scheme prevents any PP from dominating a given RAM. A fixed priority mode is also supported but is primarily intended for debugging code. Once a processor is granted access to the RAM, it can continue instruction execution, even though other processors may still be stalled, waiting for their turn to access the RAM.

6.4.1.4 Accessing Off-Chip Memory

While accesses to off-chip memory are slower than accesses to on-chip memory, either address unit can access off-chip memory. If an address unit attempts to access an address that is beyond the range of on-chip memory, a high-priority request is automatically submitted to the transfer controller to perform the memory access. This type of access to off-chip memory is referred to as a direct external access (DEA).

Table 6–1 shows the minimum DEA latency for a PP access to various types of off-chip memory. These minimum times assume that the access is attempted initially on the global port and that the TC is immediately available to service the DEA. If the access is attempted initially on the local port, the minimum latency requires an additional cycle because the access is first diverted to the global port before a DEA request is issued. The latency indicates the number of cycles that the instruction specifying the DEA spends in the execute stage of the pipeline. It includes the first execute cycle; therefore, the number of cycles during which the PP pipeline is stalled is the latency minus 1.

Table 6–1. Minimum DEA Latency

		Number of Cycles	
Access Type	**Memory Type**	**Page Hit**	**Page Miss**
Store	Any Type of Memory	8	12
Load	1 Cycle/Column Unpipelined	11	15
	1 Cycle/Column Pipelined	12	16
	2 Cycles/Column	11	16
	3 Cycles/Column	12	18

The latency for a DEA store is less than for a DEA load. For a load, the PP's pipeline remains stalled until the TC has read the data at the specified address in off-chip memory and loaded that data into the specified destination register. For a store, the PP's pipeline is stalled only until the TC has ensured that the write to off-chip memory will be able to take place. Thus, the pipeline is stalled fewer cycles than for a load because it does not have to wait until the data has actually been written to off-chip memory.

DEAs are generally used only when very little data is being transferred (typically one or two accesses). In most other cases, it is more efficient to use packet transfer requests to transfer data between on-chip and off-chip memory. Often, a packet transfer request can be submitted far ahead of when the data is actually required. Then, when the data is required, the PP needs to verify only that the packet transfer has completed. Then the PP can continue processing, and the FAE pipeline never has to stall. Packet transfers are described in more detail in Chapter 12, *Packet Transfers*, in this user's guide and in Chapter NO TAG, *Packet Transfers*, in the *MVP Transfer Controller User's Guide.*

6.4.2 Register-to-Register Moves

The instruction opcode bits and the data paths associated with the global address unit can perform a register-to-register move between any of the PP registers. Additionally, for moves specifying a D register source, the replicate, align, and sign-extend hardware can perform a number of different operations referred to as field moves.

- A **field extract move** allows any byte or halfword in the source D register to be extracted. This data is then right-justified and either sign- or zero-extended to form a 32-bit word. The result is written to the specified destination register.
- A **field replicate move** allows the LSbyte or LShalfword in the source D register to be replicated to fill a 32-bit word and then to be written to the destination register.

The following paragraphs describe the hardware and data paths used to perform these operations. For this discussion, it may be helpful to refer to Figure 1–1.

Data is read from the source register via the global source bus and routed through the replicate hardware. The output of the replicate hardware goes to the right-align and sign-extend (A/S) hardware via the global data port. The output of the A/S hardware then sends the data to the global destination bus, which can route the result to any PP register. The instruction can specify an individual byte or halfword in a register for the A/S hardware to extract, right-align, and either zero- or sign-extend to fill out 32 bits. Alternatively, the instruction can specify replications of the LSbyte or LS halfword in the replicate hardware.

In parallel with a move operation, the local address unit can perform an independent memory access by using the local address port, local data port, and local destination/source buses. If the address generated by the local address unit requires a nonlocal RAM, then the execution will stall. As soon as the global unit's move is completed, the local address unit's memory access can be completed via the global address port, global data port, and either the global source or global destination bus.

If you don't specify a parallel local address unit operation, you can make the move occur conditionally. In this case, the opcode bits normally used to specify the local address unit operation are used to specify conditional operation.

6.4.3 Address Unit Arithmetic

In place of performing a memory access, either or both of the address units can perform an address computation that is written directly to a PP register instead of being used for a memory access. This capability, referred to address unit arithmetic, supports additional arithmetic operations in a single instruction for compute-intensive algorithms.

An address unit arithmetic operation is very similar to a load. The address computation is performed by the address unit in the same manner as for a load, and the result is sent to either the local address port or the global address port. Referring to Figure 1–1, the global address port can be input through A/S (align/sign-extend hardware) logic to the global destination bus, and the local address port can be input via A/S logic to the local destination/source bus (in both cases, the A/S logic simply passes the data through). The data on the global destination bus can be written to any PP register, while the data on the local destination/source bus can be written only to a D register. Therefore, the result of the address computation (that is typically used to address memory) can instead be written directly to a PP register in the execute stage of the pipeline.

The result of any computation that can be made to generate an address (including optional index scaling) can be saved to a register in this way. The address register used in the computation can be updated by the address computation result in the address stage of the pipeline. Therefore, the address computation result can be written to both an address register in the address stage and another PP register in the execute stage.

Address unit arithmetic capabilities and restrictions for parallel operations are similar to those of memory transfers. If the global address unit is not performing an operation, then the local address unit can use the global address port, global data port, and global destination buses to save its result to any PP register. If, however, the global address unit performs an operation or if the data unit has a non-D operand, then the destination of the local address unit's arithmetic operation is limited to a D register. You can use the opcode bits normally used by the local address unit to write the result of a global address unit arithmetic operation conditionally to the destination. The address register update (if specified by the addressing mode) always occurs, even if the write to the destination is conditional.

Chapter 7

Summary of PP Registers

This chapter provides a detailed description of the PP's user-accessible registers, including D and non-D registers and upper and lower registers. It also contains information on the PP register codes.

Topics

7.1	**The PP Register Codes**	**PP: 7-2**
7.2	**Alphabetical Summary of the PP Registers**	**PP: 7-4**

7.1 The PP Register Codes

The PP register codes are shown in Table 7–1. The bulleted text immediately following explains terms and symbols used in the table.

Table 7–1. The Register Codes

Lower Registers (MSB of Bank = 0)						Upper Registers (MSB of Bank = 1)					
Coding			Coding			Coding			Coding		
Bank	Reg	Register	Bank	Reg	Register	Bank	Reg	Register	Bank	Reg	Register
0000	000	a0	0100	000	d0	1000	000	reserved	1100	000	lc0
0000	001	a1	0100	001	d1	1000	001	reserved	1100	001	lc1
0000	010	a2	0100	010	d2	1000	010	reserved	1100	010	lc2
0000	011	a3	0100	011	d3	1000	011	reserved	1100	011	reserved
0000	100	a4	0100	100	d4	1000	100	reserved	1100	100	lr0
0000	101	reserved	0100	101	d5	1000	101	reserved	1100	101	lr1
0000	110	a6 (sp)	0100	110	d6	1000	110	reserved	1100	110	lr2
0000	111	a7 (zero)	0100	111	d7	1000	111	reserved	1100	111	reserved
0001	000	a8	0101	000	reserved	1001	000	reserved	1101	000	lrse0
0001	001	a9	0101	001	sr	1001	001	reserved	1101	001	lrse1
0001	010	a10	0101	010	mf	1001	010	reserved	1101	010	lrse2
0001	011	a11	0101	011	reserved	1001	011	reserved	1101	011	reserved
0001	100	a12	0101	100	reserved	1001	100	reserved	1101	100	lrs0
0001	101	reserved	0101	101	reserved	1001	101	reserved	1101	101	lrs1
0001	110	a14 (sp)	0101	110	reserved	1001	110	reserved	1101	110	lrs2
0001	111	a15 (zero)	0101	111	reserved	1001	111	reserved	1101	111	reserved
0010	000	x0	0110	000	reserved	1010	000	reserved	1110	000	ls0
0010	001	x1	0110	001	reserved	1010	001	reserved	1110	001	ls1
0010	010	x2	0110	010	reserved	1010	010	reserved	1110	010	ls2
0010	011	reserved	0110	011	reserved	1010	011	reserved	1110	011	reserved
0010	100	reserved	0110	100	reserved	1010	100	reserved	1110	100	le0
0010	101	reserved	0110	101	reserved	1010	101	reserved	1110	101	le1
0010	110	reserved	0110	110	reserved	1010	110	reserved	1110	110	le2
0010	111	reserved	0110	111	reserved	1010	111	reserved	1110	111	reserved
0011	000	x8	0111	000	pc/call	1011	000	reserved	1111	000	reserved
0011	001	x9	0111	001	ipa/br	1011	001	reserved	1111	001	reserved
0011	010	x10	0111	010	ipe#	1011	010	reserved	1111	010	reserved
0011	011	reserved	0111	011	iprs	1011	011	reserved	1111	011	reserved
0011	100	reserved	0111	100	inten	1011	100	reserved	1111	100	tag0#
0011	101	reserved	0111	101	intflg	1011	101	reserved	1111	101	tag1#
0011	110	reserved	0111	110	comm	1011	110	reserved	1111	110	tag2#
0011	111	reserved	0111	111	lctl	1011	111	reserved	1111	111	tag3#

❑ Bank

Bank identifies the bank (or type) of register. For example, local address registers all have a bank code of 0000.

The bank code for certain operands is implied. For example, the base address register for a local transfer is, by definition, a local address register; thus, only the three LSBs of the register code are required to specify the register in the instruction word.

- **Reg**

  The three LSBs of the register code specify a register within a bank of registers. For a non-D register, these values also indicate the companion D register. For example, the three LSBs for a9 are 001; thus, the companion D register is d1.

- **pc/call**

  This register code is used to read the pc register. It can also be used to write to the pc register with an associated saving of a return address into the iprs register; thus, it is referred to as the call register code because it can be used to perform subroutine calls.

- **ipa/br**

  This register code is used to read ipa (a read-only register). It can also be used to write to the pc register without an associated saving of a return address into iprs; thus, it is referred to as the br register code because it can be used to perform branches.

- **#**

  This symbol indicates read-only registers.

Note:

Throughout the *MVP Parallel Processor User's Guide*, the term **PC** refers the to the 29-bit counter visible in the pc register. The phrase **pc register** refers to the entire 32-bit register that contains both the PC and the G and L control bits.

7.2 Alphabetical Summary of the PP Registers

The rest of this chapter describes the parallel processor's individual registers.

In the description of each register, the register or group of registers is qualified as either D or non-D and as either lower or upper. Also, the register code for each register and the corresponding companion D register are listed. The register format is given, followed by a description of the register and each individual bit field.

Two symbols are used for designating reserved bits:

- The letter **r** indicates bits that are reserved for an assigned function for possible future versions of the MVP with eight PPs.
- The **–** character indicates reserved bits that have no assigned function.

Future functions configured by bits that are currently reserved will be enabled by a value of 1. Thus, the following compatibility guidelines are suggested so that code retains the same functionality on future versions of the MVP.

- During the PP's initialization sequence after reset, reserved bits should be set to 0. Note that at reset, the currently reserved bits are not reset to any particular value.
- If it is necessary to write to any of the reserved bits in the process of setting currently implemented fields in a register, you should use a value of zero.
- If a register's previous value is popped off the stack to restore the register, you can assume that the appropriate values (which may not all be 0 in the future) are written to the reserved bits.

Name Local address registers: a0–a4

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code
a0: 0000 000
a1: 0000 001
a2: 0000 010
a3: 0000 011
a4: 0000 100

Companion Regs
a0: d0
a1: d1
a2: d2
a3: d3
a4: d4

Format

31	0

Description The a0, a1, a2, a3, and a4 registers are general-purpose 32-bit read/write address registers in the local address unit. Although they are typically used for address generation purposes, you can also use these registers as non-D register ALU operands.

Local address registers are typically used for addresses that can be accessed over the local port. However, it is possible to use local address registers for memory accesses that require the global port.

If there is no parallel global transfer, the local transfer will be performed over the global port without causing a pipeline stall. If a parallel global transfer is specified, the local transfer is attempted over the local port. If the address is not in the PP's local RAM, the access cannot complete over the local port and the pipeline is stalled. As a result, after the global transfer completes, the access is retried over the global port.

Name Global address registers: a8–a12

Register Type ☑ non-D register ☑ lower register ☐ upper register

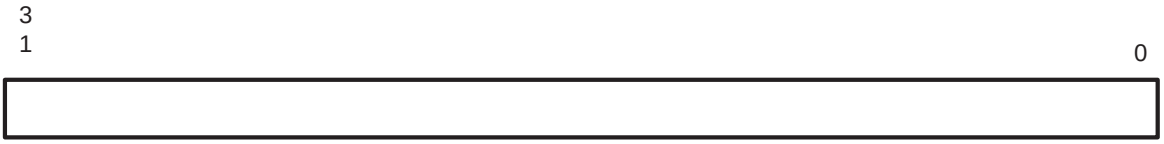
Register Code

a8: 0001 000
a9: 0001 001
a10: 0001 010
a11: 0001 011
a12: 0001 100

Companion Regs

a8: d0
a9: d1
a10: d2
a11: d3
a12: d4

Format



Description The a8, a9, a10, a11, and a12 registers are general-purpose 32-bit read/write address registers in the global address unit. Although typically used for address generation by the global address unit, these registers can also be used as non-D register operands for ALU operations.

Name Communications register: comm

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **comm:** 0111 110

Companion Regs **comm:** d6

Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
H	S	Q	P	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	r	PP#	

Packet Transfer Request
PTR Queued
Suspend
High Priority

PP Number

Description The communications register is a special-purpose 32-bit register that contains packet transfer request handshake bits (H, S, Q, and P), and bits indicating the PP number. The H, S, Q and P bits are all cleared to 0 by reset.

❑ **H: High Priority**

Read/write bit that sets the priority of a packet transfer at the time the packet transfer request is submitted.

- 1 = High Priority
- 0 = Low Priority

❑ **S: Suspend**

Read/write bit used to suspend the queued packet transfer request.

❑ **Q: PTR Queued**

Read-only bit indicating that a packet transfer request is queued. Note that there is one delay slot instruction between the submission of a packet transfer request (by setting the P bit) and the setting of the Q bit.

❑ **P: Packet Transfer Request**

Bit used to submit a packet transfer request. Can only be written to by software if S is 0 or being set to 0.

❑ PP#: PP Number

The PP number represents a read-only two-bit encoding of the PP number (00–PP0, 01–PP1, 10–PP2, 11–PP3). You can use this number to set up command words in a PP-relative fashion. The PP number bits are one of two parts of the PPs that are unique to each PP device. Note that bit 2, currently implemented as 0, is reserved for possible future MVP devices with up to eight PPs. For future code compatibility use all three LSBs when performing operations based on the PP number.

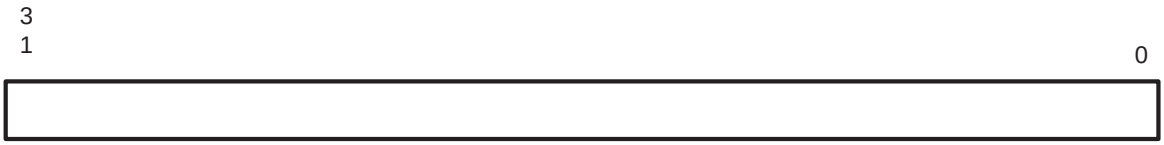
Name D registers: d0–d7

Register Type ☐ non-D register ☑ lower register ☐ upper register

Register Code
d0: 0100 000
d1: 0100 001
d2: 0100 010
d3: 0100 011
d4: 0100 100
d5: 0100 101
d6: 0100 110
d7: 0100 111

Companion Regs
d0: d0
d1: d1
d2: d2
d3: d3
d4: d4
d5: d5
d6: d6
d7: d7

Format



Description D registers are the primary data registers used by the ALU. D registers are the only operands allowed for the multiplier and ALU operations involving the bit-detection operations. D registers are located directly in the data unit. Up to seven reads and four writes can be made to the D registers in a given cycle:

Reads	Writes
Gsrc	Gdst
Lds	Lds
ALU source 1	ALU result
ALU source 2	Multiplier (or barrel rotator) result
Multiplier source 1	
Multiplier source 2	
d0 special functions	

As discussed in the description of the d0 register on the following page, register d0 is also used to configure extended ALU (EALU) operations and the default rotate amount for the barrel rotator.

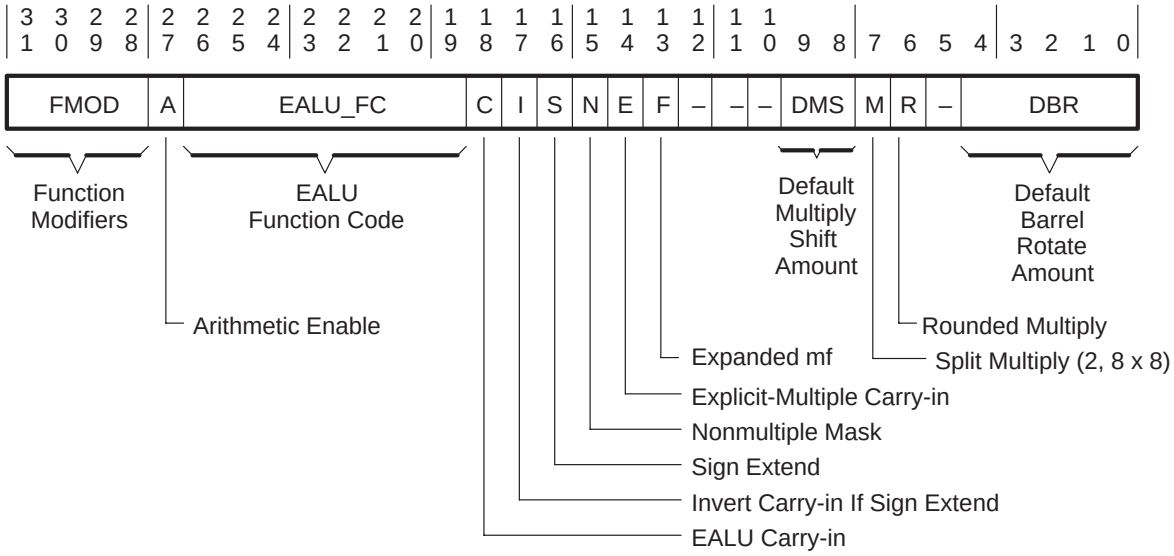
Name Special-function D register: d0

Register Type ☐ non-D register ☑ lower register ☐ upper register

Register Code **d0:** 0100 000

Companion Regs **d0:** d0

Format



Description The d0 register can be used as a normal data register in the same manner as d1–d7. It can also be used to configure the extended ALU (EALU) operation (bits 31–13), various multiply options (bits 9–6) for MPY||EALU, and the default rotate amount (bits 4–0). The d0 bit fields are described briefly below. For a more detailed explanation of how each field is used, see Section 8.4*, The Extended Set of ALU Operations (EALU)*.

❑ FMOD: Function Modifier

The four FMOD bits allow the ALU function specified by bits 27–19 of d0 to be modified. Modifiers include multiple arithmetic, carry-in from previous carry-out (sr⟦C⟧ bit), and bit-detection operations (lmo, rmo, lmbc, or rmbc). These modifier options are exactly the same as those supported by the base set arithmetics (described in subsection 8.3.4, *ALU Function Modifiers*). For EALUs, the function modifiers apply to both arithmetic and Boolean functions, while for base set ALU operations, the function modifiers are available only for arithmetics.

❑ A: Arithmetic Enable

This bit indicates whether the function performed by an extended ALU is arithmetic (A = 1) or bitwise Boolean (A = 0). This bit controls the enabling of the carry-generates in the ALU. Unlike the base set of ALU operations, the carry-in behavior is unaffected by this bit (see C bit).

❑ EALU_FC: Extended ALU Function Code

These eight bits determine the function code for the extended ALU operation. The extended ALU function code may be modified by the sign-extend selection (S) bit. Additionally, ealuf operations cause the EALU function code to be inverted before proceeding to the sign-extend logic that may further alter its value.

❑ C: EALU Carry-in Starting Value

Extended ALUs specify the carry-in directly in d0. The C bit contains the starting value for the ALU's bit 0 carry-in and is equally applicable to Boolean and arithmetic operations. The actual carry-in to the ALU is determined not only by the C bit but also the I,S, and E bits and the function modifier code.

Refer to Table 13–2*, EALU Carry-In and Sign-Extension Specification* for more information on carry-in and sign-extension specifications.

❑ I: Invert Carry-in

If the S bit, the I bit, and the sign of the data input to the barrel rotator all have a value of 1, then the carry-in to the ALU will be inverted. This is useful with certain EALU function codes to perform absolute value operations such as:

A – |B| and A + |B|.

❑ S: Sign-Extend Selection

This bit selects the sign extension required for sign-extended right shift, >>s. Sign extension is implemented by inverting the four LSBs of the EALU function code if the input to the barrel rotator is negative.

❑ N: Nonmultiple Mask

This bit determines whether the mask produced by the mask generator for a multiple arithmetic EALU operation is nonmultiple (a single 32-bit mask) or multiple (split into the same number of pieces as the ALU).

- N = 0 (default).

  The mask generator produces a multiple mask whenever multiple arithmetic is specified.

- N = 1.

  The mask generator produces a single (nonmultiple) 32-bit mask for EALU operations, regardless of whether multiple arithmetic is specified.

❑ E: Explicit-Multiple-Carry-in

This bit allows the carry-in value to be determined at runtime based on the input to the C port. This supports multiple arithmetic functions such as the following, which performs either an add or subtract in each individual portion of the split ALU as determined by the value you input to the C port:

A + (B & C) | (–B & ~C)

The E bit takes effect only if the A bit is also 1 and the function modifier code is not cin. When you use a function modifier that specifies a bit-detection operation, do not set the E bit. When the E bit is set, the effects of the S, I, and C bits are annulled.

❑ F: Expanded mf

This bit causes the expanded mf (@mf) to be sent to the C port of the ALU. This bit can be used in combination with the explicit-multiple-carry-in to have the expanded mf value control the carry-ins for each split of the ALU.

The F bit is ignored when there is an FMOD function (bits 31–28 of d0) that causes the bit-detection logic (lmo, rmo, lmbc, or rmbc) or the %! option of the mask generator to be employed.

❑ DMS: Default Multiply Shift

When a multiply is performed in parallel with an EALU, the default multiply shift indicates the number of bits (0, 1, 2, or 3) by which the multiplier result should be left-shifted before being written (via the rounding logic, if specified) to the multiply destination register. The MSBs shifted out are discarded, and 0s are shifted into the LSBs.

You can use the default multiply shift to automatically align the desired portion of the result to the upper halfword of the destination register.

❑ M: Split Multiply

- M = 0.

  Multiplier performs one 16 x 16 = 32 multiply.

- M = 1.

  Multiplier performs two 8 x 8 = 16 multiplies when a multiply is performed in parallel with an EALU operation.

❑ R: Rounded Multiply

When a multiply is specified in parallel with an EALU operation, the R bit determines whether rounding and some associated functions are performed.

- R = 0.

  Rounding is not performed. The full 32 bits of the multiplier output are written to the destination register.

- R = 1.

  Rounding is performed. Bit 15 of the multiplier output is added to multiplier output bits 31–16. This results in half-bit rounding of the MS halfword. Then, bits 15–0 of the multiplier output are discarded.

  Instead, bits 31–16 of multiply source 1 are written to bits 15–0 of the multiply destination register. This allows back-to-back rounded multiplies to be performed on packed halfwords, automatically merging the results (see subsection 4.7.2, *Rounding and Scaling 16x16 Multiply Operations*).

❑ DBR: Default Barrel Rotate

The five LSBs of d0 specify the default rotate amount used by both extended ALU operations and various base set ALU operation classes.

Name Interrupt enable register: inten

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **inten:** 0111 100

Companion Regs **inten:** d4

Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r	r	r	r	E	E	E	E	–	–	–	E	E	E	E	–	–	E	–	–	–	–	–	–	–	–	–	–	–	–	–	W
Interrupts				PP3MSG	PP2MSG	PP1MSG	PP0MSG				MPMSG	PTENDR	PTERR	PTQ			TASK														

Description The interrupt enable register contains bits for enabling individual interrupt sources and also a bit that configures the behavior for software writes to the intflg register.

❑ **E: Enable**

Each interrupt source can be individually enabled by setting a one in the corresponding enable (E) bit of the inten register.

❑ **W: Write Mode**

Bit 0 of the inten register is used to modify the behavior of writes to intflg. The W bit is normally left as zero, except when restoring state information upon returning to a task.

- W = 0.

  The normal state of the W bit. Each bit of intflg can be cleared to 0 by writing a 1 to it. Writing a 0 has no effect.

- W = 1.

  Each bit of intflg can be set to 1 by writing a 1 to it. Writing a 0 has no effect.

Initialization At reset, inten is in a random state. The inten register should be initialized before interrupts are globally enabled. Because of pipeline considerations discussed in Section 9.2, *Enabling and Disabling Interrupts*, there should be at least one delay-slot instruction between an instruction that initializes or modifies inten and the enabling of interrupts with eint.

Name Interrupt flag register: intflg

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **intflg:** 0111 101

Companion Regs **intflg:** d5

Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r	r	r	r	I	I	I	I	–	–	–	I	I	I	I	–	–	I	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Interrupts				PPM3MSG	PPM2MSG	PPM1MSG	PPM0MSG					MPMSG	PTEND	PTER	PTQ		TASK														

Description The interrupt flag register contains bits representing interrupt flags from the nine interrupt sources. These interrupt sources are described in more detail in subsection 9.1.3, *The intflg Register*.

The location of an interrupt flag in the inten register indicates its priority relative to other interrupts. Interrupt priority increases as you move from left to right in the intflg register. Thus, the task interrupt has the highest priority.

I: Interrupt Flags.

Bit 0 is not affected by software writes to the intflg register. All other intflg bits can be modified by a software write. The behavior for software writes to the intflg register is configured by the W bit of the inten register, as follows:

When W = 0 (the normal state for the W bit), writing a 1 to an intflg bit will cause it to be cleared to zero. Writing a 0 has no effect. This allows interrupt flags to be cleared without disturbing the state of others. If a particular interrupt source is trying to set a flag at the same time as the software is trying to clear it, the flag will remain a 1.

When W = 1, writing a 1 to an intflg bit will set the bit to 1. Writing a 0 has no effect. This mode is used primarily for restoring state information upon returning to a pre-empted task.

Initialization

At reset, intflg is in an unknown state. The intflg register should be initialized to 0 before interrupts are enabled. Because of pipeline considerations discussed in Section 9.2, *Enabling and Disabling Interrupts*, there should be at least one delay-slot instruction between an instruction that initializes or modifies intflg and the enabling of interrupts with eint.

Note:

Reserved bits in the intflg register cause an interrupt to be taken if they are set to 1, interrupts are globally enabled, and the corresponding bit in the interrupt enable register is set to 1. Thus, on current generations of the MVP, these reserved bits should be initialized to 0 and never set to 1.

Name Instruction pointer, address stage register (read only): ipa

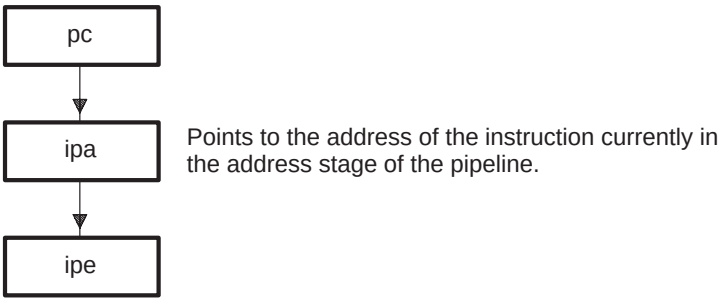
Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **ipa:** 0111 001

Format

31 … 0
32-Bit Copy of the Previous pc Register Value

Description The instruction pointer address stage register is a read-only history register for the program counter; ipa indicates the address from which the instruction currently executing the address stage of the pipeline was fetched. The ipa register is loaded with a copy of the previous value of the entire 32 bits of the pc register each time that the pipeline advances. The only exception to this behavior is during the first two pseudoinstructions of an interrupt when bit 0 is recorded as a 1, regardless of the L bit value of the pc register, and bit 1 is loaded with the effective value of the corresponding bit in the pc register. Refer to subsection 9.1.1, *The pc Register,* for more information.



Note:

The ipa register is read-only. It cannot be used as the destination for a PP operation.

Name Instruction pointer, execute stage register (read only): ipe

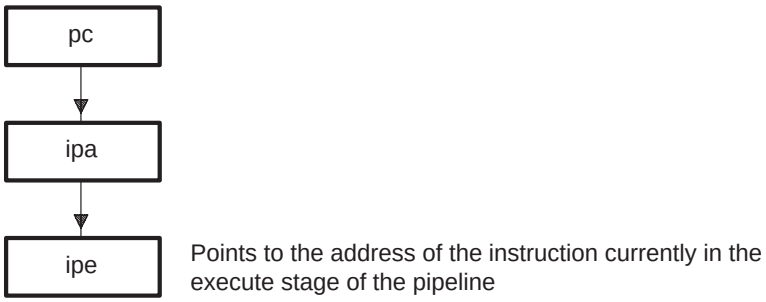
Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **ipe:** 0111 010

Format

31	0
32–Bit Copy of the Previous ipa Register	

Description The instruction pointer execute stage register is a read-only history register for the program counter; ipe indicates the address from which the instruction currently in the execute stage of the pipeline was fetched. ipe is loaded with a copy of the previous value of the 32-bit ipa register each time that the pipeline advances.



Notes:

- Because the ipe register contains the address of the instruction being executed, it can be used to perform PC-relative computations. For example, the ipe register is used to perform relative branches (see Section 10.1, *Software Branching*).

  Note that the value of the pc register cannot be reliably used for relative branches, because a delay slot branch, loop end, or interrupt may cause a PC discontinuity.

- The three LSBs of any immediate or register value added to the ipe register should be 0 to prevent undesirable carries from the G **and** L control bits into the PC.

Name Instruction pointer, return from subroutine register: iprs

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **iprs:** 0111 011

Companion Regs **iprs:** d3

Format

31	3	2	1	0
29-Bit Doubleword Address		–	–	–

Description The instruction pointer, return-from-subroutine, register gets updated with a return address each time a subroutine call is performed. When a write is performed to the call register code (refer to pc register description for details), the iprs register is loaded with the call return address. This address is the PC value plus 1 at the time the call is executed (the instruction immediately following the second branch delay slot instruction).

When the iprs register is written to as the result of a call, only the 29 MSBs of the pc register are copied to the iprs register. The 3 LSBs of iprs are loaded with 0s. Bits 2–0 of the pc register are not copied into the iprs register, because they contain a reserved bit and the G and L control bits, which should not be affected by a return from call.

Note:

If a call within a call is to be performed, you must first explicitly save the iprs contents (onto the stack). Otherwise, the first call return address is overwritten and lost.

Name Loop counter registers: lc0–lc2

Register Type ☑ non-D register ☐ lower register ☑ upper register

Register Code
lc0: 1100 000
lc1: 1100 001
lc2: 1100 010

Companion Regs
lc0: d0
lc1: d1
lc2: d2

Format

31 … 0
32-Bit Current Loop Count

Description The three 32-bit loop counters indicate the number of repetitions remaining in their associated loops. Each time that an enabled loop end is encountered, the loop counter designated in the lctl register is decremented. When two or three loop ends coincide, the prioritization order from highest to lowest is lc2, lc1, lc0.

Software writes to a loop counter dominate over the count decrement associated with a loop end.

The end condition for a loop is loop count = 0. When an enabled loop end is encountered and the designated loop counter is 0, the loop counter is reloaded from its associated loop reload register and the PC increments by one instruction instead of going to the associated loop start address.

Name Loop control register: lctl

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **lctl:** 0111 111

Companion Regs **lctl:** d7

Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	E	LCD2			E	LCD1			E	LCD0		
																				le2				le1				le0			

Description The loop control register contains an individual enable bit for each loop end register (le2, le1, and le0). The lctl register also designates the loop counter associated with each loop end.

❑ **E: Loop End Enable**

Three loop end enable bits correspond to le2, le1, and le0. When E=0, the corresponding loop end is disabled and no comparison is done with the program counter to check for the loop end.

❑ **LCD*n*: Loop Counter Designator**

The three LCD*n* fields, each consisting of three bits, indicate which loop counter (if any) should be decremented when the loop end register is successfully compared against the program counter.

- 000—None.

  When the corresponding loop end is encountered, no loop counter is checked; the loop/branch to the loop start address is always taken. This allows you to perform zero-delay branches without having to maintain a non-zero loop count.

- 001—lc0.
- 010—lc1.
- 011—lc2.
- 1xx—Reserved.

Currently, the only permitted loop counter designations are:

- LCD0: lc0 or none.
- LCD1: lc1, lc0 or none.
- LCD2: lc2, lc0 or none.

All other combinations are reserved. If the lc0 is assigned to two or three loop end registers, multiple ends to the same loop can be supported.

Note:

The lrs*n* and lrse*n* registers' fast looping initializations assign bits lc2 to le2, bits lc1 to le1, or bits lc0 to le0 for *n* = 2,1,or 0, respectively.

Name Loop end registers: le0–le2

Register Type ☑ non-D register ☐ lower register ☑ upper register

Register Code
le0: 1110 100
le1: 1110 101
le2: 1110 110

Companion Regs
le0: d4
le1: d5
le2: d6

Format

31	3 2 1 0
29-Bit Loop End Address (MSBs)	LSBs

Description

❑ **29 MSBs**

The loop end registers (le0–le2) each contain a 29-bit value indicating a loop end address. The comparison to the address in each of the loop end registers is individually enabled by the lctl register. All enabled loop end registers are compared against the 29 MSBs of the pc register during the fetch stage of the pipeline. If any matches are found, the associated loop start address of the highest priority matched loop end (le2, le1, or le0) with a nonzero loop count is loaded to the pc register before the following instruction is fetched (no delay slots). Also, the loop count register associated with the loop end register (as specified in the lctl register) is decremented.

❑ **3 LSBs**

These bits are not used by the loop logic but are implemented as read/write bits. Therefore, you can use a loop end register as a 32-bit data register if the corresponding loop end comparator is not enabled in the lctl register.

Name Loop reload registers: lr0–lr2

Register Type ☑ non-D register ☐ lower register ☑ upper register

Register Code
lr0: 1100 100
lr1: 1100 101
lr2: 1100 110

Companion Regs
lr0: d4
lr1: d5
lr2: d6

Format

31 … 0
32-Bit Loop Reload Count

Description The three 32-bit loop reload registers are used to support nested loops. Each time an enabled loop is exited because the loop counter value has reached zero, the loop counter register is reloaded with the value in the corresponding loop reload register (lr0 for lc0, lr1 for lc1, and lr2 for lc2). This prepares the loop counter for the next entry to the loop.

Note:

In order to reduce loop set-up instructions, whenever a write is performed to a loop reload register, the same value is also written to the associated loop counter register. The converse is not true (for example, writing to the lc0 register will not change the lr0 register). Therefore, when loop values are restored after task switches, loop reload registers should be restored first.

Name Multi-instruction loop fast initialization registers (write only): lrs0–lrs2

Register Type ☑ non-D register ☐ lower register ☑ upper register

Register Code
lrs0: 1101 100
lrs1: 1101 101
lrs2: 1101 110

Companion Regs
lrs0: d4
lrs1: d5
lrs2: d6

Format

31	0
32-Bit Initial Loop Count	

Description The lrs*n* register code is a write-only mapping of the loop reload registers that is used to perform a fast initialization of the ls*n*, lr*n*, lc*n*, and lctl registers for multi-instruction loops. The le*n* register must be separately loaded. A write to an lrs*n* register performs the following (*n* = the loop register number):

- Loads the lr*n* register (and thus the lc*n* register) with the specified immediate-, register- or memory-sourced count.
- Loads the ls*n* register with PC (at time instruction is executed) plus 1 instruction.
- Sets the E bit in the lctl register, thus enabling the le*n* register. Sets the loop counter designator to the lc*n* register.

Note:

The above operations all occur in a single cycle, during the execute pipeline stage. Thus, there are two delay slots between the instruction and the start of the loop.

Name Single-instruction loop fast initialization registers (write only): lrse0–lrse2

Register Type ☑ non-D register ☐ lower register ☑ upper register

Register Code
lrse0: 1101 000
lrse1: 1101 001
lrse2: 1101 010

Companion Regs
lrse0: d0
lrse1: d1
lrse2: d2

Format

31 … 0
32-Bit Initial Loop Count

Description The lrse*n* register code is a write-only mapping of the loop reload registers that is used to perform a fast initialization of single-instruction loops. A write to an lrse*n* register address performs the following (*n* = the loop register number):

- ❑ Loads the lr*n* register (and thus the lc*n* register) with the specified immediate-, register- or memory-sourced count.
- ❑ Loads the ls*n* register with the PC value (at the time the instruction is executed) plus 1.
- ❑ Loads the le*n* register with the PC value (at the time the instruction is executed) plus 1.
- ❑ Sets the E bit in the lctl register, thus enabling the le*n* register. Sets the loop counter designator to the lc*n* register.

Note:

The above operations all occur in a single cycle, during the execute pipeline stage. Thus, there are two delay slots between the instruction and the one instruction loop.

Name Loop start registers: ls0–ls2

Register Type ☑ non-D register ☐ lower register ☑ upper register

Register Code
ls0: 1110 000
ls1: 1110 001
ls2: 1110 010

Companion Regs
ls0: d0
ls1: d1
ls2: d2

Format

31	3 2 1 0
29-Bit Loop Start Address (MSBs)	LSBs

Description

- **29 MSBs**

  Each loop start register contains a 29 bit address to be loaded into the PC when the associated enabled loop end register value equals the PC, the associated loop counter is not zero, and no higher priority loop ends are matched. The PC load occurs before the fetch of the next instruction, so there are no delay slots.

- **3 LSBs**

  These bits are not used by the loop logic but are implemented as read/write bits. Therefore, you can use a loop start register as a 32-bit data register if the associated loop end is not enabled in the lctl register.

Name Multiple flags register: mf

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **mf:** 0101 010

Companion Regs **mf:** d2

Format

31	0

Description The multiple flags register records either carry-out, zero-compare, sign bit, or extended sign bit status information from each split ALU segment for multiple arithmetic operations. Additionally, bits in the mf register can be expanded to create a mask that is input to the ALU C port. This mask can be used to control the operation that is performed by each split ALU segment. Multiple arithmetic status saving to the mf register along with mf expansion allows many powerful merging and data-size translation operations.

The mf register has 32 read/write bits that can be written to directly by software. Software writes to the mf register dominate over multiple arithmetic status setting to the mf register. Status bits are written to the mf register only during multiple arithmetic operations. The mf register is not affected by status setting for nonmultiple ALU operations.

mf Bit Setting

For multiple arithmetic operations, a status flag can be set in the mf register for each segment of the split ALU. The type of status used to set mf bits is specified by functions modifiers and the MSS field in the status register. One of the ALU function modifiers allows for split ALU operations with no status bit setting in mf. There are two other types of ALU function modifiers that specify multiple arithmetic:

- The =mc type sets mf bits on the basis of the carry-out of each split ALU segment.
- The =me type uses the MSS field in the status register to choose between the zero-comparison, the sign bit, or the extended sign of each split ALU segment for mf bit setting.

The Asize field of the status register controls the number of segments that the ALU is partitioned into and thus also determines the number of mf bits affected by multiple arithmetic status setting.

The 32/(Asize) LSBs of the mf register are set by a multiple arithmetic operation. Table 7–2 summarizes mf bit settings for multiple arithmetic. As shown in Table 7–2, if Asize is 16 (halfword multiple arithmetic), bits 3 and 2 of mf are not affected by multiple arithmetic status setting.

- If the mf bit setting is on the basis of zero, then mf bit 1 is set by the zero-comparison of ALU result bits 31–16; mf bit 0 is set by the zero-comparison of ALU result bits 15–0.
- If the mf bit setting is on the basis of carry, then mf bit 1 is set by the carry-out from bit 31 of the ALU result; mf bit 0 is set by the carry-out from bit 15 of the ALU result.
- If the mf bit setting is on the basis of the sign bit, then mf bit 1 is set to bit 31 of the result (the sign bit), and mf bit 0 is set to bit 15 of the result.
- If the mf bit setting is on the basis of the extended sign, then:
  - mf bit 1 is set to 1 if the result of a sign-extended add or subtract result of bits 31–16 is negative.
  - mf bit 0 is set to 1 if the sign-extended result of bits 15–0 is negative.

The extended sign case effectively adds one bit of precision, saved in the mf bits, to each signed result of a split ALU operation. The extended sign bit differs from the sign bit of each split in the cases where the result would have overflowed (see Section C.5, *Extended-Precision Support for Signed Operations With Multiple Arithmetic*, for a more detailed explanation of how the extended sign bit works).

Table 7–2. ALU Result Bits Used to Set mf Bits

	ALU Result Bits Compared to Zero (mf Status Setting Based on Zero)			ALU Status-Setting Bit (mf Status Setting Based on Carry, Sign, or Extended Sign)		
mf Bit	**Asize = 8**	**Asize = 16**	**Asize =32**	**Asize = 8**	**Asize = 16**	**Asize = 32**
3	31—24	—	—	31	—	—
2	23—16	—	—	23	—	—
1	15—8	31—16	—	15	31	—
0	7—0	15—0	31—0	7	15	31

You can prevent writes of the split ALU status bits to the mf register by choosing the function modifier option that does not modify the mf bits.

mf Bit Expansion

Multiple arithmetic status flags saved in the mf register can be used to supply mask data to the ALU C port. A bit expander along the data path into the C port of the ALU takes a number, n, of mf bits and replicates each bit 32/n times in order to create a 32-bit mask. The @ symbol represents the function performed by the expander. For multiple-byte arithmetic, four mf bits are used; two are used for multiple-halfword arithmetic; and one is used for multiple-word arithmetic.

When the expander output (@mf) is the selected input to the ALU C port, each byte of C port data is derived from an mf bit, and that mf bit is replicated to all bits of the C port byte. The bit used for each byte is determined by the Msize field (bits 5–3) and the R bit (bit 6) in the status register as shown in Table 7–3. The rotation option selects whether the input to the expander comes from the MSBs of mf (R=1) or the LSBs of mf (R=0).

Table 7–3. mf Bits Supplied to ALU C Port by mf Expander

	mf Bit Supplied to the C Port by the expander					
	R = 0			R = 1		
ALU C Port Bits	Msize = 8	Msize = 16	Msize =32	Msize = 8	Msize = 16	Msize = 32
31—24	3	1	0	31	31	31
23—16	2	1	0	30	31	31
15—8	1	0	0	29	30	31
7—0	0	0	0	28	30	31

mf Rotation

The mf register can be automatically rotated left by one, two or four bit positions. Rotation can be specified two different ways: an ALU function modifier that specifies rotation according to Asize, or the R bit set to 1 that, when @mf is specified, causes rotation according to Msize. If both the ALU function modifier and the R bit specify rotation, the rotation amount specified by Asize dominates. If the ALU function modifier does not specify rotation (including all multiple arithmetic operations that do not explicitly specify rotation by Asize), then rotation is controlled by the R bit value, and the rotate amount is determined by Msize.

The following summarizes the mf register rotation:

- If the ALU function modifier specifies multiple arithmetic with mf rotation (= mr, = mrc, or = mre), then the rotation amount is specified by Asize.

  This allows new status flags to be written to the mf register without overwriting previous flags. Therefore, you can accumulate a full 32-bit word of multiple status flags.

Note:

If specified by the ALU function modifier, rotation by Asize occurs, regardless of the R bit setting. This overrides the rotation amount specified by the R bit and Msize. However, Msize and the R bit always determine the number and location of the bits used by the expander.

- If the ALU function modifier does not specify mf rotation and the operation uses the bit expander (@mf) and R = 1, then mf is rotated by the amount specified by Msize.

  The purpose of this is to rotate the mf register bits as they are used by the mf register expander.

For multiple arithmetic, multiple status setting to the mf register dominates over the rotation for those bits that are being set. Direct software writes to the mf register dominate over both mf rotation and multiple arithmetic status setting to mf.

mf Clearing

An ALU function modifier makes it possible to clear all of the mf bits that are not being set by split ALU status flags. Software writes to the mf register dominate over mf clearing.

mf and Divides

The mf register is used by the divi (divide iteration) operation. For each divi operation, the contents of mf are left-shifted by one, and the LSB is filled with the OR of the bit being shifted out from the MSB and the sign of the input to the barrel rotator. After a sequence of divi operations, the mf register contains the final quotient of the desired divide. For details, see Section 8.6, *The Divide Operation*.

Name Program counter register: pc

Register Type ☑ non-D register ☑ lower register ☐ upper register

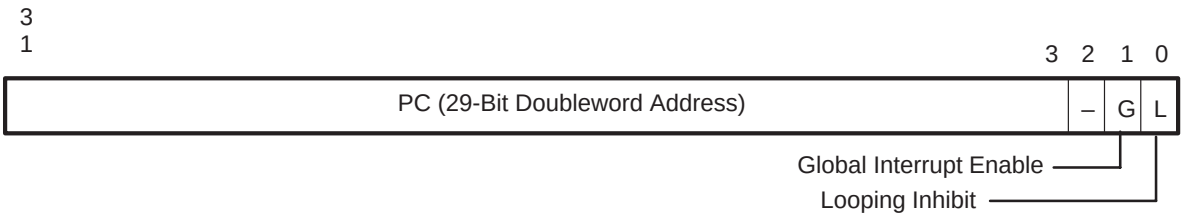
Register Code

pc:	read	0111 000
call:	call write	0111 000
br:	branch write	0111 001

Companion Regs

call: d0
br: d1

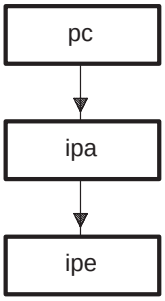
Format



Description The program counter register is a special-purpose 32-bit register that contains a 29-bit pointer to the doubleword address of the 64-bit instruction being fetched. Instructions are always aligned on 64-bit boundaries; thus, the three LSBs of the instruction address are always 0s. The two LSBs of the pc register are used for control bits: global interrupt enable (G), and loop inhibit (L). Bit 2 of the pc register is currently reserved with no assigned function.

Note:

The PC refers to the 29-bit counter visible in the pc register. The pc register refers to the entire 32-bit register that contains both the PC and the G and L control bits.



Points to the address of instruction currently in the fetch stage of the pipeline

The pc register has two different register codes for writes (br and call) but only one read register code (pc). **call** is used for subroutine calls. When a write is performed to call, the iprs register is modified to contain the subroutine return address. **br** is used for branches with no modification of the iprs register. Branches and calls are described in more detail in Section 10.1, *Software Branching*. You can specify a branch or a call by simply writing an address to br or call, respectively.

The behavior of the pc register when interrupts and returns occur is described in subsection 9.1.3, *The intflg Register*.

- **PC: Program Counter**

  A 29-bit pointer to the doubleword address of the 64-bit instruction being fetched. The 29-bit counter is aligned to the 29 MSBs of the register. The counter increments by one, each time that an instruction fetch is successful, unless the PC is modified by a software write to the pc register (branch or call), by looping, or by an interrupt.

- **G: Global Interrupt Enable**

  - G = 0 disables the recognition of all interrupt sources.
  - G = 1 enables the interrupt sources individually enabled in the inten register.

  G is set to 1 before the address stage of an eint operation and cleared to 0 before the address stage of a dint operation.

  When a PP is unhalted after reset, the G bit is set to the value of bit 1 of the task interrupt vector. You should ensure that bit 1 of the task interrupt vector is 0 to initially disable interrupts after reset.

- **L: Loop Inhibit**

  L = 1 disables the loop logic. This bit is normally set only during the return sequence from an interrupt. When a PP is unhalted after reset, the L bit is set to the value of bit 0 of the task interrupt vector. You should ensure that bit 0 of the task interrupt vector is 0 so that the loop control logic will be enabled after reset.

Note:

Usually, writes or loads to the pc register affect only the program counter and not the G and L bits. However, when the global address unit performs a load to the pc register using an unscaled immediate offset, you can modify the G and L bits.

- If bit 1 of the offset is equal to 1, then the G bit is set to the value of bit 1 of the 32-bit word being read from memory.
- If bit 0 of the offset is equal to 1, then the L bit is set to the value of bit 0 of the 32-bit word being read from memory.

This enables efficient return-from-interrupt (reti) operations. It is not intended to be used in any other situation. In all other instances, the G bit should be managed by using the eint and dint operations or by the PP hardware itself. The L bit should not be explicitly manipulated.

Name Stack pointer: sp (a6 or a14)

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **sp:** The assembler selects between codes for a14 for global (default) and a6 for local.
a6: 0000 110
a14: 0001 110

Companion Regs **a6:** d6
a14: d6

Format

31 … 2	1	0
Word-Aligned Address	0	0

Description The stack pointer is a special-purpose 32-bit address register that contains the address of the bottom of the system stack. Each PP contains only one stack pointer register; however, this stack pointer register can be addressed by either the global address unit (using register code a14) or the local address unit (using register code a6).

The assembler accepts the **sp** keyword to indicate the stack pointer register. When the sp register is specified, the assembler selects either register a14 or a6, depending on which address unit is available to perform the specified operation. The a14 register is used by default.

The stack pointer is initialized to 0x0100 #7F0 (# = PP number) during the reset sequence (see Section 9.6, *Reset*, for more details). The PP stack grows towards smaller addresses starting from 0x0100#7F0. Pushes are predecrement stores, and pops are postincrement loads.

The two LSBs of the stack pointer are hard-wired to 0 (writing to them has no effect). Thus, only aligned 32-bit stack operations are allowed.

During interrupts, the PC, IPA, and IPE values, the G and L bits, and the sr register are pushed onto the stack. This is discussed in detail in Section 9.5, *Interrupt Processing*.

Note:

It is possible to use the stack pointer for both the global address unit and the local address unit operations specified in the same instruction, as long as no more than one of the address units specifies modification of the stack pointer.

Parallel use of the stack pointer results in code compaction. However, it does not result directly in faster execution, because crossbar contention prevents both accesses from occurring in a single cycle. If both accesses are to the same address and at least one of the accesses is a store, the resulting behavior is unpredictable because of the crossbar contention round-robin.

Restrictions

The PP assembler detects some incorrect stack operations. An error condition is flagged by the assembler if both address units specify modification of the stack pointer in the same instruction.

If an instruction specifies parallel stack accesses (with at least one access being a store) to the same address, the resulting behavior is unpredictable.

Name Status register: sr

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **sr:** 0101 001

Companion Regs **sr:** d1

Format

31	30	29	28	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	8	7	6	5	3	2	0
N	C	V	Z	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	MSS	R	Msize		Asize		

Status Bits

Description The status register is a special-purpose, 32-bit register that contains four status bits that are set according to the result of PP ALU operations. Additionally, the sr register contains:

- The **MSS field** (mf status selection) selects which type of ALU result information is used to set the mf register for the me, mre, and mze function modifiers.
- The **R bit** which specifies rotation of the mf register for ALU operations using the expander
- The **Msize field** which determines the data size for the expander
- The **Asize field** which determines the split ALU data size for multiple arithmetic

Status-Setting

The four status bits are set based on the result of ALU operations. The following is a summary of status-setting behavior:

- The status bits are set based on the result of every ALU operation unless:
  - Status protection is specified in the instruction (individual status bits can be protected from modification by both status setting and software writes to the sr as described in subsection 8.9.4, *Status Bit Protection*),
  - The ALU write to destination is conditional, and the condition is false, or
  - The destination of the ALU operation is the pc register (br or call). This allows status information to be preserved for use within a subroutine after a call. It also allows branches in delay slots of a conditional branch based on mutually exclusive conditions.
  - The status register is written to explicitly.

- Status-setting behavior for Boolean and arithmetic operations is identical. The carry and overflow bits are both cleared by normal Boolean operations.
- Special status-setting behavior exists for multiple arithmetic, bit-detection operations (lmo, rmo, lmbc, rmbc), and the divi operation (see the status bit discussion below).
- Multiplies do not affect the status bits.
- Address unit operations do not affect the status bits unless the status register is the destination for the operation. In that case, the software write to the status register overrides status setting.

Status Bits

The individual status bits are set as follows:

- **N: Negative (sr bit 31)**
  - Normal

    Set to the sign bit (bit 31) of the result of the ALU operation. N = 1 indicates the result was negative; N = 0 indicates the results was positive.
  - Multiple Arithmetic

    Set to the AND of the zero compares for the split ALU.
  - Bit Detection

    Set to the AND of the zero compares for the split ALU.
  - Divide

    If you are executing the divi operation, the N bit is set according to the following expression:

    carry-out | bit 31 of the input to the barrel rotator

    For details, see Section 8.6, *The Divide Operation*.

❑ C: Carry (sr bit 30)

- Normal

  Set to the value of the carry-out from bit 31 of the ALU. C = 1 indicates that the result of the ALU operation caused a carry-out from bit 31 of the ALU. Note that for subtract operations (performed by adding the 2s complement) the carry bit is set on borrow (this is the natural output of the ALU).

- Multiple Arithmetic

  Records the OR of the split ALU carry-outs. If C = 1, this indicates that at least 1 of the split ALUs had a carry-out.

- Bit Detection

  Records the OR of the split ALU carry-outs. If C = 1, this indicates that at least 1 of the Split ALUs had a carry out.

❑ V: Overflow (sr bit 29)

- Normal

  Set to the XOR of the carry-in and carry-out for bit 31. V = 1 indicates that the result of the ALU operation experienced a 2s complement overflow condition.

- Multiple Arithmetic.

  Set to the AND of the split ALU carry-outs.

- Bit Detection.

  For the lmo and rmo operations, V = 1 indicates that no 1s were present in the input word.

  For the lmbc and rmbc operations, V=1 indicates that all bits in the source were the same.

❑ Z: Zero (sr bit 28)

- Normal

  Set to 1 if the result of the ALU operation was equal to 0; set to 0 if the result was nonzero.

- Multiple Arithmetic

  Set to the OR of the zero compares for the split ALU.

- Bit Detection

  Set to the OR of the zero compares for the split ALU.

R (Rotation) Bit

The R bit determines which bits are used by the expander and specifies automatic rotation of the mf register when it is expanded.

- **R = 1**

  The bits used by the expander are taken from the MSBs of the mf register. For an operation involving the expander in which the ALU function modifier **does not** specify mf rotation according to Asize, the mf register is post-rotated according to Msize.

  If the ALU function modifier **does** specify mf rotation according to Asize, the rotation associated with the R bit and Msize is not performed. Instead, mf is rotated by the amount specified by Asize. However, the bits used by the expander are still taken from the MSBs of the mf register.

- **R = 0**

  The bits used by the expander are taken from the LSBs of the mf register. This setting disables mf rotation by Msize, but mf rotation by Asize still occurs if specified by the ALU function modifier.

MSS Field

The mf status selection (MSS) field in the status register selects which type of ALU result information is used to set the mf register for the me, mre, and mze function modifiers. The function modifiers enable the setting of bits in the mf register as shown in Table 7–4.

Table 7–4. MSS Field Values

MSS Field		
Bit 8	**Bit 7**	**Value/Action**
0	0	mf bit(s) set by zero result(s). This causes a bit in the mf register to be set to 1 for each split ALU result that is zero.
0	1	mf bit(s) set by sign bit(s). This causes the MSB of precision (the sign bit) of each split ALU result to be copied into the mf register.
1	0	mf bit(s) set by extended result bit(s) for signed arithmetic. This causes the mf bits to be set to the sign bit of the result, assuming the two original numbers were sign-extended by one bit.
1	1	Reserved

The extended sign mode follows the rules of signed arithmetic, assuming that the mf register bit gives one more bit of precision that is saved in the mf register. In practice, this becomes:

(propagate of MSB) ^ (carry-out of MSB)

For example, if **signed** 8-bit multiple arithmetic was being performed, then four bits in the mf register would be set to what would have been the ninth result bit of each split ALU, assuming that the two 8-bit numbers had been sign-extended to 9 bits.

Note that setting bits in the mf register on the basis of the split-ALU carry-out bits is selected explicitly in the opcode by mc, mrc, and mzc function-modifier options.

Msize Field

The 3-bit Msize code indicates the data size for the expander and, in some instances, an mf rotation amount.

- **Expander Data Size**

  Msize specifies the data size for the expander. This determines how many mf register bits are used to create the mask input to the C port. These bits come from the MSBs of mf if R = 1 and the LSBs of mf if R = 0.

- **mf Rotation Amount**

  For expander operations in which the ALU function modifier does not specify rotation by Asize and R = 1, the mf register is automatically post-rotated left by the amount specified by Msize.

Table 7–5 summarizes the effect of the R bit in combination with Msize.

Table 7–5. Msize and R-Bit Values and Effects

Msize Coding	Expander Data Size (Bits)	mf Rotate Amount	Number of mf Bits Used	mf Bit(s) Used	
				R = 1	R = 0
0xx	Reserved				
100	8	4	4	31–28	3–0
101	16	2	2	31–30	1–0
110	32	1	1	31	0
111	Reserved				

Note: The mf rotate amount applies only for operations using the expander in which the ALU function modifier does not specify mf rotation by Asize.

Asize Field

The 3-bit Asize field indicates the split ALU data size for multiple arithmetic operations. It also indicates an mf rotation amount for certain ALU function modifiers.

- **Multiple Arithmetic Data Size**

  When the ALU function modifier selects multiple arithmetic, Asize indicates into how many pieces the ALU is divided. If the function modifier specifies mf-status saving, it also indicates how many multiple-status flag bits are set in the mf register. Codings for these bits are shown in Table 7–6.

- **mf Rotation Amount**

  When the ALU function modifier selects multiple arithmetic with rotation, the Asize field determines by how many bits the mf register is post-rotated left. The rotate amount indicated by Asize dominates over the Msize/R rotate amount when the ALU function modifier indicates multiple arithmetic with rotation.

Table 7–6. Asize Values and Effects

Asize Coding	Split ALU Data Size (Bits)	mf Rotate Amount	Number of mf Bits Set	mf Bit(s) Set
0xx	Reserved			
100	8	4	4	3–0
101	16	2	2	1–0
110	32	1	1	0
111	Reserved			

Note: The mf rotate amount applies for operations in which the ALU function modifier specifies mf rotation by Asize (= mrc or = mre).

Name Cache tag and present bit registers (read only): tag0–tag3

Register Type ☑ non-D register ☐ lower register ☑ upper register

Register Code

tag0: 1111 100
tag1: 1111 101
tag2: 1111 110
tag3: 1111 111

Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
23-Bit Tag Value																							P	P	P	P	–	–	–	LRU	

Subblock Present Bits

Description The cache tag registers appear in the register map for test purposes. They should not be written to by users. Reading the cache tag registers can sometimes provide useful debug information.

❑ **Tag Value**

The 23 MSBs of the tag registers hold the address of the start of the cache block.

❑ **Present Bits**

Bits 8–5 contain the present bits associated with each cache subblock. From left to right, they represent the most significant subblock (bit 8) through the least significant subblock (bit 5).

❑ **Bits 4 to 2**

Reserved.

❑ **LRU**

Least recently used code. Bits 1–0 indicate how recently the associated cache block was used relative to the cache blocks associated with the other cache tag registers. They cannot be written to by software.

- 00—Most recently used
- 01—Second most recently used
- 10—Second least recently used
- 11—Least recently used.

Name Local index registers: x0–x2

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code
x0: 0010 000
x1: 0010 001
x2: 0010 010

Companion Regs
x0: d0
x1: d1
x2: d2

Format

31	0

Description The local address unit contains three index registers: x0, x1, and x2. The PP has addressing modes that allow the offset in an index register to be added to or subtracted from the base address in an address register. Both preindexing (with or without address register modification) and postindexing are supported.

Index registers can be used as ALU source operand 1 and/or the ALU destination. Therefore, while index registers are often used for address computations, they can also be used as general-purpose registers.

Name Global index registers: x8–x10

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code
x8: 0011 000
x9: 0011 001
x10: 0011 010

Companion Regs
x8: d0
x9: d1
x10: d2

Format

31	0

Description The global address unit contains three index registers: x8, x9, and x10. The PP has addressing modes that allow the offset in an index register to be added to or subtracted from the base address in an address register. Both preindexing (with or without address register modification) and postindexing are supported.

Index registers can be used as ALU source operand 1 and/or the ALU destination. Therefore, while index registers are often used for address computations, they can also be used as general-purpose registers.

Name Read-as-zero address registers: zero (a7 or a15)

Register Type ☑ non-D register ☑ lower register ☐ upper register

Register Code **a7:** 0000 111
a15: 0001 111

Companion Regs **a7:** d7
a15: d7

Format

31																															0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Description The a7 and a15 registers are read-as-zero address registers. Writes to a7 or a15 are ignored and can be specified when the result of an operation is to be discarded.

When an address unit operation is specified without a base address (a base address of 0), the assembler encodes a7 or a15 as the base address in the instruction.

Chapter 8

The PP Assembly Language Instruction Set

This chapter describes the PP instruction set. If you want general information describing the PP assembler, including how to invoke it, refer to Chapter NO TAG, *Assembler Description*, in the *MVP Code Generation Tools User's Guide*. For information on the assembler directives, refer to Chapter NO TAG, *Assembler Directives*, in the *MVP Code Generation Tools User's Guide*.

Topics

8.1	Overview of the Assembly Language Instruction Set	PP: 8-2
8.2	PP Data Unit Operators	PP: 8-12
8.3	The Base Set of ALU Operations	PP: 8-30
8.4	The Extended Set of ALU Operations (EALU)	PP: 8-61
8.5	Multiply Operations	PP: 8-87
8.6	The Divide Operation	PP: 8-96
8.7	Miscellaneous Operations	PP: 8-113
8.8	Parallel Transfers	PP: 8-116
8.9	Conditional Operations	PP: 8-144
8.10	Parallel-Operation Combinations	PP: 8-156
8.11	Parallel Write Priority	PP: 8-171
8.12	Programming Restrictions	PP: 8-175

8.1 Overview of the Assembly Language Instruction Set

This chapter describes the PP instruction set by presenting the PP's algebraic assembly language. Instruction word formats and individual opcode fields are described as necessary to explain restrictions. Refer to Appendix A, *The PP Opcodes*, for more detail on the PP's instruction word format and the individual operation code fields.

Unlike other assembly language instruction sets, the PP instructions are represented by algebraic expressions for the operations performed in parallel by the multiplier, ALU, global address unit, and local address unit.

Although expressed as separate parallel operations in the PP assembly language, the operations performed by the multiplier and the ALU are specified together in the instruction. Thus, these operations are referred to collectively as the **data unit operations**. Table 8–1 shows a summary of the supported data unit operations. Sections 8.2 through 8.6 describe these data unit operations in detail.

Global address unit operations (referred to as global transfers) and local address unit operations (referred to as local transfers) are represented by the same kinds of expressions; thus, they are described together and referred to collectively as **parallel transfers**. Table 8–2 shows a summary of the supported parallel transfers. Section 8.8 describes these operations in detail.

Also included in this chapter is an overview of the different functions available with the PP, see Table 8–3. Note that Table 8–3 contains only a small sample of the operations that can be performed on the PP. There are more complex operations that you can use that would typically require several instructions on other microprocessors.

While you can specify parallel operations in a single PP instruction, parallel code is not a requirement. It is a good idea to focus on learning the operations that you can perform with the data unit and parallel transfers individually. Once you understand these individual operations, proceed to Section 8.10, which describes the supported combinations of data unit operations and parallel transfers.

8.1.1 Data Unit Operations

In the PP opcode, 25 of the 64 bits specify the data unit operation. Since both a multiplier and the ALU are in the data unit, this portion of the opcode may specify more than one operation.

Alternatively, you can use the data unit portion of the opcode to specify miscellaneous operations that do not use the data unit hardware. The operations that you can specify in a single instruction by the data unit portion of the opcode are shown in Table 8–1.

8.1.2 Parallel Transfers

Up to two of the operations in Table 8–2 can be specified in parallel with a data unit operation in a single PP instruction

These operations are referred to collectively as parallel transfers.

A parallel transfer operation falls into one of the following two categories:

- **Global Transfer**

  A parallel transfer performed by the global address unit (this can be a load, store, address unit arithmetic, or register-to-register move).

- **Local Transfer**

  A parallel transfer performed by the local address unit (this can be a load, store, or address unit arithmetic).

Parallel transfers are described in more detail in Section 8.8, *Parallel Transfers*.

Table 8–1. Summary of Data Unit Operations

Operation	**Base set ALUs**
Description	Perform an ALU operation in which the opcode specifies an ALU function, one destination and two source operands, and the routing of the operands to the inputs of the ALU. The ALU function can be any one of the 256 three-input Boolean operations or, alternatively, one of 17 arithmetic ALU operations (shown in Table 8–11, *Class-Independent Base Set Arithmetic Functions*, and Table 8–12, *Class-Specific Base Set Arithmetic Functions*) combined with one of 15 ALU function modifiers.
Syntax	dst = [fmod] [⟦cond [.pro] ⟧] ALU_EXPRESSION
Examples	`d6 = (d6 ^ d4) & d2` `d3 =[nn.nv] d1 – 1`
See Page	PP:8-30
Operation	**EALU||ROTATE**
Description	Perform an extended ALU (EALU) operation (as specified in d0) with one of two routings of data to the ALU (as specified in the opcode and d0); the output from the barrel rotator can optionally be written to a second destination register. The ALU function can be any of the 256 Boolean or 256 arithmetic operations. The opcode specifies four source operands and two destination operands. You must enter this syntax on one line.
Syntax	dst1 = [⟦cond [.pro] ⟧] ealu (src2, [dst2 =] [⟦cond⟧] src1[⟦n⟧src1–1]\\src3, [%] src4) dst1 = [fmod] [⟦cond [.pro] ⟧] ealu(label: EALU_EXPRESSION [|| dst2 = ⟦cond⟧src1[⟦n⟧src1–1]\\src3])
Examples	`d7 =[nn] ealu(d2, d6 =[nn] d3\\d1, %d4)` `d3 =mzc ealu(foobar: d4 + (d5\\d6 & %d7) || d1 = d5\\d6)`
See Page	PP:8-66

Table 8–1. Summary of Data Unit Operations (Continued)

Operation	**MPY||ADD**
Description	Perform a 16-bit-by-16-bit multiply, yielding a 32-bit result with an add or subtract optionally performed in parallel. The multiply operands can be signed or unsigned. If either operation is conditional, both multiply and add must be conditional, and both must use the same condition code.
Syntax	dst2 = [sign] [⟦cond⟧] src3 * src4 [‖ dst = [⟦cond [.pro] ⟧] src2 + src1[⟦n ⟧src1−1]]
	dst2 = [sign] [⟦cond⟧] src3 * src4 [‖ dst = [⟦cond [.pro] ⟧] src2 – src1[⟦n ⟧src1−1]]
Example	`d7 =u d6 * d5` `|| d5 = d4 – d1`
See Page	PP:8-88
Operation	**MPY||SADD**
Description	Perform a 16-bit-by-16-bit multiply, yielding a 32-bit result with a right-shift and add or subtract performed in parallel. The multiply operands can be signed or unsigned. If either operation is conditional, both the multiply and the shift and add must be conditional, and both must use the same condition code.
Syntax	dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ dst = [⟦cond [.pro] ⟧] src2 + src1[⟦n ⟧src1−1]>>–d0
	dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ dst = [⟦cond [.pro] ⟧] src2 – src1[⟦n ⟧src1−1]>>–d0
Example	`d7 =u d6 * d5` `|| d5 = d4 – d1 >> –d0`
See Page	PP:8-88

Table 8–1. Summary of Data Unit Operations (Continued)

Operation	**MPY||EALU**
Description	Perform a multiply and an optional EALU operation in parallel. The multiply can use special features such as rounding, scaling, or splitting (two 8-bit-by-8-bit multiplies instead of one 16-bit-by-16-bit multiply). The EALU provides additional ALU arithmetic functions beyond an add or subtract and enables the special multiply modes. The actual ALU function performed by the EALU is specified in bits 27–19 of the d0 register (ealu) or the 1s complement of those bits (ealuf).
Syntax	**Generic form:** dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ dst1 = [⟦cond [.pro] ⟧] ealu[f] (src2, src1[⟦n⟧src1–1]\\d0, %d0) dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ ealu() **Explicit form:** dst2 = [sign] [opt] [⟦cond⟧] src3 * src4 [<< dms] ‖ dst1 = [fmod] [⟦cond [.pro] ⟧] ealu(label: EALU_EXPRESSION) dst2 = [sign] [opt] [⟦cond⟧] src3 * src4 [<< dms] ‖ ealu(label)

Example	`d7 =[p] d5 * d3 ; generic form`
	`  || d2 =[p] ealu(d1, d6\\d0, %d0)`
	`d2 =m d4 * d7 ; explicit form`
	`  ||d3 = ealu(foobar: d3 + d2 >> 9)`

See Page	PP:8-66

Operation	**divi**
Description	Perform one iteration of an unsigned divide algorithm. Generates one quotient bit per execution by using an iterative subtraction. Additional instructions can be used to implement a signed divide function.
Syntax	dst1=[⟦cond[.pro]⟧] divi (src2, dst2= ⟦cond⟧ src1 [⟦n⟧src1–1])
Examples	`d3 = divi(d1, d2 = d2)` `d3 = divi(d1, d2 = d3[n]d2)`
See Page	PP:8-98

Table 8–1. Summary of Data Unit Operations (Continued)

Miscellaneous Operations	
Operation	**dint**
Description	Globally disable interrupts.
Syntax	dint
See Page	PP:8-113
Operation	**eint**
Description	Globally enable interrupts.
Syntax	eint
See Page	PP:8-114
Operation	**nop**
Description	Do nothing in the data unit.
Syntax	nop
See Page	PP:8-115

Table 8–2. Summary of Parallel Transfers

Operation	**Load**
Description	Memory-to-register transfer
Syntax	dst = [sign] [size] [⟦ cond ⟧] ∗ addrexp
	dst = [sign] [size] [⟦ cond ⟧] ∗ an.element
Examples	`d3 =uh[n] *(a9++=[2])` `d1 = * a2.sPTC_BCount`
See Page	PP:8-126
Operation	**Store**
Description	Register-to-memory transfer
Syntax	∗ addrexp = [size] src [⟦ n ⟧ src − 1]
	∗ an.element = [size] src [⟦ n ⟧ src − 1]
Examples	`*--a2 = d3` `*a9.sPT_Next = a3`
See Page	PP:8-130
Operation	**Address Unit Arithmetic**
Description	Computed address-to-register transfer
Syntax	dst = [size] [⟦ cond ⟧] & ∗ addrexp
	dst = [size] [⟦ cond ⟧] & ∗ an.element
Examples	`d2 = &*(a3+x0)` `a1 = &*a9.sPT_DstCCount`
See Page	PP:8-139

Table 8–2. Summary of Parallel Transfers (Continued)

Moves	
Operation	**Move**
Description	Register-to-register transfer
Syntax	dst = [g] [⟦ cond ⟧] src
Examples	`x2 = mf` `d1 =g d3`
See Page	PP:8-134
Operation	**Field extract move**
Description	Register-to-register transfer extracting and right-aligning one byte or halfword
Syntax	dst = [sign] [size item] ⟦ cond ⟧ src
Example	`d3 =ub2 d1`
See Page	PP:8-136
Operation	**Field replicate move**
Description	Register-to-register transfer replicating the LSbyte or LShalfword to 32 bits
Syntax	dst = r [size] ⟦ cond ⟧ src
Example	`d7 =rh d3`
See Page	PP:8-138

Table 8–3. Syntax For Various PP Functions

Function	PP Syntax
Absolute Value	dst = |src|
Add	dst = src1 + src2
Add with Carry	dst = src1 + src2 + cin
Compare	dst = src1 – src2
Two's Complement	dst = –src
Subtract	dst = src1 – src2
Subtract with Borrow	dst = src1 – src2 –~cin
Multiply (Signed)	dst2 = src3 * src4
Multiply (Unsigned)	dst2 =u src3 * src4
Multiply (Signed)	dst2 = src3 * src4
with parallel add	||dst1 = src1 + src2
Multiply (Signed)	dst2 = src3 * src4
with parallel subtract	||dst1 = src1 – src2
Multiply (Unsigned)	dst2 =u src3 * src4
with parallel add	||dst1 = src1 + src2
Multiply (Unsigned)	dst2 =u src3 * src4
with parallel subtract	||dst1 = src1 – src2
Rotate Left	dst = src1 \\ src2
Shift Right Arithmetic & Add	dst = src2 + (src1 >>–d0)
Shift Left	dst = src1 << src2
Shift Right Arithmetic	dst = src1 >> –src2
Shift Right Arithmetic	dst = src1 >> imm
Shift Right Logical	dst = src1 >>u –src2
Shift Right Logical	dst = src1 >>u imm
Shift Right Arithmetic & Sub	dst = src2 – (src1 >>–d0)
AND	dst = src1 & src2
ANDn	dst = src1 & ~src2
NOR	dst = ~(src1 | src2)
NOT	dst = ~src
NAND	dst = ~(src1 & src2)
OR	dst = src1 | src2
ORn	dst = src1 | ~src2
XNOR	dst = src1 ^ ~src2
XOR	dst = src1 ^ src2
Branch (Absolute)	br = address
Branch (Relative)	br = ipe + offset
Conditional Branch (Absolute)	br =[cond] address
Conditional Branch (Relative)	br =[cond] ipe + offset
Check Queued PT Request	a15 = comm & 0x1\\29
Request a Packet Transfer	comm = comm | 0x1\\28

Table 8–3. PP Syntax For Various PP Functions (Continued)

Function	PP Syntax
Clear Bit	dst = src1 & ~(0x1\\src2)
Invert Bit	dst = src1 ^(0x1\\src2)
Set Bit	dst = src1 | (0x1\\src2)
Test Bit	dst = src1 & (0x1\\src2)
Clear Interrupts (the W bit must be = 0)	intflg = intflg
Disable Interrupts	dint
Enable Interrupts	eint
Byte Replicate	dst =rb src
Halfword Replicate	dst =rh src
Load (Memory to Register)	dst = *address
Load immediate	dst = &*(imm)
Move (Register to Register)	dst = src
Pop (Global Address Unit)	dst = *a14++
(Local Address Unit)	dst = *a6++
(Either Address Unit)	dst = *sp++
Push (Global Address Unit)	*—a14 = src
(Local Address Unit)	*—a6 = src
(Either Address Unit)	*—sp= src
Store (Register to Memory)	address = src
Swap	src1 = src2 ||src2 = src1
Leftmost Bit Change	dst = src2 + lmbc(src1)
Leftmost One	dst = src2 + lmo(src1)
Rightmost Bit Change	dst = src2 + rmbc(src1)
Rightmost One	dst = src2 + rmo(src1)

8.2 PP Data Unit Operators

This section describes the PP data unit operators. Table 8–4 lists the topics discussed in this section.

Table 8–4. Discussion of the PP Assembly Language Operators

Section	Topic
8.2.1	Operator Precedence
8.2.2	The Assignment Operators
8.2.4	The Arithmetic Operators
8.2.5	The Multiplication Operators
8.2.6	The Bitwise Boolean Operators
8.2.7	The Expand Operator
8.2.8	The Mask Generator Operator
8.2.9	The Rotate Operator
8.2.10	The Shift Operators

8.2.1 Operator Precedence

The first step in learning the PP assembly language is to become familiar with the data unit operators. The PP assembly language uses arithmetic and Boolean operators similar to those in C. In addition, the following operators are introduced:

Operator Name	Operator
Multiple flags expander	@mf
Mask generator	%
Left rotate	\\
Conditional assignment	=[cond]
Conditional assignment with status protection	[=cond.pro]

Table 8–5 contains a list of the PP assembly language operators in order of precedence. Operators not separated by a horizontal line are of equal precedence and are evaluated left to right.

The assembler also accepts other operators for use in arithmetic expressions that are used with directives. See subsection NO TAG, *Operators*, in the *MVP Code Generation Tools User's Guide* for more information about these operators.

Table 8–5. PP ALU Operator Precedence

Operator	Function	See Page
src1[n]src1–1	Selection of either the odd (n is true) or even (n is false) register in a D register pair based on the negative condition code	PP: 8-17
( )	Subexpression delimiters	PP: 8-21
@mf	Expander operator (for expanding each bit to a byte, halfword or word)	PP: 8-22
%	Mask generator	PP: 8-24
%%	Nonmultiple mask generator (EALUs only)	PP: 8-24
%!	Modified (shift right) mask generator, 0xFFFF FFFF output for 0 input. Same as % for nonzero input.	PP: 8-24
%%!	Nonmultiple shift right mask generator (EALUs only)	PP: 8-24
\\	Rotate left	PP: 8-26
<<	Shift left (assembler pseudo-operator for rotate & mask)	PP: 8-27
>>u	Unsigned shift right	PP: 8-27
>> or >> s	Signed shift right	PP: 8-29
&	Bitwise AND	PP: 8-20
^	Bitwise XOR	PP: 8-20
|	Bitwise OR	PP: 8-20
+	Addition	PP: 8-18
–	Subtraction	PP: 8-18
=[cond]	Conditional assignment	PP: 8-15
=[cond.pro]	Conditional assignment with status protection	PP: 8-16
=	Equate	PP: 8-15

8.2.2 The Assignment Operators

The **=** symbol is the assignment operator. As shown below, operations (with the exception of a few miscellaneous operations described in Section 8.7) are represented by an expression to the right-hand side of the assignment operator and a destination register on the left-hand side.

```
dst = expression
```

8.2.2.1 Conditional Assignment Operators

The assignment of the result of an operation can be performed conditionally depending on the contents of the status register at the beginning of the execute stage of the instruction.

Conditional assignment is represented by ⟦**cond**⟧, where **cond** is one of the 16 condition codes shown in Table 8–36. If the condition is true, the result of the expression on the right-hand side of the assignment operator is written to the destination register. Notice that the ⟦ symbol is used to represent required square brackets for the cond operator; normally, squared brackets represent optional fields.

If the condition is not true, the result of the expression on the right-hand side is not written to the destination register, and status setting is not performed. As discussed in Section 8.9, *Conditional Operations*, conditional assignment is also available for register-to-register moves and loads.

Example 8–1. Conditional Assignment

```
;  BEFORE:      d6 = 32

   a15 = d6 – 31      ; Compare d6 to 31

   d6 = [le] d6 + 1   ; Increment d6 if it is less than or
                      ;  equal to 31

;  AFTER:    d6 = 32
;            (Since d6 was not less than or equal to 31,
;            the result of the increment of d6 was not
;            written to d6.)
```

8.2.2.2 Conditional Assignment Operators With Status Protection

When conditional assignment is specified by the ALU operation, individual status bits can be protected from modification. Status protection is specified in the assembly syntax by ⟦**cond.pro**⟧,where **cond** is one of the 16 condition codes shown in Table 8–36 and **pro** is a lowercase list of the status bits ([n] [c] [v] [z]) you want protected.

For example, =[p.nz] would modify the destination register if the status in sr is positive, and the n and z bits would be preserved, regardless of whether or not these bits would be affected normally.

As shown in Example 8–2, if you want status protection for an unconditional ALU operation, you can use the unconditional condition code u. Status protection is described in detail in subsection 8.9.4, *Status Bit Protection*.

Example 8–2. Status Protection of the Negative Status Bit

```
;  BEFORE:      d6 = –10
;               d7 = 5
;               sr = 0 (N=0)

   d7 = [u.n] d7 + d6    ;Unconditional add with
                         ;protection of the negative
                         ;status bit

;  AFTER:       d7 = –5
;               sr = 0(Even though result was negative,
;                      negative status bit is not
;                      modified.)
```

Note:

The assembler also accepts the [.ncvz] status bits for unconditional assignments with status protection.

8.2.3 Conditional Source Selection Operator

A PP ALU expression can involve a conditional selection between the odd and even register in a D register pair. The odd register (d7, d5, d3, or d1) specified by src1 is selected if the negative status bit is 1; the corresponding even register (d6, d4, d2, or d0, respectively) is selected if the negative status bit is 0. As described in subsection 8.9.3, *Conditional Source Selection*, conditional source selection is also available for stores.

Example 8–3. Conditional Source Selection

```
;  BEFORE:   d5 = 10
;            d6 = 1
;            d7 = 3
;            sr = 0x80000000 (N = 1)

   d5 = d5 + d7[n] d6    ; Add d7 (if n=1) or d6 (if n=0)
                         ; to d5

:  AFTER:    d5 = 13  (Since negative status bit was set,
;                       d7 was added to d5.)
```

8.2.4 The Arithmetic Operators

The PP's arithmetic operators are:

- **Addition: +**
- **Subtraction: –**

The types of supported expressions that use the arithmetic operators (+ and –) are described in subsection 8.3.2.2, *The Arithmetic Operations*.

8.2.5 The Multiplication Operators

In a PP instruction you can use *****, the multiplication operator, to specify a multiplication operation between the contents of two D registers, assigning the result to a D register. Multiplication by an immediate value is not supported.

Subsection 8.5.1, *Using the MPY||ADD and MPY||SADD Operations*, describes 16-bit by 16-bit multiplications with a 32-bit result that can be performed in parallel with adds or subtracts with or without shifts in the ALU.

Subsection 8.5.2, *Multiplying in Parallel With an EALU Operation,* describes both a 16-bit by 16-bit multiply with a 32-bit result or two 8-bit by 8-bit multiplies with two 16-bit results that can be performed in parallel with extended ALU operations.

```
;  dst =[sign] src1 * src2
   d7 = d6 * d5             ; signed multiply
   d7 =u d6 * d5            ; unsigned multiply

;  dst:   D register destination
:  src1:  D register source 1
;  src2:  D register source 2

;  [sign]:  Optional designation signed (=s)  or
;           unsigned (=u) multiply.
;           Signed multiply is the default.
```

8.2.6 The Bitwise Boolean Operators

The PP assembly language supports the following bitwise Boolean operators (same as those operators used in the C programming language):

- **The bitwise AND operator: &**

  You can use the AND operator to mask off a particular set of bits or to test individual bits, as shown in Example 8–4.

Example 8–4. Bitwise AND Operations

```
; BEFORE:    d7 = 0x01234567
;            comm = 0x20000000

   d7  = d7 & 0xFF          ; Zero all but bits 7–0 in
                            ; d7.
   a15 = comm & 0x20000000  ; Test comm bit 29.

;  AFTER:    d7 = 0x00000067
;            Z status bit = 0
```

- **The bitwise OR operator: |**

  You can use the OR operator to set bits, as shown in Example 8–5.

Example 8–5. Bitwise OR Operations

```
;  BEFORE:   comm = 0x00000001

   comm = comm | 0x10000000 ; Set comm bit 28 to 1.

;  AFTER:    comm = 0x10000001
```

- **The bitwise exclusive OR (XOR) operator: ^**

  You can use the exclusive OR operator to set a 1 in each bit position where its operands have different bits, and 0 where they are the same, as shown in Example 8–6.

Example 8–6. Bitwise XOR Operation

```
;  BEFORE:   d6 = 0x12345678
;            d5 = 0x0000FFFF

   d7 = d6 ^ d5

;  AFTER:    d7 = 0x1234A987
```

❑ The unary 1s complement (NOT) operator: ~

You can use the 1s complement operator to convert each 1 bit to a 0 bit and vice versa, as shown in Example 8–7.

Example 8–7. 1s-Complement Operation

```
;BEFORE:                d7=0x01234567
    d7 = ~d7
;AFTER:                 d7=0xFEDCBA98
```

As described in subsection 8.3.2.1, the PP ALU supports any expression that is a Boolean combination of three or fewer inputs to the PP's three-input ALU.

In Example 8–8, a merge operation is shown. Bits from one of two registers are combined according to a mask in another register.

Example 8–8. Merge Operation

```
; BEFORE:                d7=0xFF0000FF
;                        d6=0x01234567
;                        d5=0xFEDCBA98
    d6 = (d6&d7) | (d5&~d7)
; AFTER:                 d6=0x01DCBA67
```

8.2.7 The Expand Operator

You can use **@mf**, the expand operator, to create a mask from the multiple status flags in mf. The expand operator facilitates color expand, saturate, and motion estimation algorithms, as shown in the examples in Chapter 15, *Software Applications*.

The expand operator can be applied only to the mf (multiple flags) register's contents. This operator expands bits from the mf register. For example, a bit value of 1 in the mf is expanded to a byte, halfword, or word of all 1s, according to the data size specified in the Msize (bits 5–3) field of the sr (status register).

Similarly, a 0 bit is expanded to a byte, halfword, or word of all 0s. Given a data size of n bits, the resulting 32-bit mask is based on 32/n bits from the mf register.

- If the R bit in the status register is 0, then the mf bits that will be expanded are the LSBs.
- If the R bit in the status register is 1, then the mf bits that will be expanded are the MSBs.

Table 8–6 summarizes the output of the expand operator corresponding to all the possible inputs.

The merge operation, shown in Example 8–8, can be modified to use the expand operator, where the LSBs of the mf register contain flags that choose between one of the two input fields.

Table 8–6. Output From the Expand Operator (@mf)

Values for mf Bits 3 2 1 0 (R=0) or Bits 31 30 29 28 (R=1)	Msize =4 (Byte)	Values for mf Bits 3 2 1 0	(R=0) Msize =5 (Halfword)	Values for mf Bits 3 2 1 0	(R=0) Msize =6 (Word)
0000	0x0000 0000	xx00	0x0000 0000	xxx0	0x0000 0000
0001	0x0000 00FF	xx01	0x0000 FFFF	xxx1	0xFFFF FFFF
0010	0x0000 FF00	xx10	0xFFFF 0000		
0011	0x0000 FFFF	xx11	0xFFFF FFFF		
0100	0x00FF 0000				
0101	0x00FF 00FF				
0110	0x00FF FF00				
0111	0x00FF FFFF	**Values for mf Bits 31 30 29 28**	**(R=1) Msize =5 (Halfword)**	**Values for mf Bits 31 30 29 28**	**(R=1) Msize =6 (Word)**
1000	0xFF00 0000				
1001	0xFF00 00FF	00xx	0x0000 0000	0xxx	0x0000 0000
1010	0xFF00 FF00	01xx	0x0000 FFFF	1xxx	0xFFFF FFFF
1011	0xFF00 FFFF	10xx	0xFFFF 0000		
1100	0xFFFF 0000	11xx	0xFFFF FFFF		
1101	0xFFFF 00FF				
1110	0xFFFF FF00				
1111	0xFFFF FFFF				

Example 8–9. Merge Operation Using the Expand Operator (@mf)

```
;BEFORE:     mf   =    0x9          (1001 binary)
;            sr   =    0x20         (Msize = byte, R = 0)
;            d6   =    0x01234567
;            d5   =    0xFEDCBA98
;            d1   =    0x00000000
             d1   =    (d6 & @mf) | (d5 & ~@mf)
;AFTER:      d1   =    0x01DCBA67
```

8.2.8 The Mask Generator Operator

You can use **%**, the mask generator operator, to generate a mask with the number of 1s specified in the five LSBs of the source register or in the immediate value, right-justified with zero extension. If the mask generator is supplied with a 5-bit quantity of value n, the resulting mask will have 1s in the n LSBs (or n right-justified 1s), and zeros in the 32 – n MSBs (in other words, $\%n = 2^n - 1$).

For multiple arithmetic, the mask generator operator creates a mask composed of the number of 1s specified in the 5, 4, or 3 LSBs (word, halfword, or byte data sizes, respectively, according to Asize, sr bits 2–0) of the source register or the immediate value, right-justified with zero extension and replicated in each word, halfword, or byte in the word. Table 8–7 summarizes the output of the mask generator for each possible input.

For EALU operations, you can append a second mask generator operator to the mask generator as a modifier (%%). This indicates that a single 32-bit (nonmultiple) mask will be produced, even when you are using multiple arithmetic. The %% form of the mask generator is controlled by the N bit in d0.

You can use the **right-shift-mask modifier** (**!**) to generate all 1s from an input of 0 (rather than generating all 0s). For inputs other than 0, the mask generator produces the same output as is produced without the right-shift-mask modifier. Example 8–10 shows the effects of using a shift right mask modifier. Refer to subsection 8.3.4.5 for more information on the shift right mask modifier.

Example 8–10. Using a Mask Generator Operator (%) With a Shift Right Mask Modifier (!)

```
%0 = 0x00000000
%!0 = 0xFFFFFFFF
%7 = 0x0000007F
%!7 = 0x0000007F
```

You must use the right-shift mask modifier for the mask generator in conjunction with the rotate operator to perform variable shift right operations, as described in subsection 8.2.10, *The Shift Operators*.

You can specify the shift right mask modifier (!) only for an arithmetic ALU operation or EALU operation. You cannot use the right-shift mask modifier with the base set of Boolean operations.

The mask generator cannot be used to control carry-in when bits A and E in the d0 register are both set to 1.

Table 8–7. Mask Generator Output

Five LSBs of Input (43210)	Nonmultiple	Multiple		
		Asize = Byte	Asize = Halfword	Asize = Word
00000	0x0000 0000	0x0000 0000	0x0000 0000	0x0000 0000
00001	0x0000 0001	0x0101 0101	0x0001 0001	0x0000 0001
00010	0x0000 0003	0x0303 0303	0x0003 0003	0x0000 0003
00011	0x0000 0007	0x0707 0707	0x0007 0007	0x0000 0007
00100	0x0000 000F	0x0F0F 0F0F	0x000F 000F	0x0000 000F
00101	0x0000 001F	0x1F1F 1F1F	0x001F 001F	0x0000 001F
00110	0x0000 003F	0x3F3F 3F3F	0x003F 003F	0x0000 003F
00111	0x0000 007F	0x7F7F 7F7F	0x007F 007F	0x0000 007F
01000	0x0000 00FF	0x0000 0000	0x00FF 00FF	0x0000 00FF
01001	0x0000 01FF	0x0101 0101	0x01FF 01FF	0x0000 01FF
01010	0x0000 03FF	0x0303 0303	0x03FF 03FF	0x0000 03FF
01011	0x0000 07FF	0x0707 0707	0x07FF 07FF	0x0000 07FF
01100	0x0000 0FFF	0x0F0F 0F0F	0x0FFF 0FFF	0x0000 0FFF
01101	0x0000 1FFF	0x1F1F 1F1F	0x1FFF 1FFF	0x0000 1FFF
01110	0x0000 3FFF	0x3F3F 3F3F	0x3FFF 3FFF	0x0000 3FFF
01111	0x0000 7FFF	0x7F7F 7F7F	0x7FFF 7FFF	0x0000 7FFF
10000	0x0000 FFFF	0x0000 0000	0x0000 0000	0x0000 FFFF
10001	0x0001 FFFF	0x0101 0101	0x0001 0001	0x0001 FFFF
10010	0x0003 FFFF	0x0303 0303	0x0003 0003	0x0003 FFFF
10011	0x0007 FFFF	0x0707 0707	0x0007 0007	0x0007 FFFF
10100	0x000F FFFF	0x0F0F 0F0F	0x000F 000F	0x000F FFFF
10101	0x001F FFFF	0x1F1F 1F1F	0x001F 001F	0x001F FFFF
10110	0x003F FFFF	0x3F3F 3F3F	0x003F 003F	0x003F FFFF
10111	0x007F FFFF	0x7F7F 7F7F	0x007F 007F	0x007F FFFF
11000	0x00FF FFFF	0x0000 0000	0x00FF 00FF	0x00FF FFFF
11001	0x01FF FFFF	0x0101 0101	0x01FF 01FF	0x01FF FFFF
11010	0x03FF FFFF	0x0303 0303	0x03FF 03FF	0x03FF FFFF
11011	0x07FF FFFF	0x0707 0707	0x07FF 07FF	0x07FF FFFF
11100	0x0FFF FFFF	0x0F0F 0F0F	0x0FFF 0FFF	0x0FFF FFFF
11101	0x1FFF FFFF	0x1F1F 1F1F	0x1FFF 1FFF	0x1FFF FFFF
11110	0x3FFF FFFF	0x3F3F 3F3F	0x3FFF 3FFF	0x3FFF FFFF
11111	0x7FFF FFFF	0x7F7F 7F7F	0x7FFF 7FFF	0x7FFF FFFF

8.2.9 The Rotate Operator

All shifts and rotates in the PP are accomplished via a 32-bit barrel rotator that takes as input the left rotate amount. Since C does not have a left rotate symbol, \\ was chosen for this operation. Note that a right rotate can be accomplished by left rotating by the 2s complement of the desired right rotate amount. Subsection 8.2.10, *The Shift Operators*, explains shift pseudo-operators that evaluate to rotate/mask operations.

8.2.10 The Shift Operators

You can perform shifts on the PP by combining the barrel-rotated input to the B port with mask information on the C port using the ALU. Signed shift right (>> (default) or >>s) is treated as an operator allowed in the class-specific functions presented in Table 8–12. As described in the following subsections, shift left (<<) and unsigned shift right (>>u) are assembler pseudo-operators.

8.2.10.1 Shift Left Pseudo-Operator

On the PP, a shift left is performed by combining the barrel rotated input to the B port with mask information on the C port. When the assembler encounters an expression using the left shift symbol (<<), it is converted by the following pseudo-operator expansion.

b << n :–> b\\n & ~%n

b: Register input to the barrel rotator.
n: Either a constant or the 5 LSBs contained in a register.

8.2.10.2 Unsigned Shift Right Pseudo-Operator

The unsigned shift right symbol (>>u) is also an assembler pseudo-operator. Since the shift right pseudo-operator expansion includes %! for shift amounts in a register, which requires a function modifier, it can be used only with a base set arithmetic or an EALU operation. It cannot be used with base set Booleans and a register shift amount. A shift right by a constant amount other than zero can be performed for Boolean functions by using the full rotate and merge expression with % instead of %!.

The unsigned shift right pseudo-operator (and signed shift right operator) evaluates differently, according to whether the shift amount is specified as a constant or in a register.

- Register shift amount

  Since the input to the barrel rotator specifies the number of bits to rotate left, the register that specifies the shift amount for a shift right actually contains the 2s complement of the shift right amount that will result.

  In order to remind you of this fact, the assembler requires a minus sign, –, before the shift amount register source. The unsigned shift right pseudo-operator expansion for a register sourced shift amount is as shown in Example 8–11.

Example 8–11. Using the Unsigned Shift Right Pseudo-Operator With a Register Shift Amount

```
b >>u –dj :–> (b\\dj) & ~%!dj
```

b: The register input to the barrel rotator.

dj: The D register that contains the 2s complement of the resulting shift right. The 5 LSBs in dj are used as the (left) rotate amount and as the input to the mask generator to produce the desired shift right operation.

- Constant shift amount

  When the shift right amount is specified by a constant in the source code (for example, >> 4), the assembler automatically encodes the 2s complement of the constant (32 – 4 = 28) in the instruction as the input to the barrel rotator and the mask generator.

  The desired shift right by 4 is then performed by a left rotate by 28 (the five-bit 2s complement of 28) followed by a logical AND with a mask created by %28 (0s in the four MSBs and 1s in the 28 LSBs) to clear the four MSBs.

  The unsigned shift right pseudo-operator expansion for a shift right by a constant is as shown in Example 8–12.

Example 8–12. Using the Unsigned Shift Right Pseudo-Operator With a Constant Shift Amount

```
                    { (b\\(32 – constant)) & %(32 – constant)   ; if constant ≠ 0
b >>u constant : →  {
                    { b                                         ; if constant = 0
```

b: The register input to the barrel rotator.

constant: Constant is the desired shift right amount. The assembler encodes the 2s complement of this constant, (32-constant), as the (left) rotate amount and as the input to the mask generator to produce the desired shift right operation.

8.2.10.3 Signed Shift Right

To perform a signed shift right (>>s), special logic is required to modify the ALU function according to the input sign. This operation is treated as a class-specific arithmetic function rather than a pseudo-operator. See page PP:8-41 for more information about class-specific arithmetics.

Note that the signed shift right function does not allow multiple arithmetic.

The distinction between register sourced and constant shift amounts is the same as discussed for unsigned shift right. If a constant shift amount is specified, the assembler performs a 2s complement to form the required immediate. This allows you to specify the desired shift right amount in the source code.

For a register shift amount, the shift right amount is the 2s complement of the contents of the five LSBs of the register. The assembler requires a minus sign (–) before the register containing the shift amount to remind you of this.

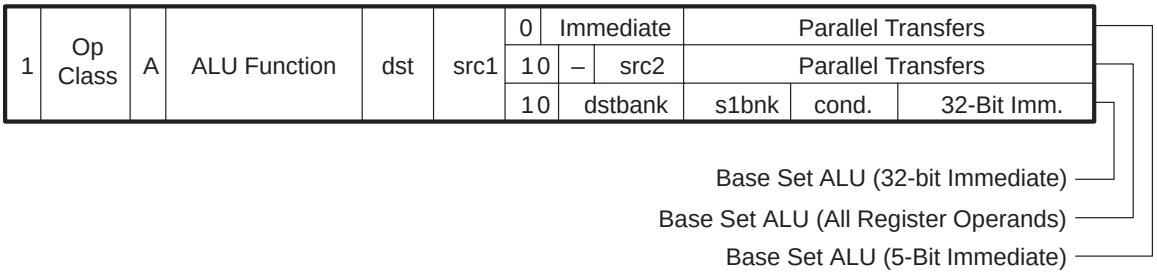
If you do not specify **s** or **u** at the end of the shift right operator, then it defaults to signed shift right.

8.3 The Base Set of ALU Operations

Figure 8–1 shows the opcode format for the base set of ALU operations (base set ALUs). This opcode format allows three operands to be specified directly in the opcode.

The src2 operand can be either a 5-bit immediate, a 32-bit immediate, or a D register. If src2 is a 32-bit immediate, the full opcode is used to specify the data unit operation; therefore, no parallel transfers can be specified.

Figure 8–1. Base Set ALU Opcode Formats



A base set ALU operation is specified by three elements:

- **Op Class (Operation Class)**

  Specifies one of eight different operand routings for the ALU's data path. The operation class code determines the ALU input data path for each of the opcode operands and can also specify operands associated directly with the operation class. The operation class allows different ALU data path operations to be performed by the same ALU function.

- **A (Arithmetic Enable)**

  The A bit selects between arithmetic and Boolean operations of the ALU. If A = 1, then ALU carry-generate is enabled and the 8-bit ALU code specifies an arithmetic operation; if A = 0, a Boolean operation (no carry-generate) is specified.

- **ALU Function**

  If A = 0, the 8-bit ALU function code specifies one of the 256 possible Boolean combinations of three inputs. If A = 1, the four odd bits specify an arithmetic ALU operation, and the four even bits specify ALU function modifiers, such as split ALU arithmetic.

Table 8–8 lists the topics discussed in this section.

Table 8–8. Discussion of the Base Set of ALU Operations

Section	Topic
8.3.1	ALU Operands
8.3.2	Boolean and Arithmetic Functions
8.3.3	Operation Classes
8.3.4	ALU Function Modifiers
8.3.5	Summary of the ALU Function Modifier Codes

8.3.1 ALU Operands

For the base set ALU format shown in Figure 8–1, three operands are specified directly in the instruction:

- **Source operand 1 (src1)**
- **Source operand 2 (src2 or immediate data)**
- **Destination operand (dst)**

As discussed in subsection 8.3.3, *Operation Classes*, the operation class (data path combination) determines where in the ALU data path the source operands are routed. For example, one operation class uses src1 as the input to the barrel rotator; another operation class uses src1 as the input to the A port of the ALU.

The operation class also defines some implied inputs. For example, certain operation classes use the default rotate amount in d0 as the barrel rotate amount. Another operation class uses src2 as the rotate amount but feeds the A port with the contents of the ALU destination companion register.

Here are some guidelines regarding base set ALU operands:

- src2 can be a D register (specified by src2 in the opcode), a 5-bit immediate, or a 32-bit immediate. If a 32-bit immediate is used, parallel transfers and status protection cannot be specified.
- src1 and dst can, in some cases, be a non-D register. When a non-D register operand is used, you cannot specify a global address unit operation in the same instruction.
- The numbering of the source operands (src1 and src2) does not imply any required ordering for the operands in the algebraic expression used to specify an operation. The assembler automatically associates operands with their associated fields in the opcode.
- More than one operand can specify the same register.

8.3.1.1 Register Operands

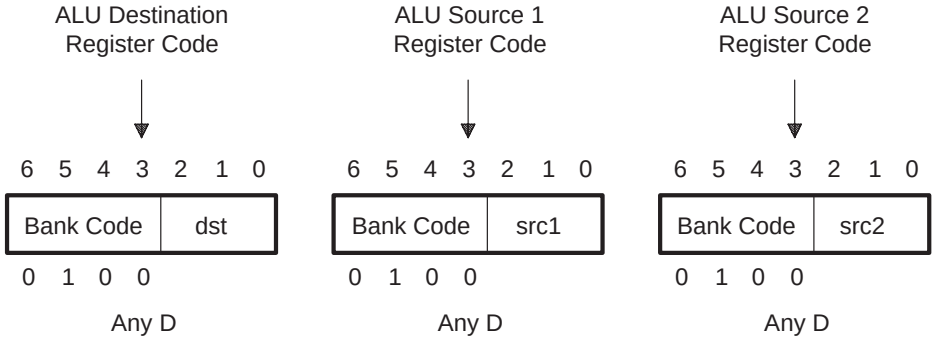
Each PP register is identified by a 7-bit register code (see Table 7–1). The four MSBs of the register code define the register type and are referred to as the register bank. The three LSBs of the register code specify the individual register within the bank of registers defined by the four MSBs.

For example, register a8 is represented by the register code 0001 000. The four MSBs (0001) specify the register bank for global address registers (a8–a12 and a14–a15). The three LSBs (000) select a8 from the bank of global address registers.

- **D registers**

  Base set ALU operands are represented by 3-bit codes dst, src1, and src2 in the data unit portion of the opcode. These codes specify the three LSBs of the corresponding ALU operand. Unless additional bits are specified in the global transfer portion of the opcode, these operands are assumed to be D registers (the bank code for D registers, 0100, is used by default).

Figure 8–2. ALU D Register Operands



❑ Non-D registers

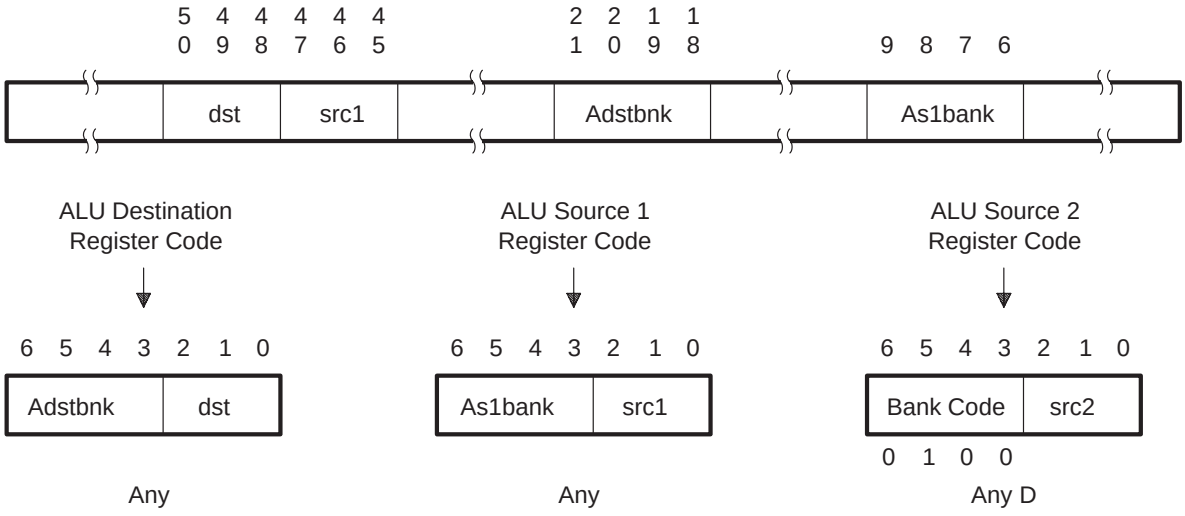
To specify a non-D register for the ALU src1 operand and/or dst operand, use the global transfer portion of the opcode to specify the register bank to combine with the three LSBs of the register code indicated in src1 and/or dst. Thus, when an ALU operation uses a non-D register operand, a global transfer cannot be specified in parallel, because the opcode bits required to specify a global transfer are used instead to specify the register banks for the ALU dst and src 1.

As shown in Figure 8–3, the As1bank field can be combined with the src1 operand to define the full 7-bit register code for a non-D source 1 for the ALU; the Adstbnk field can also be combined with the dst operand to define the full 7-bit register code for the destination of the ALU result.

There are some restrictions on using non-D register operands:

- Non-D registers cannot be input to the bit-detection logic.
- Non-D registers cannot be the destination of an operation involving the bit-detection logic.

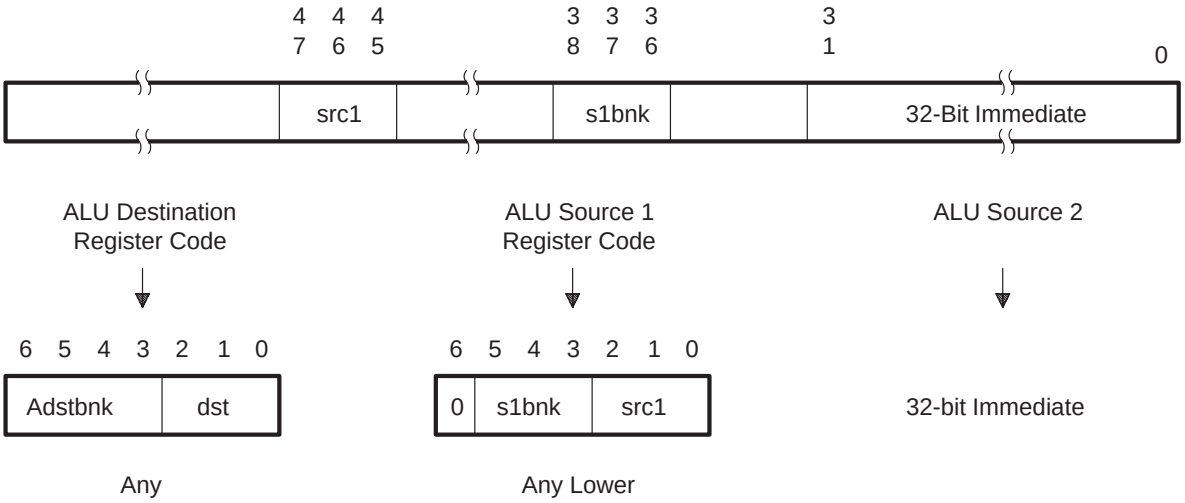
Figure 8–3. Non-D Register ALU Operand Opcode Fields



❑ Lower and upper registers

For base set ALU operations with a 32-bit immediate operand, only three bits (s1bnk) are available to assign the bank for src1 (as shown in Figure 8–4). For this case, the upper bit of the register code is assumed to be 0. This creates a group of registers referred to as **lower registers**. All PP registers are lower registers except the cache tag registers and all but one of the loop control registers (lctl is a lower register). PP registers whose MSB is set are **upper registers**. Table 8–9 indicates the lower and upper registers.

Figure 8–4. Lower Register Source Operand for Base Set ALU With 32-Bit Immediate



❑ Companion registers

As discussed in subsection 8.3.3, some operation classes allow both the input to the A port of the ALU and the ALU destination to be derived from the dst operand in the opcode. For this case, the A port input is always the D register specified by the dst 3-bit code. However, the ALU destination can optionally use the 4-bit Adstbnk field to specify a register bank other than the D registers. This allows four different operands to be used in a restricted sense. While the ALU destination and A port source operands can be from different banks, they must have the same lower 3-bit code. Throughout this user's guide, the term companion registers is used to describe registers with the same register code in the three LSBs. Table 8–9 shows the companion D register corresponding other PP registers.

Figure 8–5. Companion Registers

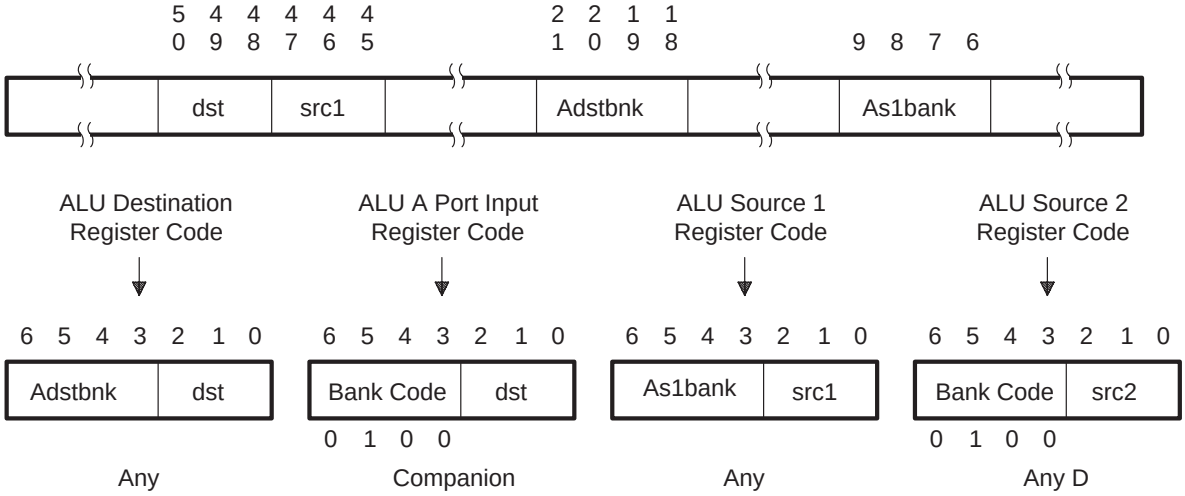


Table 8–9. Companion D Registers, Lower and Upper Registers

Companion D Register	Lower Registers		Upper Registers			
d0	a0 a8 x0 x8	pc/call	lc0	lrse0	ls0	
d1	a1 a9 x1 x9 sr	br/ipa	lc1	lrse1	ls1	
d2	a2 a10 x2 x10 mf	ipe	lc2	lrse2	ls2	
d3	a3 a11	iprs				
d4	a4 a12	inten	lr0	lrs0	le0	tag0
d5		intflg	lr1	lrs1	le1	tag1
d6	a6/sp a14/sp	comm	lr2	lrs2	le2	tag2
d7	a7/0 a15/0	lctl				tag3

8.3.1.2 Immediate Operand

For each operation class, the src2 operand can be an immediate. The PP opcode supports two immediate data sizes: 32 bits and unsigned 5 bits.

❑ **32-Bit Immediate**

A 32-bit immediate can be used as src2, but the following restrictions apply:

- No parallel transfers can be specified, because that portion of the opcode is used to specify the immediate data.
- The operation can be conditional but cannot specify status protection.
- The source operand 1 is restricted to a lower register.

❑ **Unsigned 5-Bit Immediate**

ALU source operand 2 (src2) can be an unsigned 5-bit immediate. No restrictions are imposed on using a 5-bit immediate instead of a D register for src2.

8.3.2 Boolean and Arithmetic Functions

The arithmetic enable bit (A) and the ALU function code shown in Figure 8–6 allow the following ALU functions to be specified:

- When A = 0, it specifies Boolean operations (no carry generate), and the 8-bit ALU function code enables any of the 256 possible Boolean combinations of the three inputs to the ALU.

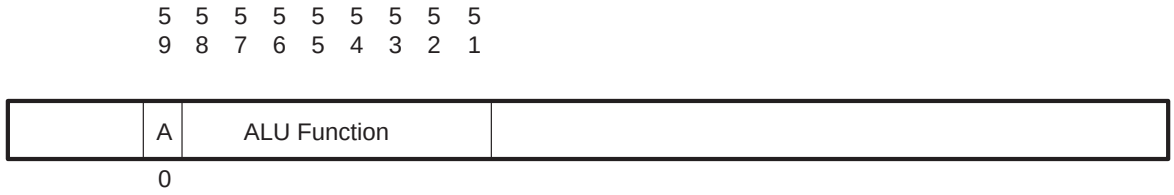
  Refer to subsection 8.3.2.1 for more information on Boolean operations.

- When A = 1, it specifies arithmetic operations (carry generate), and a 4-bit code (FUNC) lets you use 16 different ALU function code settings. The meanings of the bits are not important for programming the PP, but these meanings are listed in Table A–5, *Base Set Arithmetics*. Some of these codes have different meanings, depending on the operation class being used (see subsection 8.3.3, *Operation Classes*).

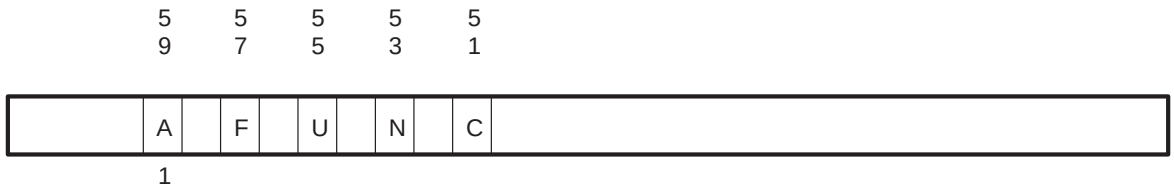
  Refer to subsection 8.3.2.2 for more information on arithmetic operations.

Figure 8–6. ALU Function Code

Base Set Booleans:



Base Set Arithmetics:



8.3.2.1 The Boolean Operations

The PP ALU can perform any of the 256 bitwise Boolean combinations of three inputs in a single cycle. The PP assembler has a built-in Boolean expression evaluator so that any Boolean operation can be specified.

The set of three-input Boolean operations includes all sixteen two-input Booleans as a subset. Table 8–10 shows some examples of three-input Boolean functions.

One of the most useful three-input Booleans is the multiplexer function: (A&C)|(B&~C). This function merges the input data of the A and B ports (that is, it chooses A or chooses B) on a bit-by-bit basis according to the input to the C port.

Color expand, MAX, MIN, z-buffering, saturate, and patterning can all be accomplished on the PP with this Boolean function and the appropriate operation class. Refer to Chapter 15, *Software Applications*, for specific examples.

Table 8–10. Some Interesting Three-Input Booleans

Expression	Function
A&C | B&~C	Merge A and B on a bit-by-bit basis according to C (used for color expand, MAX, MIN, z-buffering, saturate, and patterning)
A&B|B&C|A&B	Majority (2 of 3 bits are a 1)
A&(~(B^C))	Binary morphology (bit correlation) erosion operator (blob-shrinking operator)
A|(B&C)	Binary morphology dilation operator (blob-expansion operator)

8.3.2.2 The Arithmetic Operations

The PP ALU supports 256 possible arithmetic functions (arithmetics). A key subset of arithmetic functions (referred to as the base set arithmetics) is supported in the base set of ALU operations. Additional useful arithmetic functions are supported by the extended ALU opcode format, as described in Section 8.4, *The Extended Set of ALU Operations (EALU)*.

The base set arithmetics are divided into two categories: class-independent arithmetics and class-specific arithmetics.

- **Class-independent arithmetics** are functions that can be specified for any of the eight operation classes. Table 8–11 summarizes the class-independent arithmetic functions.

Table 8–11. Class-Independent Base Set Arithmetic Functions

ALU Function
A + B
A + C
A – B
A – C
A + (B|C)
A – (B|C)
A + (B&C)
A – (B&C)
A + (B&~C)
A – (B&~C)
(A & C) + (B & C)

Notes: 1) A = Input to the A port of the ALU
2) B = Input to the B port of the ALU
3) C = Input to the C port of the ALU

The first four functions in Table 8–11 are two-input arithmetics. There is a distinction between AB arithmetic (for example, A + B) and AC arithmetic (for example, A + C) because the data paths into the B and C ports are different.

The next seven ALU functions in Table 8–11 involve arithmetic operations upon Boolean combinations of the ALU inputs. These mixed arithmetic and Boolean operations, when combined with the various operation classes, support extremely powerful merging, data translation, and bit-manipulation.

For example, since the input to the B port can be a rotated value and the input to the C port can be a mask generated from the rotate amount, A + (B&C) can be used to perform A plus a value that has been shifted right with zero extension.

- **Class-specific arithmetic** functions support sign-extended shift right and absolute value operations. You can use them only in the forms shown in Table 8–12. These forms imply certain operation classes.

Table 8–12. Class-Specific Base Set Arithmetic Functions

Algebraic Description	Typical Function
A + b>>s n	A, plus the signed shift right of the input to the barrel rotator b by n bits. n can be either an immediate or the 2s complement of the contents in a D register (for example, –d0).
A – b>>s n	A, minus the signed shift right of the input to the barrel rotator b by n bits. n can be either an immediate or the 2s complement of the contents in a D register (for example, –d0).
A +|B|	A plus the absolute value of B
A – |B|	A minus the absolute value of B
A + (B&@mf | –B&~@mf)	Multiple sum of absolute values
A + (–B&@mf | B&~@mf)	Multiple difference of absolute values

Note: Lowercase **b** represents the input to the barrel rotator. Uppercase **B** is the input to the ALU.

- **Signed shift right**. The first two functions in Table 8–12 are used specifically for signed shift right operations. You can specify the shift amount using a register or an immediate.

  - Register

    When the shift amount is specified in a register, the register contains the amount that the barrel rotator will rotate left. Thus, the resultant right shift is the 2s complement of the value specified in the register.

    The assembler requires a minus character (–) before the register specifying the shift amount. This serves as a reminder that the contents of the register must be the 2s complement of the desired shift right amount.

  - Immediate

    For immediate shift amounts, the assembler performs a 2s complement on the immediate data so that you can specify the desired right shift amount in the source code.

You cannot perform signed shift right operations with multiple arithmetic (see subsection 8.3.4.1, *The Multiple Arithmetic Modifiers*) or in conjunction with bit detection (see subsection 8.3.4.2, *The Bit-Detection Modifiers*).

- **Absolute value**. The next two functions in Table 8–12 support A plus the absolute value of B, and A minus the absolute value of B. Some restrictions apply to these functions:
  - They cannot be used with the two operation classes that specify expand operator (@mf) on the C port.
  - The cin ALU function modifier (see subsection 8.3.4.4, *The Carry-In Modifier*) cannot be specified with these functions.
  - These functions cannot be used with multiple arithmetic. Note that you can perform multiple absolute value operations by using the next set of functions listed in Table 8–12.
- **Multiple absolute value.** The final two functions in Table 8–12 allow you to perform either an add or subtract operation in each segment of the split ALU, as determined by the multiple status flags in the mf register. One example of such an operation is the multiple sum of absolute values (the core operation required by some motion estimation algorithms). Refer to Chapter 15, *Software Applications,* for appropriate software applications.

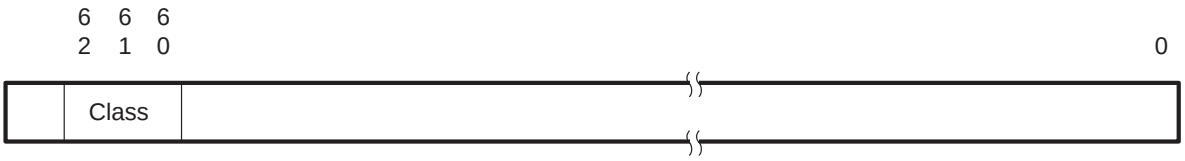
  These operations require the C input to be the expand operator (@mf) and thus are operation class-specific.

8.3.3 Operation Classes

In Figure 4–1, *Data Unit Block Diagram*, note that the multiplexers along the input paths to the ALU make many unique data path combinations possible. The eight data path combinations shown in Figure 4–3, *Base Set ALU Operation Classes*, are supported for the base set of ALU operations and are called **operation classes** because they affect the type of operation that is performed.

Figure 4–3 illustrates the ALU data routing for each of the operation classes. The operation class is specified directly in the PP opcode (see Figure 8–7) and can change every cycle.

Figure 8–7. Operation Class Field



8.3.3.1 Summarizing the Operation Classes

Table 8–13 summarizes the base set ALU operation classes. Note that the assembler determines the operation class required for an operation and associates the operands with the corresponding opcode field.

Table 8–13. Base Set ALU Operation Class Summary

Class	Dest	A Port Input	B Port Input	C Port Input
0	dst	src2	src1	@mf
1	dst	dstc	src1\\d0	src2
2	dst	dstc	src1	%src2
3	dst	dstc	src1\\src2	%src2
4	dst	src2	src1\\d0	%d0
5	dst	src2	src1\\d0	@mf
6	dst	dstc	src1	src2
7	dst	src1	1\\src2	src2

- **dst** can be any register. If it is a non-D register, some restrictions apply to which operations can be specified in parallel.
- **dstc** is the companion D register to the specified destination register (dst) in classes 1, 2, 3, and 6 and is used as the input to the A port. Companion D registers are described in subsection 8.3.1.2, *Immediate Operand*. Note that if dst is a D register, then dstc is that same D register (a D register is its own companion register).

- **src1** can be any register. If src1 is a non-D register, some restrictions apply to which operations can be specified in parallel.

  You can specify conditional source selection between an odd D register (d7, d5, d3, d1) specified in src1 or the corresponding paired even D register (d6, d4, d2, d0, respectively) on the basis of the n condition.

- **src2** can be either a D register or a 5-bit or 32-bit immediate. If it is a 32-bit immediate, no parallel operations can be specified.

- **\\d0** specifies rotation left by the default rotation amount specified in d0's five LSBs.

- **%d0** specifies mask generation based on the mask specified in d0's 5 LSBs.

- **@mf** specifies bit expansion of the LSBs (if sr⟦R⟧ = 0) or MSBs (if sr⟦R⟧ = 1) of the mf register.

The following general features make operation classes more flexible:

- Inputs to the ALU can be ignored, depending on the ALU function. For example, if the ALU function is A + B, the C input will be ignored.

- More than one operand can specify the same register.

The following section illustrates how the unique features of the various operation classes allow different operations to be performed by the same ALU function. Once you are familiar with the types of operations that are available, the operation class for a given operation is often transparent. The assembler selects the appropriate operation class for the specified operation.

8.3.3.2 Understanding Operation Class Influence on ALU Functioning

A PP ALU operation is the combination of the ALU function with the inputs specified by the operation class. This applies for both Boolean operations and arithmetics. This section shows how the operation classes work and gives examples of different operations that can be performed with ALU functions A + B and A + C.

Table 8–14 shows a supported A + B expression for each operation class. Note that since A + B does not involve a C port input, some of the classes result in the same operation.

Table 8–14. A + B for Each Operation Class With dst = d7, src1 = d3, src2 = d4

Expression	Operation Class
d7 = d4 + d3	Class 0
d7 = d7 + d3\\d0	Class 1
d7 = d7 + d3	Class 2 (subset of class 0 for AB functions)
d7 = d7 + d3\\d4	Class 3
d7 = d4 + d3\\d0	Class 4
d7 = d4 + d3\\d0	Class 5 (subset of class 4 for AB functions)
d7 = d7 + d3	Class 6 (subset of class 0 for AB functions)
d7 = d3 + 0x1\\d4	Class 7

Table 8–15 shows a supported A + C expression for each operation class. Note that although ALU functions A+B and A+C both specify an add, these two functions support different types of operations because of the different routing of inputs to the B and C ports.

Table 8–15. A + C for Each Operation Class With dst = d7, src1 = d3, src2 = d4

Expression	Operation Class
d7 = d4 + @mf	Class 0
d7 = d7 + d4	Class 1
d7 = d7 + %d4	Class 2
d7 = d7 + %d4	Class 3 (subset of class 2 for AC functions)
d7 = d4 + %d0	Class 4
d7 = d4 + @mf	Class 5 (subset of class 0 for AC functions)
d7 = d7 + d4	Class 6 (subset of class 1 for AC functions)
d7 = d3 + d4	Class 7

For ALU functions that involve all three ALU input ports, each operation class allows a unique ALU operation to be specified. For ALU functions that ignore one of more of the ALU inputs, not all of the operation classes result in unique operations.

Table 8–16 shows all of the unique base set arithmetic operations that can be specified by combining the class-independent functions with the different operation classes. Likewise, Table 8–17 shows the unique base set arithmetic operations that can be specified by combining the class-specific functions with their associated operation classes.

The purpose of these tables is not to serve as a reference but simply to demonstrate combining the operation classes with the ALU functions. The best way to determine the supported operations is to become familiar with Table 8–13 and use it when trying to decide what can be specified for a given ALU function.

Note that 256 base set Boolean functions can be combined with the 8 different operation classes, producing more than a thousand unique operations.

Table 8–16. Table of Class-Independent Base Set Arithmetic Operations

Addition Operations	Subtraction Operations
A + B dst = src2 + src1 dst = dstc + src1\\src2 dst = src2 + src1\\d0 dst = src1 + 0x1\\src2	**A – B** dst = src2 + src1 dst = dstc – src1\\src2 dst = src2 – src1\\d0 dst = src1 – 0x1\\src2
A + C dst = src2 + @mf dst = dstc + %src2 dst = src2 + %d0 dst = src1 + src2	**A – C** dst = src2 – @mf dst = dstc – %src2 dst = src2 – %d0 dst = src1 – src2
A + B | C dst = src2 + (src1 | @mf) dst = dstc + (src1\\d0 | src2) dst = dstc + (src1 | %src2) dst = dstc + (src1\\src2 | %src2) dst = src2 + (src1\\d0 | %d0) dst = src2 + (src1\\d0 | @mf) dst = dstc + (src1 | src2) dst = src1 + (0x1\\src2 | src2)	**A – B | C** dst = src2 – (src1 | @mf) dst = dstc – (src1\\d0 | src2) dst = dstc – (src1 | %src2) dst = dstc – (src1\\src2 | %src2) dst = src2 – (src1\\d0 | %d0) dst = src2 – (src1\\d0 | @mf) dst = dstc – (src1 | src2) dst = src1 – (0x1\\src2 | src2)
A + B & C dst = src2 + (src1 & @mf) dst = dstc + (src1\\d0 & src2) dst = dstc + (src1 & %src2) dst = dstc + (src1\\src2 & %src2) dst = src2 + (src1\\d0 & %d0) dst = src2 + (src1\\d0 & @mf) dst = dstc + (src1 & src2) dst = src1 + (0x1\\src2 & src2)	**A – B & C** dst = src2 – (src1 & @mf) dst = dstc – (src1\\d0 & src2) dst = dstc – (src1 & %src2) dst = dstc – (src1\\src2 & %src2) dst = src2 + (src1\\d0 & %d0) dst = src2 + (src1\\d0 & @mf) dst = dstc + (src1 & src2) dst = src1 + (0x1\\src2 & src2)
A + B & ~C dst = src2 + (src1 & ~@mf) dst = dstc + (src1\\d0 & ~src2) dst = dstc + (src1 & ~%src2) dst = dstc + (src1\\src2 & ~%src2) dst = src2 + (src1\\d0 & ~%d0) dst = src2 + (src1\\d0 & ~@mf) dst = dstc + (src1 & ~src2) dst = src1 + (0x1\\src2 & ~src2)	**A – B & ~C** dst = src2 – (src1 & ~@mf) dst = dstc – (src1\\d0 & ~src2) dst = dstc – (src1 & ~%src2) dst = dstc – (src1\\src2 & ~%src2) dst = src2 + (src1\\d0 & ~%d0) dst = src2 + (src1\\d0 & ~@mf) dst = dstc + (src1 & ~src2) dst = src1 + (0x1\\src2 & ~src2)
A & C + B & C dst = (src2 & @mf) + (src1 & @mf) dst = (dstc & src2) + (src1\\d0 & src2) dst = (dstc & %src2) + (src1 & %src2) dst = (dstc & %src2) + (src1\\src2 & %src2) dst = (src2 & %d0) + (src1\\d0 & %d0) dst = (src2 & @mf) + (src1\\d0 & @mf) dst = (dstc & src2) + (src1 & src2) dst = (src1 & src2) + (0x1\\src2 & src2)	

Note: **dst** can be any PP register. **dstc** refers to the companion D register for dst (which is the destination register). **src1** can be any register if it is not the input to barrel rotator and if src2 is not a 32-bit immediate. If src2 is a 32-bit immediate, then src1 must be a lower register. If src1 is the input to the barrel rotator, then it must be a D register. **src2** can be either a D register or an immediate.

Table 8–17. Table of Class–Specific Base Set Arithmetic Operations

A + |B| (Class 1, 2, 3, or 6)

dst = dstc + |src1|

A – |B| (Class 1, 2, 3, or 6)

dst = dstc + |src1|

A + ((B&@mf) | (–B&~@mf)) (Class 0 or 5)

dst = src2 + ((src1&@mf) | (–src1&~@mf))
dst = src2 + ((src1\\d0 & @mf) | (–src1\\d0 & ~@mf))

A – ((B&@mf) | (–B&~@mf)) (Class 0 or 5)

dst = src2 – ((src1&@mf) | (–src1&~@mf))
dst = src2 – ((src1\\d0 & @mf) | (–src1\\d0 & ~@mf))

A + b>>n (Class 3 or 4)

dst = dstc + src1>> –src2	if src2 is a D register
dst = dstc + src1>> constant	if src2 is an immediate
dst = src2 + src1>> –d0	

A – b>>n (Class 3 or 4)

dst = dstc – src1>> –src2	if src2 is a D register
dst = dstc – src1>> constant	if src2 is an immediate
dst = src2 – src1>> –d0	

Note: **dst** can be any PP register. **dstc** refers to the companion D register for dst (which is the destination register). **src1** can be any register if src2 is not a 32-bit immediate. If src2 is a 32-bit immediate, then src1 must be a lower register. **src2** can be either a D register or an immediate.

8.3.4 ALU Function Modifiers

Base set arithmetics can be modified by a four-bit ALU function modifier (FMOD) code shown in Figure 8–8. This modifier code specifies various ALU data path features:

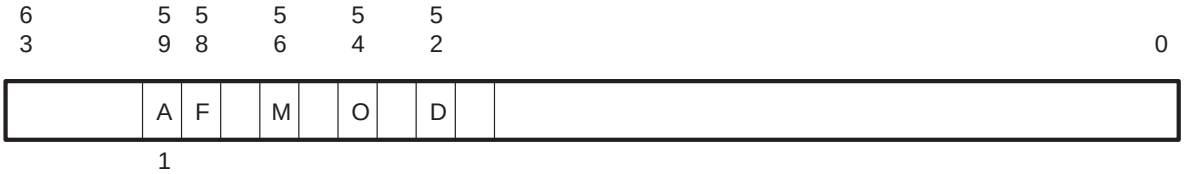
- The multiple arithmetic modifiers (split ALU)
- The bit-detection modifiers (lmo, rmo, lmbc, rmbc)
- The A port = 0 modifier
- The carry-in modifier
- The right shift mask modifier

For the base set ALU format, the ALU function modifier code applies only for arithmetic ALU functions (when the A bit is set to 1). ALU function modifiers cannot be applied to base set Booleans. Note that the ALU function modifiers described in this section can also be applied to EALU operations.

For EALU operations, the function modifier code is contained in d0 and can be applied to both Boolean and arithmetic ALU functions. See Section 8.4, *The Extended Set of ALU Operations (EALU),* for more information about EALU operations.

Knowing the bit assignments of function modifiers to the FMOD bits in not important for PP programming; however, these assignments are listed in Table A–6, *Class 0 to 7 Function Modifier Opcodes.*

Figure 8–8. Base Set Arithmetic Function Modifier



8.3.4.1 The Multiple Arithmetic Modifiers

Multiple arithmetic refers to a special ALU mode that allows the ALU to operate as four 8-bit ALUs, two 16-bit ALUs, or one 32-bit ALU. This essentially allows parallel ALU operations to be performed when the data size is halfword or byte.

Since processing byte-sized pixels or pixel components is common, multiple arithmetic can improve performance by a factor of four for many algorithms.

When multiple arithmetic is specified, the Asize field (bits 2–0) of the status register (sr) determines whether the ALU operates as four 8-bit ALUs (byte-multiple arithmetic), two 16-bit ALUs (halfword-multiple arithmetic), or one 32-bit ALU (word-multiple arithmetic). Special status setting to the multiple flags (mf) register is coupled with split ALU operation.

The function-modifier field of the instruction or d0, in the case of EALU operations, can select one of several options for how the mf register bits are set. Setting mf bits on the basis of carry out for each split of the ALU is specified directly in the FMOD field by the mc, mzc, and mrc options shown in Table 8–18. The MSS field in the status register determines how the mf bits are set for the me, mze, and mre function modifier options. The MSS field determines if a zero-result compare, the sign bit, or the extended sign bit sets an mf bit for each split of the ALU.

The mx function modifier causes split ALU operations to be performed without setting any bits in mf.

The combination of split ALU operations with multiple status setting in the mf register supports extremely powerful operations that are explained in detail in Chapter 15, *Software Applications*.

Table 8–18 shows the possible combinations of multiple-arithmetic modifiers. Modifier **m** indicates that multiple-arithmetic operation is to be performed.

Table 8–18. Multiple Arithmetic Modifiers

Operation	Function Performed
dst = m expression	Multiple arithmetic (m), defaults to dst = mc
dst = mc expression	Multiple arithmetic (m), set mf on carries (c)
dst = me expression	Multiple arithmetic (m), set mf on the basis of the MSS field in the status register
dst = mzc expression	Multiple arithmetic (m), zero mf (z), set mf on carries (c)
dst = mze expression	Multiple arithmetic (m), zero mf (z), set mf on the basis of the MSS field in the status register
dst = mrc expression	Multiple arithmetic (m), rotate mf (r), set mf on carries (c)
dst = mre expression	Multiple arithmetic (m), rotate mf (r), set mf on the basis of the MSS field in the status register
dst = mx expression	Multiple arithmetic (m) with no mf bit setting

Modifier **z** indicates that the mf register should be zeroed, with the exception of the bits being set by multiple-arithmetic status setting. Modifier **r** indicates that the mf register should be rotated left by the amount specified by the type of multiple arithmetic being performed (four bits for byte, two bits for halfword, and one bit for word). Multiple-arithmetic status setting writes over the LSBs of the rotated mf.

Table 8–18 shows that zeroing and rotating of the mf register are mutually exclusive operations.

Modifier **c** indicates that the resulting carry status bit(s) should be placed in the mf register (the default), and modifier **e** indicates that the MSS field of the sr (status) register determines what causes bits in the mf register to be set, as shown in Table 7–4, *MSS Field Values*.

8.3.4.2 The Bit-Detection Modifiers

The PP has special hardware referred to as the bit-detection logic that can detect the first 1 or first bit change in an input data word. The bit detection output becomes the input to the C port of the ALU. The following bit-detection operations can be performed on a D register operand:

- **lmo (Leftmost One)**

  Returns the position of the leftmost (most significant) bit that is a 1. If no bit in the source has a value of 1, the overflow status bit is set, and the bit-detection hardware outputs 32 (0x20) to the C port of the ALU.

- **rmo (Rightmost One)**

  Returns the position of the rightmost (least significant) bit that is a 1. If no bit in the source has a value of 1, the overflow status bit is set, and the bit-detection hardware outputs 32 (0x20) to the C port of the ALU.

- **lmbc (Leftmost Bit Change)**

  Returns the position of the leftmost (most significant) bit that is not the same as the sign bit. If all bits in the source are equal, the overflow status bit is set, and the bit-detection hardware outputs 32 (0x20) to the C port of the ALU.

- **rmbc (Rightmost Bit Change)**

  Returns the position of the rightmost (least significant) bit that is not the same as bit 0. If all bits in the source are equal, the overflow status bit is set, and the bit-detection hardware outputs 32 (0x20) to the C port of the ALU.

The bit-detection operations return the values given in Table 8–19. These values are effectively the 1s complement of the bit number.

When a bit-detection operation is specified, the data routing to the ALU for each of the eight base set ALU operation classes is modified. The input to the barrel rotator is also used as the input to the bit-detection logic. The output of the bit-detection logic is used as the input the to the C port. For bit-detection operations, the input to the barrel rotator and the bit-detection logic must be a D register. The destination for the ALU output must also be a D register. The input to the A port of the ALU can be any register or an immediate.

Table 8–19. The lmo, rmo, lmbc, and rmbc Operation Results

bit position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
result	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

With the above modifications to the base set ALU operation classes, only two unique data routings are supported for bit-detection operations. These data routings are summarized in Table 8–20.

Table 8–20. Modified Base Set ALU Operation Classes for Bit-Detection Operations

Class	Dest	A Port Input	B Port Input	C Port Input
0	dst	src2	src1	lmo(src1)
5	dst	src2	src1\\d0	lmo(src1)

Notes: 1) dst must be a D register
2) src1 must be a D register; used as the input to both the barrel rotator and the bit detection logic
3) src2 can be a D register or an immediate
4) lmo can be replaced by rmo, lmbc, or rmbc

Bit-detection operations can be specified with any of the class-independent arithmetic functions in the base set. However, since the same input is fed to the barrel rotator and the bit-detection logic, functions that involve the B port are not very useful. Base set Booleans cannot be used with the bit-detection logic, because the instruction word does not allow function modifiers to be specified. For EALU operations, bit-detection operations can be specified for both arithmetic and Boolean ALU functions that use the C port. Other function modifiers (multiple arithmetic, cin, A port = 0, and %!) cannot be specified in the same operation as bit-detection operations.

The most useful supported functions with bit detection are A + C and A – C. These functions allow the ALU to adjust the output of the bit-detection logic for the desired application. In Example 8–13, the leftmost-one output is adjusted by the ALU so that it indicates the number of bits required to represent the unsigned 32-bit input.

Example 8–13. Determine Size of Magnitude Using lmo

```
;; d7 = 0x0000000F (15 decimal)
;; d6 = 0x00000081 (129 decimal)

   d5 = 32 – lmo(d7)
   d4 = 32 – lmo(d6)

;; Results:
;; d5 = 0x00000004    lmo(d7) = 28
;; d4 = 0x00000008    lmo(d6) = 24
```

When a bit-detection operation is performed, the ALU is split at eight bits. Thus, if the ALU is used to adjust the output of the bit-detection logic, the ALU result may be correct only through the first eight bits because (as illustrated in Example 8–14) the carry-out from bit 7 is not fed in as the carry-in to bit 8. If necessary, the eight LSBs of the result (which are correct) can be sign-extended via a field extract move, as shown in Example 8–14.

Example 8–14. Split Carry Path for Bit-Detection Operation

```
;; Before:
;;    d6 =  0xFFFFFF00

   d6 = 4 – lmbc(d6)  ;  Subtract leftmost-bit-change
                      ;  result (in this case 24) from 4.

;; After:
;;    d6 = 0x000000EC

   d6 =sb0  d6     ;  Sign extend using Field Extract
                   ;  Move.

;; After:
;;    d6 = 0xFFFFFFEC (–20 decimal)
```

Often the output from the bit detection and ALU operation is used only for a 5-bit shift or mask amount. 8- or 32-bit accuracy is not required. In this case, a 5-bit immediate and an add or subtract can give any variation of src2 +/– lmo (src1) to a 5-bit accuracy.

Status setting for bit-detection operations is summarized as follows:

- **V**

  Set to 1 if no bit equal to 1 is found for lmo or rmo operations (if all bits are 0); set to 1 if no bit change is detected for lmbc or rmbc (all bits are the same).

- **N**

  Same as for multiple arithmetic. Set to the AND of the zero compares for each of the split ALU results. If N = 1, it indicates that all of the split ALU results are 0.

- **C**

  Same as for multiple arithmetic. Set to the OR of the carries for the split ALU results. If C = 1, it indicates that at least one of the split ALUs has a carry-out.

- **Z**

  Same as for multiple arithmetic. Set to the OR of the zero compares for each of the split ALU results. If Z = 1, it indicates that at least one of the split ALU results was equal to 0.

Note that no status saving to the mf register is performed.

When comparing a bit-detection result, it is important to note the behavior of the bit-detection split ALU status setting described above. In Example 8–15, the output of the leftmost-one detection is compared with 15. If the output of the bit-detection logic is not 15 (indicating that the leftmost-one in the source register is not little-endian bit 16), then a clean-up subroutine is called.

To perform an operation conditionally if the bit-detection output does not match a specified number, the correct status condition (as shown in Example 8–15) is **nn** (the AND of the zero compares of the split ALU results does not equal 1). If the condition code used is **nz**, then the operation occurs conditionally on the basis of the OR of the zero compares for each of the split ALU results. As in Example 8–15, this typically does not produce the desired effect, because the three upper split ALU results are frequently zero.

Example 8–15. Compare Result of Bit Detection

```
;; d6 = 0x12345678
     d7 = 15 – lmo(d6)
;; d7 = 0x0000000C
;;   status register N bit       =    0    ;    Not all split ALU
;;                                         ;    result are zero
;;   status register Z bit       =    1    ;    The OR of the split ALU
;;                                         ;    zero compares is 1.
     call =[nn] clean_up                   ;    Call clean_up subroutine if
                                           ;    leftmost-one is not bit 16
                                           ;    (big-endian bit 15).
          .
          .
          .
          .
clean_up:
     <Instruction>
                .
```

8.3.4.3 The A Port = 0 Modifier

The A port = 0 function modifier forces an input of 0x0 to the A port of the three-input ALU. This function modifier can be used with any of the base set arithmetics, described in subsection 8.3.2.2. For example, when the A port = 0 modifier is applied to the ALU function A – |B|, the resulting function is –|B|. Some other functions that use this modifier are shown in Table 8–21. These functions act exactly like the unmodified function with a 0x0 immediate input to the A port.

The advantage of using A port = 0 over using an immediate is that it is allowed with any operation class, even those that do not allow the input to the A port to be an immediate (see Table 8–13). For operation classes 1, 2, 3, and 6 that don't associate the immediate source operand with the A port, an immediate data source operand can be used in addition to the 0x0 input to the A port specified by the function modifier.

The A port = 0 function modifier can be specified in parallel with **cin** (carry-in = sr⟦C⟧) and with the mask generator operator modifiers (%!). It cannot be used with multiple arithmetic or bit detection.

Table 8–21. Some Useful Arithmetic Functions That Use the A Port = 0 Function Modifier

Algebraic Description	Description
–B	Negate B port input.
–C	Negate C port input.
|B|	Absolute value of B.
–|B|	Negate absolute value of B.
b >> n	Signed shift right the input to barrel rotator by n (either an immediate or the 2s complement of the contents in a D register—for example, –d0).
b >> u n	Unsigned shift right the input to barrel rotator by n (either an immediate or the 2s complement of the contents in a D register—for example, –d0)

8.3.4.4 The Carry-In Modifier

Using a function modifier, the carry-in to the LSB of the ALU can be input from the status register's carry status bit (typically the carry-out from the previous ALU operation). This allows operations such as add-with-carry and subtract-with-borrow to be specified.

Example 8–16 and Example 8–17 use the carry-in function modifier to perform a 64-bit add and 64-bit subtract, respectively. cin is a keyword reserved by the assembler to indicate the value of the C status bit in sr. Note that for subtracts with borrow, the 1s complement of the C bit (the borrow) is subtracted from the result of the upper word subtract.

Example 8–16. 64-Bit Add

```
d4 = d0 + d2          ; Add lower words.
d5 = d1 + d3 + cin    ; Add upper words with carry-in
                      ; from lower word add
```

Example 8–17. 64-Bit Subtract

```
d4 = d0 – d2          ; Subtract lower words.
d5 = d1 – d3 – ~cin   ; Subtract upper words with
                      ; borrow (~cin) from lower
                      ; word subtract.
```

The **cin** function modifier can be specified in parallel with A port = 0 and/or %!. cin cannot be used with multiple arithmetic or bit detection. It also cannot be used with absolute value or signed shift right operations.

8.3.4.5 The Right-Shift-Mask Modifier

The right-shift-mask function modifier (%!) causes an input of 0 to the mask generator to output a mask of 32 1s instead of all 0s. This function modifier is required for right shift operations where the shift amount may be zero. The right shift mask modifier can also be specified in parallel with A port = 0 and **cin**. It cannot be used with multiple arithmetic or a bit-detection operation.

Note that the right-shift-mask function modifier behaves exactly like the mask generator operator (%) given a nonzero input.

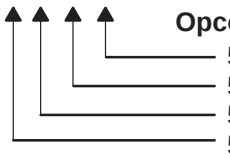
8.3.5 Summary of ALU Function Modifier Codes

Table 8–22 shows the ALU function modifier codes. Some of the codes take on different meanings, depending on the operation class. For example, code 0010 specifies the right-shift-mask modifier (%!) if the operation uses the mask generator and lmo if the operation does not use the mask generator.

In Table 8–22, note that neither multiple arithmetic nor bit detection can be specified in conjunction with any of the other ALU function modifiers. A function modifier code is available for each combination of the zero input to the A port = 0, cin, and right-shift-mask (%!) modifiers.

Table 8–22. ALU Function Modifier Codes

Function Modifier Code	Modification Performed
0 0 0 0	Normal operation
0 0 0 1	cin
0 0 1 0	%! if ALU operation uses mask generator; otherwise, lmo
0 0 1 1	%! and cin if ALU operation uses mask generator; otherwise, rmo
0 1 0 0	A port = 0
0 1 0 1	A port = 0 and cin
0 1 1 0	A port = 0 and %! if ALU operation uses mask generator; otherwise, lmbc
0 1 1 1	A port = 0, %!, and cin if ALU operation uses mask generator; otherwise, rmbc
1 0 0 0	mc: multiple arithmetic, mf bit(s) set by carry-out(s)
1 0 0 1	me: multiple arithmetic, mf bit(s) set on the basis of the MSS field in the status register.
1 0 1 0	mrc: multiple arithmetic, rotate mf, mf bit(s) set by carry-out(s).
1 0 1 1	mre: multiple arithmetic, rotate mf, mf bit(s) set on the basis of the MSS field in the status register.
1 1 0 0	mzc: multiple arithmetic, zero mf, mf bit(s) set by carry-out(s).
1 1 0 1	mze: multiple arithmetic, zero mf, mf bit(s) set on the basis of the MSS field in the status register.
1 1 1 0	mx: multiple arithmetic with no mf bits set.
1 1 1 1	Reserved.



Opcode bit	EALU d0 bit
52	28
54	29
56	30
58	31

Note: **cin** is a carry-in to ALU from the carry bit in the status register, **sr**⟦**C**⟧.
%! is the right-shift-mask modifier (%!0 = 0xFFFFFFFF).
A port = 0 forces a zero input to the A port.

8.4 The Extended Set of ALU Operations (EALU)

Extended ALU (EALU) operations are defined by the contents of register d0. The opcode contains the register operands for the operation and specifies that the EALU operation defined in the d0 register is to be performed.

You can perform EALU operations in parallel with a multiply, either signed or unsigned (referred to as MPY||EALU), or with a parallel assignment of the output of the barrel rotator to a destination register (referred to as EALU||ROTATE).

These are the major reasons for using an EALU operation:

- ❑ EALU operations support numerous arithmetic ALU functions beyond the base set arithmetics.
- ❑ MPY||EALU supports a multiply in parallel with any of the 256 Boolean or 256 arithmetic functions supported by the ALU.
- ❑ MPY||EALU supports two 8×8 multiplies in a single cycle (referred to as split multiply).
- ❑ MPY||EALU supports a special rounded multiply for retaining precision when only the upper halfword of a 32-bit product is retained.
- ❑ EALU||ROTATE (an EALU operation without a parallel multiply) supports six ALU operands; base set ALU operations support only three ALU operands. This flexibility of EALU||ROTATE allows you to perform an arbitrary field extract.
- ❑ EALU||ROTATE (an EALU operation without a parallel multiply) can optionally save the barrel rotator output to a destination D register.
- ❑ EALU operations allow ALU function modifiers to be used with Boolean ALU operations.
- ❑ EALU operations allow the same piece of code to perform different functions according to the value in the d0 register.

Chapter 13, *Advanced EALU*, contains details on the bits in the d0 register that specify the EALU operation. The assembler can take an expression for the desired operation and determine the required 32-bit constant to configure that operation. Thus, a detailed knowledge of the bits in d0 is not required to use EALU operations.

Table 8–23 tells where to find the topics discussed in this section.

Table 8–23. Discussion of the Extended Set of ALU Operations

Section	Topic
8.4.1	Using the EALU Data Path
8.4.2	Understanding the EALU Syntax
8.4.3	Using the Generic EALU Syntax
8.4.4	Using the Explicit EALU Syntax
8.4.5	Using Explicit EALU Expressions
8.4.6	Using EALU Operations Without a Parallel Multiply

8.4.1 Using the EALU Data Path

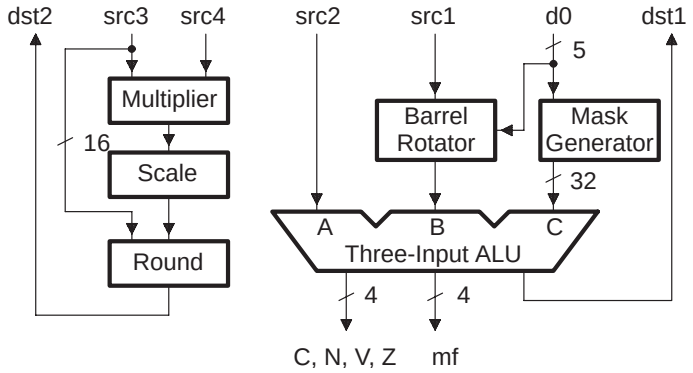
The supported EALU data paths are shown in Figure 8–9 and summarized in Table 8–24. The only difference between the extended ALU operations supported by MPY||EALU and EALU||ROTATE is that the ALU data path for MPY||EALU is less flexible in that the barrel rotate amount and mask generator input are always the d0 register for MPY||EALU. The multiplier data path features supported in parallel with an EALU are described in subsection 8.5.2.

Table 8–24. Data Path for EALU Operations

MPY||EALU			EALU||Rotate		
A	B	C	A	B	C
src2	src1\\d0	%d0 or @mf	src2 src2	src1\\src3 src1\\src3	src4 %src4 or @mf

Figure 8–9. EALU Data Path Options

(a) MPY||EALU



(b) MPY||EALU with @mf

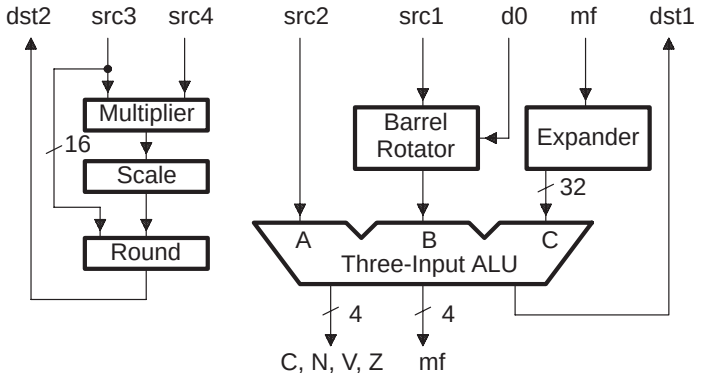
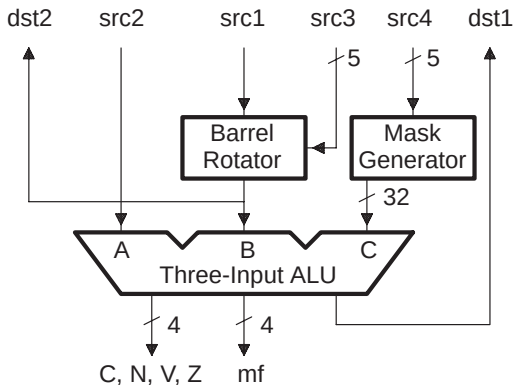
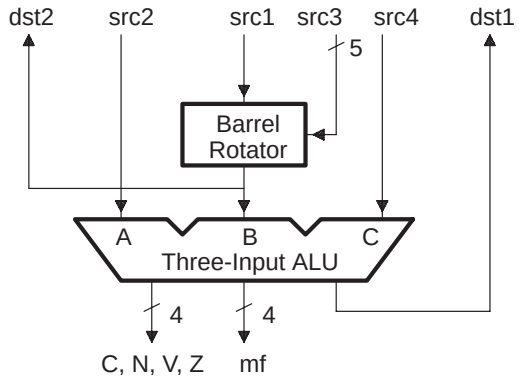


Figure 8–9. EALU Data Path Options (Continued)

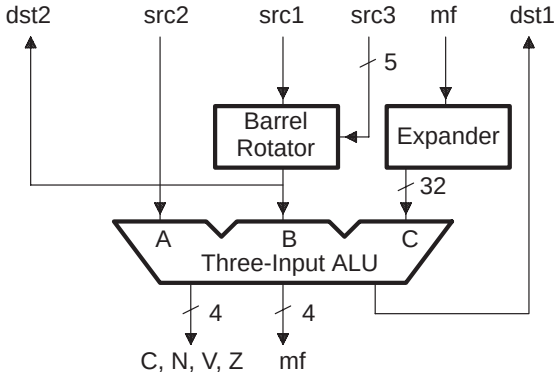
(c) EALU||ROTATE (1)



(d) EALU||ROTATE (2)



(e) EALU with @mf||ROTATE



The operators supported by EALU operations are the same as those for the base set ALU except that a nonmultiple mask operator (%% or %%!) is supported. When a nonmultiple mask is specified, a single (unsplit) mask is produced, even if multiple arithmetic is performed by the three-input ALU.

For more information on the ALU operators, refer to subsection 8.2.1, *Operator Precedence*.

8.4.2 Understanding the EALU Syntax

An EALU operation applies the operation specified in d0 to the arguments specified in the opcode. The opcode also specifies the data routing of these arguments into the ALU.

The PP assembler has two types of syntax for an EALU operation:

- **Generic syntax**

  The generic syntax specifies only the information contained in the opcode (the explicit inputs to the A, B, and C ports of the ALU). This syntax is provided to support subroutines that receive the d0 register as an input argument and thus do not have a unique operation that is performed by that EALU.

  Refer to subsection 8.4.3, *Using the Generic EALU Syntax*, for more information.

- **Explicit syntax**

  The explicit syntax specifies a label followed by an explicit expression. The explicit expression specifies what operation is to be associated with the label, and it specifies the A, B, and C port inputs within the context of the expression.

  The assembler computes a 32-bit constant that supports the explicit operation and assigns that value to the label. Thus, you can configure the d0 register for the desired explicit expression by setting d0 to the label's value.

  Note that when the resulting EALU instruction is executed, it will perform the explicit expression used in the source code only if the d0 register contains the value of the label.

  Refer to subsection 8.4.4, *Using the Explicit EALU Syntax*, for more information.

8.4.3 Using the Generic EALU Syntax

You can use EALU operations with a generic routine that has the EALU function passed to it by another routine. To do this, you must specify the instruction using a generic syntax.

The generic syntax specifies only the information that is actually contained in the instruction; it does not specify any information that is contained in the d0 register.

There are two types of generic EALU syntax (note that the ⟦ ⟧ symbols represent required square brackets that must be used with the conditional operand):

- **MPY||EALU**

  Using MPY||EALU, you can have an EALU operation with a parallel multiply. The syntax for a generic MPY||EALU operation is:

  ```
  dst2 = [sign] [ ⟦cond⟧ ] src3 * src4
    || dst1 = [ ⟦cond [.pro] ⟧ ] ealu[f] (src2, src1[ ⟦n⟧src1–1]\\d0, %d0)
  ```

  or

  ```
  dst2 = [sign] [ ⟦cond⟧ ] src3 * src4
    || ealu()
  ```

- **EALU||ROTATE**

  Using EALU||ROTATE, you can assign the rotator output to a register and specify the rotate amount and C port input from different registers. The general syntax for a generic EALU||ROTATE operation is:

```
dst1 = [ ⟦cond [.pro] ⟧ ] ealu[f] (src2, [dst2 =] [⟦cond⟧] src1[ ⟦n⟧src1–1]\\src3, [%] src4)
```

Note that when %d0 or %src4 is used with a d0 value that specifies @mf, the mask generator output is replaced with the @mf output as the input to the C port of the ALU.

The following bullets briefly describe the parameters in the MPY ||EALU and EALU||ROTATE syntax.

- dst2

  The dst2 operand represents the multiply destination or barrel rotator output destination. It must be a D register.

- dst1

  The dst1 operand represents the ALU destination. It can be a non-D register only if a global transfer is not specified in parallel; otherwise, it must be a D register.

- sign

  The sign operand indicates whether a multiply is signed or unsigned.

  - Signed multiply

    An **s** indicates signed multiply and is the default. For a 16 × 16 multiply, you must have a signed src3 operand multiplied by a signed src4 operand.

    For a split multiply, you must have a signed src3 operand multiplied by an unsigned src4 operand.

  - Unsigned multiply

    A **u** indicates unsigned multiply. For both a 16 × 16 multiply and a split multiply, the src3 and src4 operands must be unsigned.

- ⟦cond⟧

  The ⟦cond⟧ operand represents conditional assignment of both the multiply operation and EALU results. If the multiply or barrel-rotate assignment is conditional, the EALU assignment must also be conditional; the reverse is true also. The condition code can be any of the 16 supported conditions (see Table 8–36).

  Note that the ⟦ ⟧ symbols represent required square brackets that must be used with the conditional operand.

- src1

  The src1 operand is the input to the barrel rotator, and the output of the barrel rotator goes to the B port of the ALU.

- src2

  The src2 operand is the D-register input to the A port of the ALU.

- src3

  The src3 operand represents the first multiply source for MPY||EALU. For EALU||ROTATE, src3 specifies the barrel rotator amount. It must be a D register.

- [%]src4

  The src4 operand represents the second multiply source for MPY||EALU. For EALU||ROTATE, src4 specifies the input to the C port of the ALU or the mask generator input for the C port of the ALU. It must be a D register.

  If %src4 is specified and the F bit is set in the d0 register, %src4 is replaced with @mf as the input to the C port of the ALU.

- f—EALU false

  The MPY||EALU and EALU||ROTATE (only in the case where **% src4** is input to the ALU's C port) have the option in the instruction of bitwise inverting the control bits (specified in d0 bits 27–20) that are sent to the ALU.

  For arithmetic options (where this capability is most useful) inverting the ALU control bits has the net effect of changing a specified addition to subtraction or subtraction to addition. The ALU carry control logic will also invert the carry- in for the false option in order to be consistent with adding or subtracting. Table 8–25 shows a few examples of the EALU and EALUf combinations.

Table 8–25. EALU and EALUf Combinations

EALU Operation Specified in d0	EALUf Operation
src2 + src1	src2 – src1
src2 – src1	src2 + src1
src2 + src1 >> d0	src2 – src1 >> d0
src2 – src1 << src3	src2 + src1 << src3
(src2 & src4) + src1 \\ src3	(src2 & src4) – src1 \\ src3
src2 + (src1 \\ d0) & @mf	src2 – ((src1 \\ d0) & @mf)

The EALUf option allows for dynamically changing the sign of the EALU operation without having to modify the d0 register, since the **f** option is specified in the instruction. The need to use complimentary pairs (differing only in the sign of the operation) occurs commonly in signal processing, particularly Fast Fourier transform (FFT) and discrete cosine transform (DCT) butterflies.

The generic assembler support for EALUs allows a complementary pair of EALU operations, which differ only in whether an add or subtract is to be performed, to be specified with a single d0 value.

8.4.4 Using the Explicit EALU Syntax

The PP assembler has a special feature that makes it possible to specify EALU operations using an explicit expression for the desired operation. As shown in Example 8–18, a label is specified with an expression for the desired operation. The assembler determines a 32-bit value which, if in the d0 register, would configure the EALU to perform this operation and associates that value with the label. That label can then be used to initialize the d0 register before executing the EALU instruction.

Example 8–18. Explicit EALU Syntax

```
d0  = elabel                        ; setup
      .
      .
d7  = ealu(elabel:expression)       ; actual EALU operation
```

There are two types of explicit EALU syntax:

- **MPY||EALU**

  Using MPY||EALU, you can have an EALU operation with a parallel multiply. The general syntax for an explicit MPY ||EALU operation is:

  dst2 = [sign] [opt] [⟦cond⟧] src3 * src4 [<< dms]
  || dst1 = [fmod] [⟦cond [.pro] ⟧] ealu(label: EALU_EXPRESSION)

  If you only want the multiply operation, then the syntax is the MPY operation portion with **ealu(label)** in parallel:

  dst2 = [sign] [opt] [⟦cond⟧] src3 * src4 [<< dms] || ealu(label)

  If << dms is omitted, then dms is set to zero by default.

- **EALU||ROTATE**

  EALU||ROTATE allows you to assign the rotator output to a D register with an EALU operation and specify the rotator amount and C port input individually. The general syntax for an explicit EALU||ROTATE operation is:

  dst1 = [fmod] [⟦cond [.pro] ⟧] ealu(label: EALU_EXPRESSION)
  [|| dst2 = ⟦cond⟧[(]src1[⟦n⟧src1–1)]\\src3]

The following bullets briefly describe the operands in the MPY ‖EALU and EALU‖ROTATE syntax.

- dst2

  The dst2 operand represents the multiply destination or barrel rotator output destination. It must be a D register.

- sign

  The sign operand indicates whether a multiply is signed or unsigned. See subsection 8.4.3, *Using the Generic EALU Syntax*, for more information.

- opt

  The opt operand specifies one of two optional multiply features:

  - **Split multiply**. Performs two 8-bit by 8-bit multiplies producing 16-bit products. A lowercase **m** for the opt operand specifies a split multiply.
  - **Rounded multiply**. Rounds the 32 bit output of the multiply shifter by adding bit 15 to bit 16 (with carry-propagation). Also, replaces the 16 LSBs of the output from the multiply scaler with the 16 MSBs of the src3 operand. A lowercase **r** for the opt operand specifies a rounded multiply.

- ⟦cond⟧

  The ⟦cond⟧ operand represents conditional assignment of both the multiply operation or rotate operation and ALU results. If the multiply or barrel-rotate assignment is conditional, the ALU assignment must also be conditional; the reverse is true also. The condition code can be any of the 16 supported conditions (see Table 8–36).

  Note that the ⟦ symbol represents required square brackets that must be used with the conditional operand.

- src3

  The src3 operand represents the first multiply source for MPY||EALU. For EALU||ROTATE, src3 specifies the barrel rotator amount. It must be a D register.

- src4

  The src4 operand represents the second multiply source for MPY||EALU. For EALU||ROTATE, src4 specifies the input to the C port of the ALU, and is optionally passed through the mask generator. It must be a D register.

- << dms

  The dms operand represents the default multiply shift (0,1,2, or 3), and is the number of bit positions by which the output of the multiplier is shifted left (with 0's filling the vacated LSBs). By default, dms is 0.

- dst1

  The dst1 operand represents the ALU destination. It can be a non-D register only if a global transfer is not specified in parallel; otherwise, it must be a D register.

- fmod

  The fmod operand indicates multiple arithmetic function modifiers (see subsection 8.3.4.1, *The Multiple Arithmetic Modifiers*):

  - **mc** represents multiple arithmetic, where the bits in the mf register are set by carry-outs.
  - **me** represents multiple arithmetic, where the bits in the mf register are set on the basis of the MSS field in the status register.
  - **mzc** represents multiple arithmetic, where the bits in the mf register are set by carry-outs and the mf register is zeroed.
  - **mze** represents multiple arithmetic, where the bits in the mf register are set on the basis of the MSS field in the status register, and the mf register is zeroed.
  - **mrc** represents multiple arithmetic where the bits in the mf register are set by carry-outs, and the mf register is rotated.
  - **mre** represents multiple arithmetic where the bits in the mf register are set on the basis of the MSS field in the status register, and the mf register is rotated.

- elabel

  The elabel operand represents the symbol to which the assembler assigns the 32-bit constant that would produce the associated expression if the d0 register contained that constant.

- expression

  The expression operand represents a register-specific expression (using the same syntax as the base set ALU operations) for the EALU operation that is intended. The supported expressions are described in subsection 8.4.5, *Using Explicit EALU Expressions*.

Example 8–19 uses the explicit syntax to perform a simple shift and add and also a shift and add in parallel with a split multiply. As shown in this example, the same label can be shared by more than one EALU. The assembler attempts to create one 32-bit constant that supports all of the operations associated with the label. If different operations are specified with the same label that can't be implemented with the same d0 register value, the assembler returns an error.

In Example 8–19, some shift amounts are specified with a constant. When the assembler sees a constant used as the barrel rotate amount, a shift amount, or the input to the mask generator, it assumes that this value is supposed to be configured in the default barrel rotate amount in the d0 register, since immediate operands cannot be specified in the instruction opcode for EALUs.

Example 8–19 also shows a subtract and shift using the same EALU label as the add and shift operations. This is equivalent to the ealuf operations in the generic EALU syntax. The assembler detects that the ALU code in d0 for the subtract is the inverse of the ALU code for the add. It will then use the ALU value for the add, but choose the opcode for ealuf, which inverts the ALU code in d0 before sending it to the ALU. Inverting the ALU code in d0 is available only when the mask generator output or @mf goes to the C port of the ALU. This means that you can invert the ALU code in the following two cases:

- With an EALU||ROTATE when %src4 or @mf goes to the C port (inverting is not available when the C port input comes directly from a register).

- With an MPY||EALU, which has %d0 or @mf going to the C port of the ALU.

Example 8–19. Shift and Add Using Explicit EALU Syntax

```
d0  =      SHIFT_ADD
    .
    .
d7  =      ealu(SHIFT_ADD: d6 + d5>>7)
d7  =      ealu(SHIFT_ADD: d1 + d2>>–d3)
d4  =      d5 * d6
    ||d1 = ealu(SHIFT_ADD: d2 + d3>>7)
d6  =      ealu(SHIFT_ADD: d2 – d3>>7)
```

Note that while shifting by a constant implies shifting by the d0 register, with the same EALU label, shifting by other registers is also allowed—as that is specified in the opcode rather than in the d0 register.

8.4.5 Using Explicit EALU Expressions

A valid expression within an explicit EALU operation specifies a function supported by the three-input ALU using registers combined by the PP ALU operators. The set of legal EALU operations is a combination of the ALU functions supported by the ALU, the data path routings described in Table 8–24, *Data Path for EALU Operations*, the ALU function modifiers, and other bits in the d0 register that specify the carry-in to the ALU.

8.4.5.1 Boolean Functions

EALU operations support the same 256 Boolean ALU functions supported by base set ALUs but add more flexibility in the operands. The Boolean function performed by the ALU is a sum of products controlled by the ALU function code (see Table 8–26). Note that a 1 can be added (with carry-ripple) to the result of the Boolean sum. Alternately, the carry-in can be taken from the carry status bit in the status register.

Example 8–20 shows an example of a Boolean EALU operation specified with the explicit syntax. A function code that performs the specified operation can be determined by using a Karnaugh map, as shown in Figure 8–10. Note that it is not necessary for you to determine the function code because the assembler does this automatically for the explicit EALU expressions.

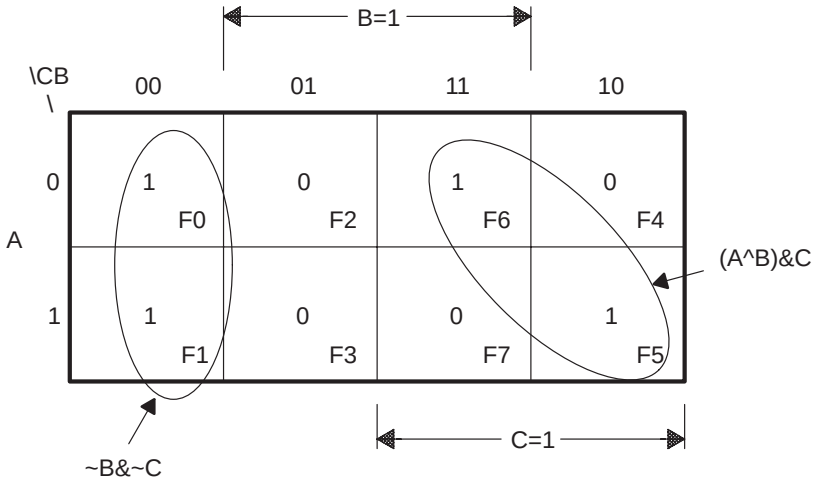
The assembler forms the 32-bit constant required in the d0 register to perform the specified operation. This constant is associated with the label for the explicit EALU operation (for example, MERGE in Example 8–20). The d0 register can then be set to the desired value by this label.

Note that the value of the label merge does not come from an include file or any predefined constant. The value is determined from the EALU expressions in the ealu instructions it is used in.

Example 8–20. Boolean Explicit EALU

```
d0 = MERGE
.
.
.
d7 = ealu (MERGE: ((d7 ^ d6\\d3) & % d1)|(~d7 & ~ %d1))

;; Can be performed using (A^B)&C | ~B&~C
;;
;; Function code can be determined to be 0110 0011 (0x63)
;; using a Karnaugh map as shown in Figure 8–10.
```

Figure 8–10. Karnaugh Map



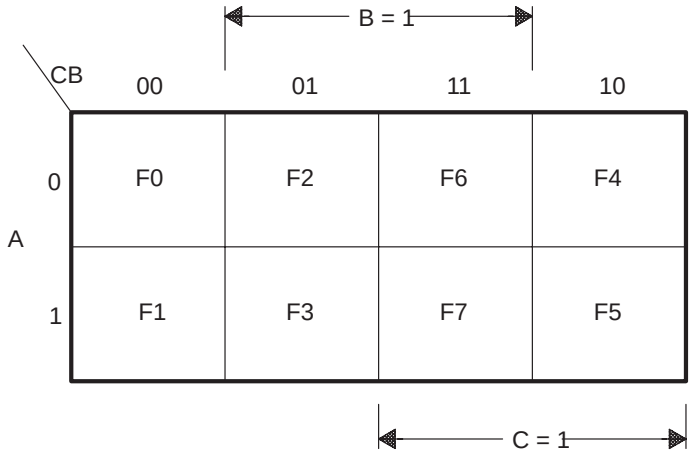
The Boolean function performed by the ALU is:

(F0&(~A&~B~&~C) | F1&(A&~B&~C) | F2&(~A&B&~C) | F3&(A&B&~C) | F4&(~A&~B~&C) | F5&(A&~B&C) | F6&(~A&B&C) | F7&(A&B&C)) [+1|+cin]

Table 8–26. ALU Function Code Signals

d0 Bit	Opcode Bit	ALU Function Signal	Products
26	58	F7	A & B & C
25	57	F6	~A & B & C
24	56	F5	A & ~B & C
23	55	F4	~A & ~B & C
22	54	F3	A & B & ~C
21	53	F2	~A & B & ~C
20	52	F1	A & ~B & ~C
19	51	F0	~A & ~B & ~C

Figure 8–11. ALU Function Signal and Karnaugh Map



8.4.5.2 Arithmetic Functions

The EALU operations support all of the 256 arithmetic functions provided by the three-input ALU. Also, carry-in is controlled in the d0 register; therefore, a 1 can be added to the result of the ALU function (for both Booleans and arithmetics) by forcing a carry-in. Alternately, the carry-in can be taken from the carry status bit in the sr register (specified by +cin). The arithmetic function performed by the ALU is as follows:

arithmetic function = A&f1(B,C) + f2(B,C) [+1 | +cin]

f1(B,C) and f2(B,C) are independent Boolean combinations of B and C, as shown in Table 8–27. The ALU function code can be determined by: f1 XOR f2.

Table 8–27. Possible f1(B,C) or f2(B,C) Functions

f1 Code	f2 Code	Subfunction	Common Use
00	00	0	Zeros one of the terms
AA	FF	all 1s = –1	All 1s or –1
88	CC	B	B
22	33	–B–1	Negate B
A0	F0	C	C
0A	0F	–C–1	Negate C
80	C0	B&C	Force bits in B to 0 where C is 0
2A	3F	–(B&C)–1	Force bits in B to 0 where C is 0 and negate
A8	FC	B|C	Force bits in B to 1 where C is 1
02	03	–(B|C)–1	Force bits in B to 1 where C is 1 and negate
08	0C	B&~C	Force bits in B to 0 where C is 1
A2	F3	–(B&~C)–1	Force bits in B to 0 where C is 1 and negate
8A	CF	B|~C	Force bits in B to 1 where C is 0
20	30	–(B|~C)–1	Force bits in B to 1 where C is 0 and negate
28	3C	(B&~C)|((–B–1)&C)	Choose B if C = all 0s and –B if C = all 1s
82	C3	(B&C)|((–B–1)&~C)	Choose B if C = all 1s and –B if C = all 0s

Special operations such as absolute value and signed shifts can be performed by using bits in the d0 register that control the ALU function on the basis of the sign of one of the inputs.

Example 8–21 shows an example of an arithmetic EALU operation specified using the explicit syntax. As shown in Example 8–21, it possible to determine a function code using Table 8–27 that supports a particular arithmetic operation.

Example 8–21. Arithmetic EALU Specified by Explicit Syntax

```
        d0 =  ARITH
        .
        .
        d7 =  ealu(ARITH: d1&(d2\\d3)&~d4 – d4)
;;      Can be performed with:
;;      A = d1
;;      B = d2\\d3
;;      C = ~d4
;;
;;       Where the ALU function determined by the assembler
;;       is:
;;       A&(B&~C) – C
;;
;;      f1(B,C) = B&~C                f1 code = 0x08
;;      f2(B,C) = C                   f2 code = 0xF0
;;
;;      Function Code for ARITH = 0x08 ^ 0xF0 = 0xF8
```

8.4.5.3 Special EALU Operations

Some EALU operations perform different ALU functions depending on whether the input to the barrel rotator or ALU is positive or negative. These operations include:

- ❑ Absolute value
- ❑ Signed shift right
- ❑ Conditional add or subtract of the input to the B port for each split ALU segment.

The assembler recognizes these operations and can evaluate the necessary 32-bit constant for d0. This includes setting the I, S, and E bits in the d0 register that controls the ALU function on the basis of sign information. Details on these bits are provided in Section 13.2, *EALU Carry-In and Sign Extension.*

EALU operations support taking the **absolute value** of B and optionally adding it to or subtracting it from A. For example:

- ❑ [A +] |B|
- ❑ [A] – |B|

Example 8–22. Multiply With Parallel Absolute Value

```
d0      =       SUB_ABS
        .
        .
d7      =u d6 * d5
        ||d1 = ealu(SUB_ABS: d2 – |d3|)
```

EALU operations support **shift operations**. The shifted result can be added to or subtracted from A. For example:

- [A +] b << r
- [A] – b << r
- [A +] b >> r
- [A +] b >>u r
- [A] – b >> r
- [A] – b >>u r

Example 8–23. Multiply With Parallel Shift Left

```
d0     =      SHIFT_LEFT
       .
       .
d7     =      d6 * d1
       ||d2 = ealu(SHIFT_LEFT: d4<<9)
```

The EALU supports **conditional add or subtract** depending on the input to the C port. These EALU functions assume that there is a uniform mask input to the C port for each split ALU segment. The uniform mask determines each ALU segment, whether B or –B is added to A.

- A + (B&C | –B&~C)
- A + (–B&C | B&~C)

Example 8–24. Add or Subtract B, As Determined by Input to C

```
d0  = ADD_SUB
    .
    .
d7  = ealu(ADD_SUB:d1+(d2\\d0&d3|-(d2\\d0)&~d3))
```

8.4.5.4 Function Modifiers

EALU operations support the same set of function modifiers as the base set. Unlike the base set, function modifiers can be applied to both arithmetic and Boolean ALU functions. The ALU function modifiers are shown in Table 8–22, *ALU Function Modifier Codes*.

- **Normal**

  This coding does not modify the natural ALU operation.

- **cin**

  If this option is specified (by the keyword cin), the carry-in to bit 0 of the ALU is taken directly from the C bit of the status register. This overrides the effect of all the other carry-related bits (C, I, S and E) in the d0 register (for details, see subsection 13.2.6, *Summary of Carry-In Determination*).

- **Right-Shift-Mask**

  A right-shift mask is generated in the same manner as the mask generator modifier (%) with the exception of an input of 0, in which case, the generated mask is all 1s instead of all 0s. This option is mutually exclusive with bit-detection operations (lmo, rmo, lmbc, rmbc).

  This modifier can also be applied to a non-multiple mask (for example, %%!). Note that a right-shift mask (%!) is required for right shifts (for example, d4 >> –d1).

- **Bit Detection**

  A bit-detection operation is specified as a function applied to a source operand; the result becomes the input to the C port of the ALU. Note that the data path of the source operand is modified when bit detection is specified.

  - **lmbc()**

    Left-most bit change.

  - **rmbc()**

    Right-most bit change.

  - **lmo()**

    Left-most one.

  - **rmo()**

    Right-most one.

❑ A port=0

If this option is specified, the ALU's function code is changed (by the hardware) so that the input to the A port of the ALU is treated as if it were zero. This function modifier is not as important for EALUs because you can zero the A-port by selecting an ALU function that ignores the A port input.

❑ Multiple Arithmetic

As in the base set, multiple arithmetic is specified by modifiers attached to the assignment operator (for example, =mc). If one of the multiple arithmetic codes is selected, the ALU carry path is split according to the Asize field in the status register. You can specify the following options for modifying the mf register:

- **mc** and **me**

  The LSBs of mf record either the individual carry-outs of each piece of the ALU (mc), or according to the condition specified in the MSS field of the status register (me). The number of bits set is the same as the number of pieces into which the ALU is split.

- **mrc** and **mre**

  These are the same as mc and me, except that the mf register is pre-rotated left before the bits are set.

- **mzc** and **mze**

  These are the same as mc and me, except that the mf bits that are not being set are zeroed.

- **mx**

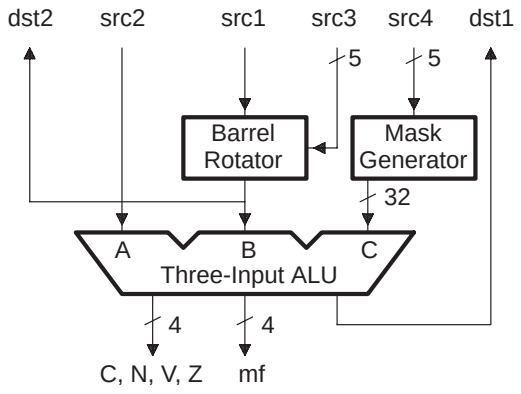
  The ALU carry path is split for the ALU operation, but no status bits are set in the mf register.

8.4.6 Using EALU Operations Without a Parallel Multiply

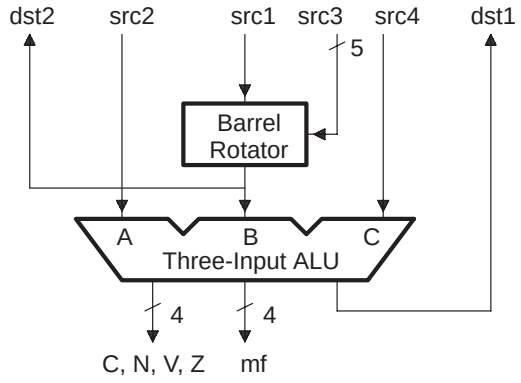
When an EALU operation is specified without a parallel multiply, both the output of the barrel rotator and the output of the ALU can be written to destination registers specified in the instruction. There are two supported data routings for EALU||ROTATE: one supplies the input to the C port directly; the other routes src4 through the mask generator or sends @mf to the C port.

Figure 8–12. EALU||ROTATE Data Routings

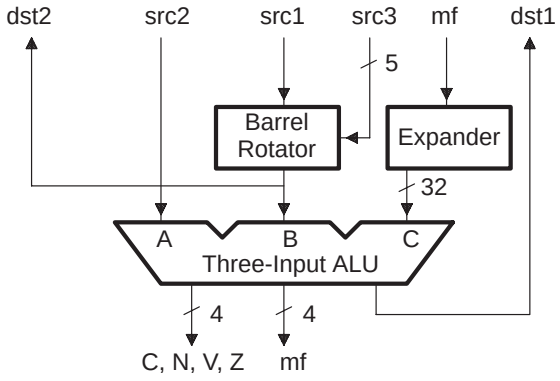
(a) EALU||ROTATE (1)



(b) EALU||ROTATE (2)



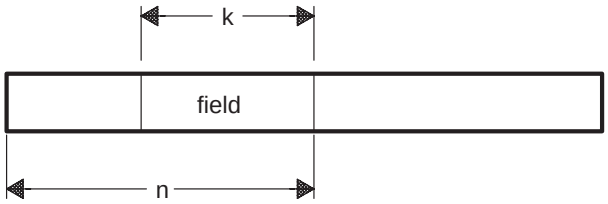
(c) EALU with @mf||ROTATE



8.4.6.1 Arbitrary Field Extract Using EALU||ROTATE

When you specify an EALU without a parallel multiply, an arbitrary field extract can be performed in a single operation. This requires a rotate amount different from the mask generate amount, as shown in Figure 8–13.

Figure 8–13. Arbitrary Field Extract




```
di\\n & %k          ; n = left rotate amount
                    ; k = field size
```

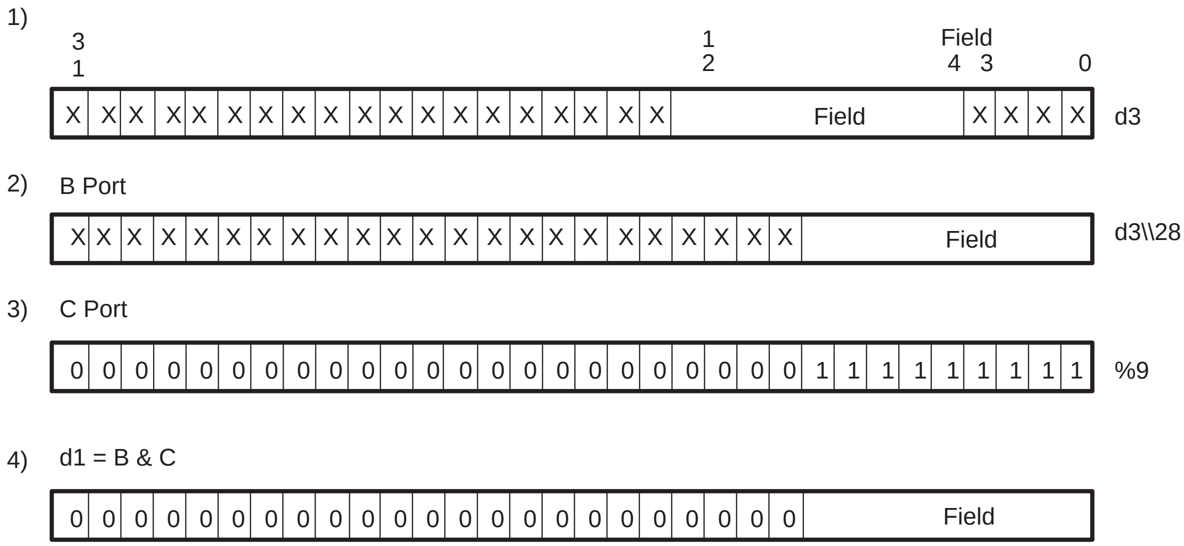
Example 8–25 shows how an EALU is used to extract a 9-bit field, starting at bit 4 in a 32-bit word. The input word is rotated by 28 to right-justify the desired field; then the 9 LSBs of the rotated word are extracted. Note that the rotate amount of 28 will actually be specified in the d0 register.

The assembler knows that for an EALU||ROTATE operation, the rotate amount must be specified in a register. If the source code specifies a constant rotate amount, the assembler encodes d0 as the source register that specifies the rotate amount (src3) and sets the 5 LSBs of the 32-bit constant associated with the EALU label to the specified rotate amount. Conversely, if the field size were a constant, the assembler would place the field size in the d0 register's default rotate amount and set src4 to d0. Moreover, two different constants are not allowed in an ealu expression, because only a single 5-bit constant will fit in d0.

In Example 8–25, the five LSBs of the constant associated with FIELD_EXT have a value of 0x1C (28). Therefore, if the d0 register is loaded with that constant at the time the EALU operation is performed, the desired field is extracted.

Example 8–25. Set-Up Field Extract EALU

```
   d5 = 9              ;field width
   d0 = FIELD_EXT      ;
;
; Rotate right by four bits and extract nine LSBs
; The ALU function is B & C
;
   d1 = ealu (FIELD_EXT: d3\\28 & % d5)
```



In Example 8–25, 1) shows d3 as input to the barrel rotator. The barrel rotator produces line 2) which is the input to the B port of the ALU. Line 3) is the output of the mask generator and the input to the C port of the ALU. Line 4) shows the output of the ALU as written to the destination register.

8.4.6.2 Unrelated Rotate and ALU Operations

When the EALU function configured in d0 does not involve the input to the B port of the ALU, the barrel rotate operation can be entirely unrelated to the ALU operation.

In Example 8–26, the three-input ALU ANDs the inputs to the A and C ports and writes the result to a destination register; the input to the B port is ignored. The output from the barrel rotator is written to a second destination register. For this case, the EALU operation acts like a two-input ALU operation in parallel with an independent rotate operation.

Example 8–26. Unrelated Rotate and ALU Operations

```
d0 = MASK_ROT      ; EALU setup
d1 = ealu ( MASK_ROT: d2&d3 ‖ d7 = d6 \\ d5)
```

8.5 Multiply Operations

A multiply can be specified in parallel with an add, an add with right-shift, a subtract, a subtract with right-shift, or an extended EALU operation. All multiply operands must be D registers; they cannot be non-D registers or immediates.

A signed or unsigned 16-bit by 16-bit multiply can be performed in parallel with either an add or subtract performed by the ALU. This operation (referred to as MPY||ADD) is described in subsection 8.5.1.

For more general ALU operations, you can perform a multiply in parallel with an extended ALU (EALU) operation (referred to as MPY||EALU). This allows a multiply in parallel with any of the 256 Boolean or 256 arithmetic functions supported by the three-input ALU. When a multiply is performed in parallel with an EALU, the multiplier data path is controlled by bits in the d0 register. These bits can optionally select:

- Split multiply (two $8 \times 8 = 16$ multiplies in a single cycle),
- A rounded multiply (for maintaining precision when retaining only 16 bits of the 32-bit product), or
- A default left shift on the 32-bit output of the multiplier.

MPY||EALU is described in detail in subsection 8.5.2. Table 8–28 lists the topics discussed in this section.

Note that source code can specify a multiply without a parallel ALU operation. To support this, the assembler simply encodes a dummy parallel add with the same destination as the specified multiply. If the ALU has the same destination register as the multiply, the ALU operation is not performed and status is not set.

Table 8–28. Discussion of the Assembly Language Multiply Operations

Section	Topic
8.5.1	Using the MPY||ADD and MPY||SADD Operations
8.5.2	Multiplying in Parallel With an EALU Operation

8.5.1 Using the MPY||ADD and MPY||SADD Operations

The MPY|| ADD operation generates a 16 × 16 signed or unsigned multiply producing a 32-bit result in parallel with either an add or subtract by the ALU. Similarly, the MPY||SADD operation generates a 16 × 16 multiply and an add or subtract with one input shifted right by the negative of the shift amount specified in d0. Parallel operations by the global and local address unit can also be specified in the same instruction.

The general syntax for an MPY||ADD or MPY||SADD operation is:

```
dst2 = [sign] [ ⟦cond⟧ ] src3 * src4
   [ || dst = [ ⟦cond [.pro] ⟧ ] src2 + src1[ ⟦n ⟧src1–1]] [>>–d0]
```

or

```
dst2 = [sign] [ ⟦cond⟧ ] src3 * src4
   [ || dst = [ ⟦cond [.pro] ⟧ ] src2 – src1[ ⟦n ⟧src1–1]] [>>–d0]
```

Note that the ⟦ symbol represents required square brackets for the conditional operand. The following text describes the parameters in the MPY||ADD operation syntax.

- dst2

  The dst2 represents the multiply destination. It must be a D register.

- sign

  The sign operand indicates whether a multiply is signed or unsigned.

  - Signed multiply

    An **s** indicates signed multiply and is the default. A signed src3 operand is multiplied by a signed src4 operand.

  - Unsigned multiply

    A **u** indicates unsigned multiply. The src3 and src4 operands are unsigned.

- ⟦cond⟧

  The cond operand represents a conditional multiply and ALU operation. If one is conditional, the other must be also.

- src3

  The src3 operand represents the first multiply source operand. It must be a D register.

- src4

  The src4 operand represents the second multiply source operand. It must be a D register.

- dst1

  The dst1 operand represents the ALU destination. It can be a non-D register only if a global transfer is not specified in parallel; otherwise, it must be a D register.

- pro (n,c,v,z)

  The pro operand determines whether or not the status bits are protected.

- src2

  The src2 operand represents the input to the A port of the ALU. Any D register is allowed.

- src1

  For MPY||ADD, the src1 operand represents the input to the B port of the ALU. For MPY||SADD, src1 is the operand that is shifted right. It can be a non-D register only if a global transfer is not specified in parallel; otherwise, it must be a D register.

- src1[n]src1–1

  The src1[n]src1–1 operand represents a D register pair conditioned on the basis of the negative condition. The possible D register pairs are:

  - d7[n]d6
  - d5[n]d4
  - d3[n]d2
  - d1[n]d0

- >>–d0

  src1 can optionally be shifted right by –d0. The shift right is signed or unsigned on the basis of whether the multiply is signed or unsigned.

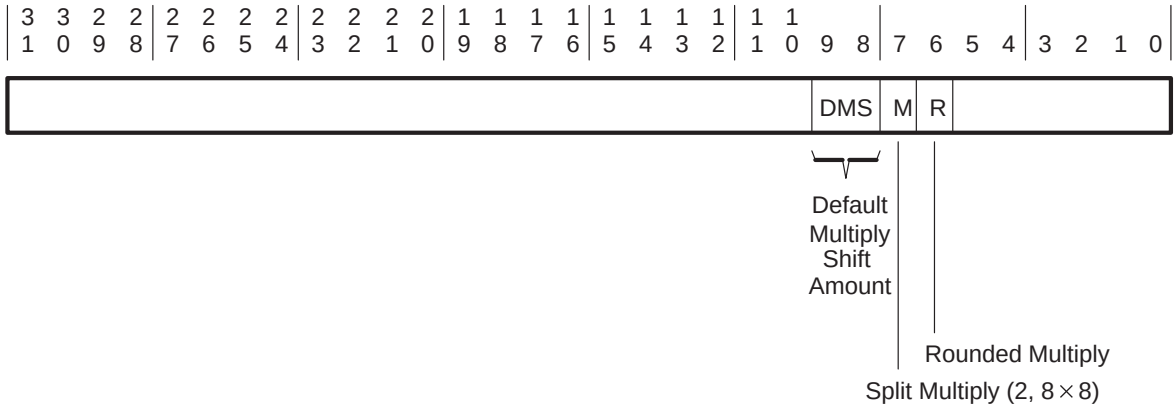
8.5.2 Multiplying in Parallel With an EALU Operation

A multiply can be performed in parallel with an extended EALU operation (referred to as MPY||EALU). This allows any of the 256 Boolean or 256 arithmetic functions supported by the ALU to be performed in parallel with the multiply. In addition to providing more flexibility in the ALU operations that can be specified in parallel with a multiply, MPY||EALU supports three multiply options specified in the d0 register:

- Split multiply,
- Scaling of the multiplier output, and
- Rounded multiply.

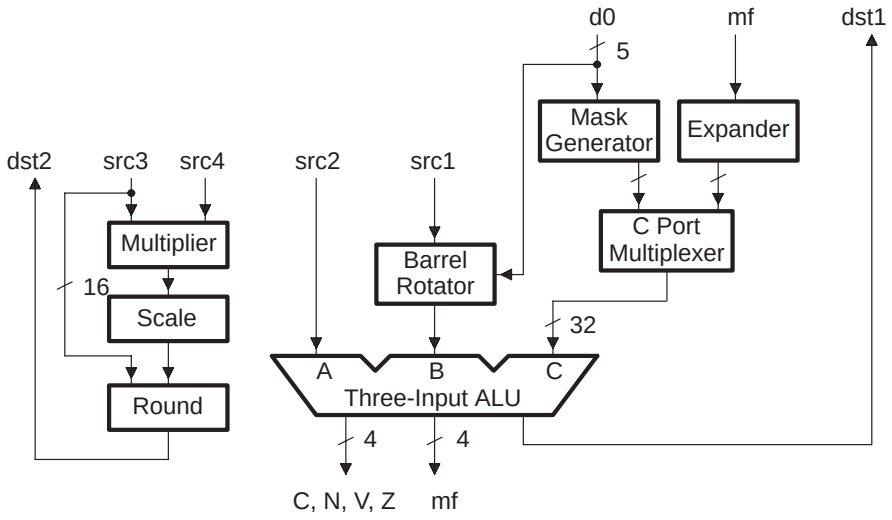
These options are not available with MPY||ADD or MPY||SADD.

Figure 8–14. d0 Multiply Fields



The MPY||EALU data path is shown in Figure 8–15. For the ALU data path, the default barrel rotate amount in the d0 register is used to specify both the rotate amount and the input to the mask generator. Looking at the multiplier data path, note that the output of the multiplier is passed through a scaler and then through rounding logic separate from the ALU data path. In addition, the C port of the ALU can receive the output of either the mask generator or the mf expander.

Figure 8–15. MPY||EALU Data Routing



This section describes the multiply hardware features supported by MPY||EALU. EALU operations performed by the ALU in parallel with the multiply are described in detail in Section 8.4, *The Extended Set of ALU Operations (EALU)*. The following sections describe the default multiply shift, split multiply, rounded multiply, and using ealu and ealuf in parallel with multiplies.

8.5.2.1 Default Multiply Shift

The multiplier data path contains hardware that can be used to shift the output of the multiplier left by 0, 1, 2, or 3 bits. For rounded multiplies, this shift occurs before the rounding logic, as described in subsection 8.5.2.3. For MPY||ADD and MPY||SADD, the multiply shift amount is always set to 0. For MPY||EALU operations, the DMS field in the d0 register specifies the left-shift amount applied to the output of the multiplier. When the shift is performed, the MSBs shifted out are discarded; zeros are shifted into the LSBs.

The multiply shift amount is specified for an explicit EALU by the shift left symbol “<<” followed by the shift amount, as shown in Example 8–27.

Example 8–27. Shifted Multiply

```
;;BEFORE                          d7     =      0x0
;;                                d6     =      0x1
;;                                d5     =      0x1234
        d0 =   DMS_EX
        .
        .
        d7 =   (d6 * d5)<< 2
        ||ealu(DMS_EX)
;;AFTER                           d7     =      0x48D0
```

Subsection 15.2.4, *Discrete Cosine Transform (DCT)*, provides another example of shifting (and rounding) with multiplies.

8.5.2.2 Split Multiply

When an MPY||EALU operation is executed, the M bit in the d0 register determines whether one 16 × 16 multiply (M=0) or a split multiply (two 8 × 8 multiplies (M=1)) is performed. Split multiply is specified in explicit EALUs by an “m” appended to the = or =u assignment operator for signed and unsigned split multiplies, respectively.

For split multiplies, the first operand (src3) contains two 8-bit unsigned numbers. The second operand (src4) contains two 8-bit numbers that can be treated as either signed or unsigned, according to the opcode. For details on the multiplier data formats, see subsection 4.7.3, *Performing Split Multiply*.

Example 8–28 shows a sample unsigned split multiply operation and its result.

Example 8–28. Split Multiply

```
;;BEFORE                    d7     =      0x0
;;                          d6     =      0x0201
;;                          d5     =      0x1111
        d0 =   SPLIT_EX     ; Spilt multiply (M) bit is
                            ; set in SPLIT_EX
        .
        .
        d7 =um d6 * d5
        ||ealu(SPLIT_EX)
;;AFTER                     d7     =      0x00220011
```

Note that since the split multiply is specified in d0 but signed/unsigned multiply is specified in the opcode, the EALU label SPLIT_EX in Example 8–28 could be used with:

```
d4 =m d3*d2
 || ealu(SPLIT_EX)
```

but could **not** be used with:

```
d4 =u d3*d2
 || ealu(SPLIT_EX)
```

There is no support for a rounded split multiply. The M and R bits should not both be set to 1 at the same time.

8.5.2.3 Rounded Multiply

When an MPY||EALU operation is executed, the R bit in d0 determines whether the full 32 bits of the multiplier result are written to dst2 (R = 0) or the multiplier result is rounded before being written to dst2 (R = 1). Rounding is specified for explicit EALUs by appending an “r” to the = or =u assignment operator, depending on the sign. If the rounding option is selected, the following occurs:

- The value of bit 15 of the multiplier result scaler output is added to the value of bits 31–16 of the same output. This results in half-bit rounding into the result’s upper halfword. The resulting 16-bit quantity is written to bits 31–16 of the dst2 register. (Note that a carry-out from bit 31 as a result of the half-bit rounding is not detected or recorded anywhere).
- Bits 15–0 of the output from the multiplier result scaler are discarded. Instead, bits 31–16 of src3 are written to bits 15–0 of dst2. This feature of the rounded multiply allows two back-to-back rounded multiplies to be performed on packed halfwords (as described in subsection 4.7.2, *Rounding and Scaling 16x16 Multiply Operations*).

Example 8–29 shows sample code and the output data from a rounded multiply that includes scaling.

Example 8–29. Rounded Multiply

```
;;BEFORE           d7 =  0x0
;;                 d6 =  0x87651234
;;                 d5 =  0x88

   d0 =  ROUNDED_EX      ; Rounding multiply (R) bit is set in
                         ; ROUNDED_EX
   .
   .
   d7 =ur (d6 * d5)<<2
      ||ealu(ROUNDED_EX)

;; Multiply:       0x1234 * 0x88         =  0x0009ABA0
;; Scale:          0x9ABA0 <<2           =  0x0026AE80
;; Round upper hw: 0x0026 + 1 (bit 15)   =  0x0027 (d7 upper-hw)
;; Swap
;;    d6(upper-hw)->d7(lower-hw)         =  0x8765 (d7 lower-hw)

;;AFTER:           d7 =  0x00278765
```

There is no support for a rounded split multiply. The M and R bits should not both be set to 1 at the same time.

Subsection 15.2.4, *Discrete Cosine Transform (DCT)*, provides another example of rounding and shifting with multiplies.

8.5.2.4 ealu and ealuf

Frequency domain transforms such as DCTs and FFTs often require both multiply-shift-and-add and multiply-shift-and-subtract operations in the same inner loop. To allow both of these operations to be performed with the same d0 register setup, you can specify multiplies in parallel with two types of EALU operations:

- **ealu()**

  The ALU function code is the EALU_FC field in the d0 register (bits 26–19).

- **ealuf()**

  The ALU function code is the inverse of the EALU_FC field in the d0 register (bits 26–19). The carry-in starting value (specified by d0's C bit) is also inverted if the operation is arithmetic.

Using ealu() and ealuf() to perform complimentary operations in the same loop without having to reload the d0 register is helpful because some algorithms use these related operations.

The shift-and-add and shift-and-subtract B-port inverted ALU functions are frequently used together. You can write a tight loop (for example, for a radix-2 FFT butterfly) by using both ealu() and ealuf() to perform multiply-shift-and-adds and multiply-shift-and-subtracts without modifying the d0 register.

The explicit form of EALU will allow both ealu() and ealuf() forms to use the same label. The assembler can detect the inversion of the B port and change the opcode between ealu() and ealuf(). Example 8–30 shows an example of this.

Example 8–30. Using ealu and ealuf

```
d0 = SRA_SRS
d7 = d6 * d5
d7 = d4 * d3
   ||d2 = ealu(SRA_SRS: d3 + d7>>16)  ;Encoded using ealu().

d7 = d6 * d5
   ||d1 = ealu(SRA_SRS:d5 – d7>> 16)  ;Encoded using ealuf().
```

8.6 The Divide Operation

The PP performs division via an iterative subtraction technique. The divi (divide iterate) data unit operation supports this technique for unsigned values. Halfword and word unsigned division require 16 and 32 iterations, respectively. You can perform signed division by using the unsigned divi operation with some additional preparatory and clean-up steps.

This section explains how to use the divi operation to perform signed and unsigned division. Table 8–29 lists the topics discussed in this section.

Table 8–29. Discussion of the Divide Operation

Section	Topic
8.6.1	Mathematical Explanation of the divi Process
8.6.2	Using the divi Operation
8.6.3	Examples of Single divi Instructions
8.6.4	Examples of divi Subroutines
8.6.5	Signed Division
8.6.6	Divide Overflow
8.6.7	Placement of Data for Divide Iterate
8.6.8	Number of Iterations for Divide Iterate

8.6.1 Mathematical Explanation of the divi Process

Mathematically, the iterative process that the divi instruction uses to produce a divide is similar to the long-hand division method.

For example, take 11 divided by 3:

11 ÷ 3 = 3 remainder 2

This same division can be represented in binary using a 4-bit dividend and 2-bit divisor:

1011b ÷ 11b = 11b (quotient) + 10b (remainder)

Figure 8–16 shows the division step by step.

Figure 8–16. Step-by-Step Division

```
1)  11)1011                 1011b ÷ 11b

        0
2)  11)1011                 11b does not go into 10b (the current dividend);
                            therefore, the first bit of the quotient = 0. That quotient
        0                   bit multiplied by the divisor is 0b (0b × 11b = 00b).
3)  11)1011                 Subtract that from the 2 MSBs of the dividend to pro-
      –00                   duce the intermediate remainder.
       ——
       10 ◄———— intermediate remainder

        0
4)  11)1011                 To form the next current dividend, "bring down" the next
      – 00                  bit of the original dividend.
        ——
        101 ◄———— current dividend

        01
5)  11)1011                 The divisor (11b) goes into the current dividend (101b)
      – 00                  one time. This produces the next quotient bit. That
        ——                  quotient bit multiplied by the divisor is 11b
        101                 (1b × 11b = 11b). Subtract that from the 2 MSBs of the
        – 11                current dividend to produce the next intermediate
        ——                  remainder.
         10 ◄———— intermediate remainder

        011
6)  11)1011                 Bring down the next dividend bit to form the current
      – 00                  dividend. 11b goes into 101b one time for a quotient bit
        —                   of 1b. 1b times 11b subtracted from 101b produces a
        101                 final remainder of 10b.
        – 11
        ——
         101 ◄———— current dividend
         – 11
         ——
          10 ◄———— final remainder
```

The final result is a quotient of 11b and a remainder of 10b.

8.6.2 Using the divi Operation

You can specify a parallel global transfer with any divi instruction. However, you cannot specify a parallel local transfer with a divi instruction when you use a conditional operation.

The divi instruction performs unsigned division. You can perform signed division by using the divi instruction; however, it requires a few extra preparatory and clean-up steps. In general, divi is used with a 64-bit dividend and a 32-bit divisor to produce a 32-bit quotient with a 32-bit remainder. Other divisions (such as a 32-bit dividend divided by a 16-bit divisor producing a 16-bit quotient with a 16-bit remainder) are subsets of the general solution.

In order to avoid divide overflow, the magnitude of the 32 MSBs of the 64-bit dividend must be less than the magnitude of the 32-bit divisor. The 32 MSBs of the 64-bit dividend must be contained in an odd D register, and the 32 LSBs of the 64-bit dividend must be contained in the mf register.

Each iteration of the 1-bit divide technique consists of the following three steps:

Step 1: Create the new current dividend by concatenating (bringing down) the next unused bit in the original dividend to the LSB of the current intermediate remainder.

Step 2: Check to see if the divisor is less than (can be divided into) the current dividend.

Step 3: If the divisor is less than or equal to the current dividend, subtract the divisor to get the new intermediate remainder. If the divisor is greater than the current dividend, the new intermediate remainder is the current dividend.

The divi instruction accomplishes these steps as follows (refer to Figure 8–17):

For step 1, the new current dividend is created by shifting the original dividend in the barrel rotator left by one bit and by shifting the mf register left by one bit. The new LSB of the barrel rotator comes from the MSB of the mf register. The current dividend is the output of the barrel rotator and is input to the B port of the ALU. It is also saved in another register for possible future reference.

The divi instruction restructures steps 2 and 3. It always does the subtract operation that is specified in step 3, but you can make the subtract conditional by choosing whether to use the result of the subtraction or the value before the subtraction. To make the subtract conditional, use the conditional source selection on the negative condition.

- If the prior result is positive or zero, the result of the prior subtraction is used.
- If the prior result is negative, the value before subtraction is used.

The ALU output provides the result of the subtraction. The saved output of the barrel rotator is the value before subtraction.

The general syntax for the divi operation is:

dst1=[⟦cond[.pro]⟧] divi (src2,dst2= [⟦cond⟧] src1 [⟦n⟧ src1–1])

Note that the ⟦ symbol represents required square brackets for the conditional operand. The following text describes the parameters in the divi operation syntax.

- dst1

  The dst1 register is the destination for the result of the ALU operation, which is the intermediate remainder for each iteration:

  intermediate remainder = current dividend + negated divisor

  The dst1 register must be an odd D register (d1, d3, d5, or d7) and is normally the same register as src1.

- ⟦cond⟧

  The cond operand is one of the conditions specified in Table 8–36, *Condition Codes*. This allows the dst1 and dst2 registers to be updated conditionally. If you don't specify a condition, the u (unconditional) condition is default. Since there is room in the divi opcode for only one condition code to be specified, if you specify any of the three conditional operations, all of the conditional operations that you specify in that instruction must be the same.

  For the divi instruction, the usefulness of a conditional update of destination registers is very limited. You can specify only the negative condition for the conditional source selection ⟦**n**⟧ **src1–1**, and ⟦n⟧ is essential for correct divi operation. Once you specify ⟦n⟧, you cannot use the other two conditions, or the divi instruction produces incorrect results for divide iterate.

- dst2

  The dst2 register is the destination of the result of the barrel rotator. This is the current dividend. The dst2 register must be an even D register (d0, d2, d4, or d6) and is normally the same register as src1–1.

- src1

  The src1 register is normally initialized to the 32 MSBs of the 64-bit dividend. The src1 register is a conditional (⟦n⟧) input to the barrel rotator. This input is used to form the current dividend if the prior divi subtraction set the negative flag. The src1 register must be an odd D register (d1, d3, d5, or d7) and must form an odd:even pair of D registers along with the src1–1 register. The src1 register is normally the same register as dst1.

- src1–1

  The src1–1 register is a conditional (⟦n⟧) input to the barrel rotator. This input is used to form the current dividend if the prior divi subtraction cleared the negative bit. The src1–1 register must be an even D register (d0, d2, d4, or d6) and must form an odd:even pair of D registers along with the src1 register. The src1–1 register is normally the same register as dst2.

- src2

  The src2 register contains the negated divisor. You must negate the divisor before storing it in the src2 register. The src2 register can be any D register that is not the dst1, dst2, src1, or src1–1 register.

Before the beginning of a series of divi iterations, you must set the src1:mf register pair to the 64-bit dividend and the src2 register to the negative of the 32-bit divisor.

During the divi iterations, the quotient is generated 1 bit per iteration and is stored in the mf register. For each iteration, the ⟦n⟧ conditional chooses between the src1 register (which contains the subtracted amount) and the src1–1 register (which contains the unsubtracted amount) as the input to the barrel rotator. The barrel rotator, along with the mf register, is shifted to the left by 1 to form the current dividend. The quotient bit is stored in the LSB of the mf register. During this shift, the MSB of the mf register goes to the LSB of the barrel rotator output, and the sign (S) of the barrel rotator input (its MSB) is used with the carry out of the ALU operation to determine the n bit of the status register. For the divi instruction, the n bit equals the logical OR of the carry and sign (N = C | S). Therefore, **the negative status bit indicates a positive result, which is the opposite of its usual behavior.**

Also during the divi iterations, the intermediate remainder (the result of the ALU operation) is stored in the dst1 register, and the current dividend (the 32 MSBs of the result of the 1-bit shift of the barrel rotator and mf register) is stored in the dst2 register.

Although not required for every divi instruction (most notably the last divi instruction in a series of divi iterations), you should follow these conventions:

- The dst1 register should be the same register as src1 because the remainder of iteration k is conditionally used as the source of the current dividend of iteration k + 1.
- The dst2 register should be the same register as src1–1 because the current dividend of iteration k is conditional used as the source of the current dividend of iteration k + 1.

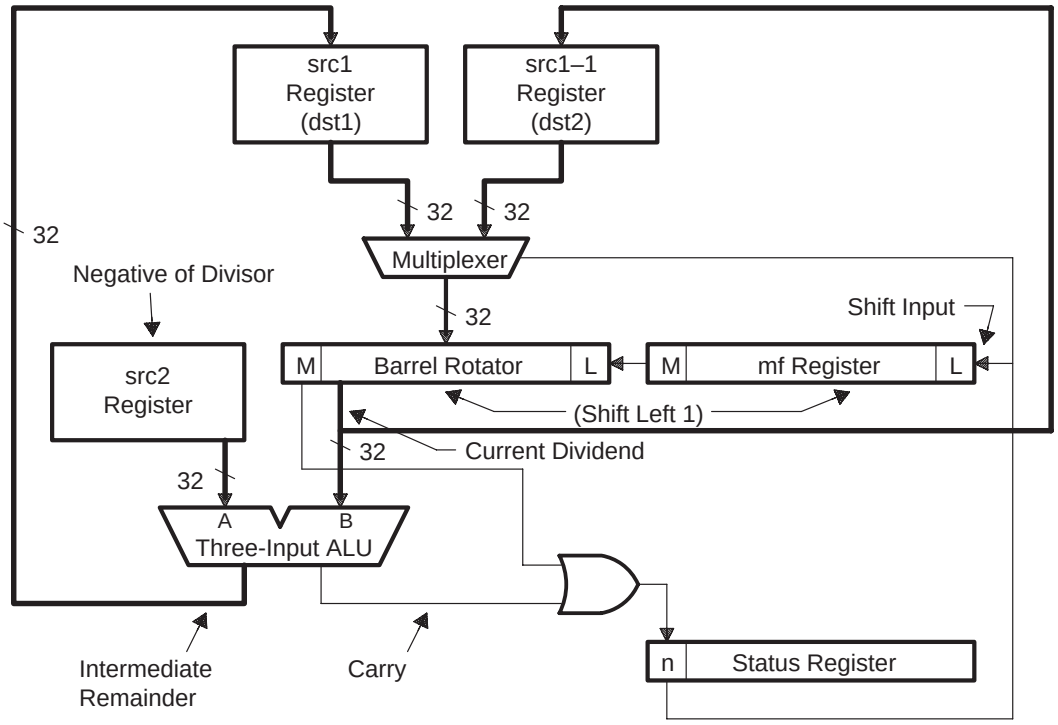
If you follow these conventions, the syntax for the divi instruction is:

dst1=[⟦cond[.pro]⟧] divi (src2,src1–1=[⟦cond⟧] src1 [⟦n⟧ src1–1])

When the dst1 register is the same as the src1 register, the intermediate remainder (the result of the ALU operation) is stored in the src1 register. When the dst2 register is the same as the src1–1 register, the current dividend is stored in the src1–1 register. This is important because it is the (src1 ⟦n⟧ src1–1) conditional that determines which operand (the k – 1 current dividend or the k – 1 remainder) is used for the current dividend.

Figure 8–17 shows the hardware configuration for the three-input ALU and the registers used to perform divi iterations. This diagram shows the typical divi configuration (dst1 = src1 and dst2 = src1–1). Note that the C port input to the ALU is disregarded for the divi instruction.

Figure 8–17. ALU Configuration for divi Iterations



Legend: M: MSB
L: LSB
n: Negative status bit

Figure 8–18 shows the encoding for the divi operation.

Figure 8–18. The divi Encoding

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	21	20	19	18	9	8	7	6	
0	1	1	1	0	1	0	-	-	-	dst2			dst1			src1			-	-	-				src2			XXXXXX			cond.			c	r	x	N	C	V	Z	X	Adstbnk		X	s1bank		X	5
0	1	1	1	0	1	0	-	-	-	dst2			dst1			src1			-	-	-				src2			XXXXXX			cond.			c	r	x	N	C	V	Z	XXXXXXXXXXXXXXXXXXXXXXXXXX						6, 6a, 8

divi

Notes: 1) cond. = condition code.

2) c = 0-unconditional src1, 1-conditional asrc1.

3) r = 0-unconditional ALU result, 1-conditional ALU result.

4) N = 0-don’t protect N status bit, 1-protect N status bit (not recommended).

5) C = 0-don’t protect C status bit, 1-protect C status bit.

6) V = 0-don’t protect V status bit, 1-protect V status bit.

7) Z = 0-don’t protect Z status bit, 1-protect Z status bit.

8.6.3 Examples of Single divi Instructions

Example 8–31 through Example 8–34 show the syntax of single divi instructions. Assume the following values for these examples:

```
src1     .set  d1
src1M1   .set  d0
src2     .set  d4
```

Example 8–31 is a typical first divi instruction in a series of divi iterations. The optional **[⟦n⟧ src1M1]** is not specified for the first iteration; you usually want the beginning dividend used as the source for the current dividend. The updates of src1 and src1M1 are unconditional (the default), and none of the three status bits (c, v, or z) are protected.

Example 8–31. First divi Instruction

```
src1 = divi( src2, src1M1 = src1 )
```

Example 8–32 shows a typical divi instruction within a series of divi iterations. The optional **[⟦n⟧ src1M1]** is specified so that src1 or src1M1 is conditionally chosen as source of the current dividend. The updates of src1 and src1M1 are unconditional (the default), and none of the three status bits (c, v, or z) are protected.

Example 8–32. Typical divi Instruction

```
src1 = divi( src2, src1M1 = src1 [n] src1M1 )
```

Example 8–33 is the same as Example 8–32, except the updates of src1 and src1M1 are conditional on the n condition. This is not very useful, because you wouldn't want to conditionally update src1 or src1M1. This example is included to illustrate the syntax.

Example 8–33. Typical divi Instruction With Conditional

```
src1 = [n] divi( src2, src1M1 = [n] src1 [n] src1M1 )
```

Example 8–34 same as Example 8–33, except the c status bit is protected.

Example 8–34. Typical divi Instruction With Conditional and Protected Status

```
src1 = [n.c] divi( src2, src1M1 = [n] src1 [n] src1M1 )
```

8.6.4 Examples of divi Subroutines

Example 8–35 and Example 8–36 illustrate divide iterate subroutines with various sizes of dividends, divisors, quotients, and remainders. Both of the examples shown have been chosen such that divide overflow does not occur.

Example 8–35 is for a 32-bit dividend divided by a 32-bit divisor with a 32-bit quotient plus a 32-bit remainder. This is performed in 32 iterations.

Example 8–35. Unsigned Divide Iterate (32 Bits)

```
   src1        .set  d1
   src1M1      .set  d0
   src2        .set  d4
   dividend    .set  d6
   divisor     .set  d7
   remainder   .set  d0 ; same as src1M1

; do a global and local transfer to load the dividend and divisor into
; local d registers

   dividend = *a8 || divisor = *a0

   src2 = –divisor   ; negate the divisor (data unit)
    || mf = dividend ; mf = 32 LSBs of 64-bit dividend (global)
    || src1 = &*(0)  ; make 32 MSBs of 64-bit dividend = all 0s (local)

; The 64-bit dividend = src1:mf register pair. Since this dividend is
; only 32 bits, you should put it into the least significant half of the
; 64-bit dividend (the mf register) in order to avoid divide overflow.

divide_32by32_32p32: ; the divide subroutine

   lrse2 = 29        ; 30 loops + 2 delay slots == 32 diviS

   src1 = divi(src2, src1M1 = src1)             ; 1st divide iterate
   src1 = divi(src2, src1M1 = src1 [n] src1M1)  ; 2nd divide iterate
   src1 = divi(src2, src1M1 = src1 [n] src1M1)  ; 3rd through 32nd divide
                                                ; iterate

   remainder =[n] src1  ; Remainder may be in src1M1 register or src1
                        ; register, depending on the n status bit

; At the end of the 32 divide iterates, the 32-bit quotient is found in
; the mf register, and the 32-bit remainder is found in the remainder
; register.
```

Example 8–36 is for a 16-bit dividend divided by a 16-bit divisor with a 16-bit quotient plus a 16-bit remainder. This is performed in 16 iterations.

Example 8–36. Unsigned Divide Iterate (16 Bits)

```
   src1        .set  d1
   src1M1      .set  d0
   src2        .set  d4
   dividend    .set  d6
   divisor     .set  d7
   remainder   .set  d0 ; same as src1M1

; do a global and local transfer to load the dividend and divisor into
; local d registers. Here it is assumed that the 16-bit divisor is placed
; in the least significant half of the *a0 register before the example
; begins. Also, it is assumed that the 16-bit dividend is placed in the
; least significant half of the *a8 register before the example begins.

   dividend = *a8 || divisor = *a0

   src2 = –divisor      ; negate the divisor (data unit)

   mf = dividend << 16  ; mf = 32 LSBs of 64-bit dividend
    || src1 = &*(0)     ; make 32 MSBs of 64-bit dividend = all 0s

; The 64-bit dividend = src1:mf register pair. Since this dividend is
; only 16 bits, you should put it into the least significant half of the
; 64-bit dividend (the mf register) in order to avoid divide overflow.
; Here it is put into the 16 MSBs of the mf register in order to do only
; 16 iterations to form the 16-bit quotient.

divide_16by16_16p16:    ; enter the divide subroutine

   lrse2 = 13           ; 14 loops + 2 delay slots == 16 diviS

   src1 = divi(src2, src1M1 = src1)             ; 1st divide iterate
   src1 = divi(src2, src1M1 = src1 [n] src1M1)  ; 2nd divide iterate
   src1 = divi(src2, src1M1 = src1 [n] src1M1)  ; 3rd through 16th divide
                                                ; iterate

   remainder = [n] src1 ; Remainder may be in src1M1 register or src1
                        ; register, depending on the n status bit

; At the end of the 16 divide iterates, the 16-bit quotient is found in
; the 16 LSBs of the mf register, and the 16-bit remainder is found in
; the 16 LSBs of the remainder register.
```

8.6.5 Signed Division

You can perform signed division by using the unsigned divi instruction as follows:

1) Save the sign of the dividend. This becomes the sign of the nonzero remainder.
2) Determine the sign of the quotient by taking the exclusive OR of the sign of the dividend and the sign of the divisor.
3) Take the absolute value of both the dividend and the divisor.
4) Negate the divisor.
5) Generate the unsigned quotient and remainder by using the absolute value of the dividend and the negative of the absolute value of the divisor.
6) If needed, correct the sign of both the quotient and remainder.

Example 8–37 is for a 32-bit dividend divided by a 32-bit divisor with a 32-bit quotient plus a 32-bit remainder. This is performed in 32 iterations with sign.

Example 8–37. Signed Divide Iterate

```
   src1        .set  d1
   src1M1      .set  d0
   src2        .set  d4
   dividend    .set  d6
   divisor     .set  d7
   quotient    .set  d3
   remainder   .set  d0 ; same as src1M1 register

; do a global and local transfer to load the dividend and divisor into
; local d registers

   dividend = *a8 || divisor = *a0

   src2 = –|divisor|    ; negate the magnitude of the divisor
   mf = |dividend|      ; mf = absolute value of the dividend
    || src1 = &*(0)     ; make 32 MSBs of 64-bit dividend = all 0s

; Remember 64-bit dividend = src1:mf register pair. Since this dividend
; is only 32 bits, you should put it into the least significant half of
; the 64-bit dividend (the mf register) in order to avoid divide
; overflow.

divide_32by32_32p32: ; the divide subroutine

   lrse2 = 29           ; 30 loops + 2 delay slots == 32 diviS

   src1 = divi(src2, src1M1 = src1)             ; 1st divide iterate
   src1 = divi(src2, src1M1 = src1 [n] src1M1)  ; 2nd divide iterate
   src1 = divi(src2, src1M1 = src1 [n] src1M1)  ; 3rd through 32nd divide
                                                ; iterate

; At the end of the 32 divide iterates, the 32-bit quotient is found in
; the mf register, and the 32-bit remainder is found in the src1 or
; src1M1 register.

; Immediately after the last divi instruction, do a conditional move of
; src1 into src1M1. If the n status bit is 0, the remainder was found in
; src1M1, and the move does not take place. If the n status bit is 1, the
; remainder was in src1, and it is moved into the remainder register.

   remainder = [n] src1

; Do a bitwise XOR of the original dividend and divisor. The result of
; the XOR of the two MSBs (the sign bits) is found in the n status bit.
; Then do a conditional negate of the quotient on the basis of that
; status bit. This corrects the sign of the quotient.

   src2 = dividend ^ divisor  ; src2 is a convenient scratch pad register
   mf = [n] – mf              ; conditional sign correction of quotient

; Make the sign of the 32-bit remainder (found in the remainder register)
; the same as the sign of the original dividend.

   src2 = 0 + dividend           ; src2 is a convenient scratch pad
                                 ; register
   remainder = [n] – remainder   ; conditional sign correction of
                                 ; remainder
```

8.6.6 Divide Overflow

Divide overflow occurs when the magnitude of the result of a divide is larger than the largest magnitude of the divisor. A simple example of divide overflow using a 4-bit dividend and a 2-bit divisor with a 2-bit quotient would be:

```
1011 ÷ 11 = 11 (11 ÷ 3 = 3 + a remainder of 2)
  |     |     |
  |     |     2-bit quotient (no divide overflow)
  |     2-bit divisor
  4-bit dividend

1100 ÷ 11 = 0100 (12 ÷ 3 = 4)
  |     |     |
  |     |     4-bit quotient (divide overflow)
  |     2-bit divisor
  4-bit dividend
```

In the second example, the resultant quotient requires three bits to correctly represent it, requiring four iterations (rather than two iterations) in order to produce it. This is a divide overflow (the result overflowed the 2-bit quotient field). Any 4-bit dividend divided by any 2-bit divisor overflows when the quotient is greater than 3. In general, a large number divided by a small number can result in a quotient that is greater that the largest possible divisor.

This concept can be expanded to larger width dividends and divisors. Previous examples that involved 32-bit dividends divided by 32-bit divisors stated that divide overflow would not occur. This is true because it was insured that the magnitude of the 32-bit quotient would not exceed the largest magnitude possible from a 32-bit divisor. This was insured by placing the real 32-bit dividend in the least significant half of the 64-bit dividend. This works because if the magnitude of the most significant half of the 64-bit dividend is less than the magnitude of the 32-bit divisor, the quotient is guaranteed to be small enough as to not cause divide overflow.

If divide overflow occurs using the divi instruction you can do one of two things:

- ❑ Use more iterations for a larger width result. For example, if you were performing iterations to produce a 16-bit quotient, you could use 32 iterations to produce a 32-bit quotient.
- ❑ Use a different method for producing results for integer divides.

8.6.7 Placement of Data for Divide Iterate

Figure 8–19 through Figure 8–21 show the format for data placement in the registers, src1, src2, and mf for several divide iterate routines. Assume that no divide overflow has occurred and you are doing unsigned divide iterate.

The src1–1 register (the even D register of the odd:even pair src1:src1–1) is not used during the first iteration; as long as the optional ⟦**n**⟧ **src1–1** field is not specified for that divi instruction, the src1–1 register does not have to be initialized.

The remainder is found in the src1 or the src1–1 register, depending on the n status bit of the last divide iterate. See subsection 8.6.5 for an example of determining which register.

Figure 8–19. 32-Bit Dividend/32-Bit Divisor = 32-Bit Quotient + 32-Bit Remainder in 32 Iterations

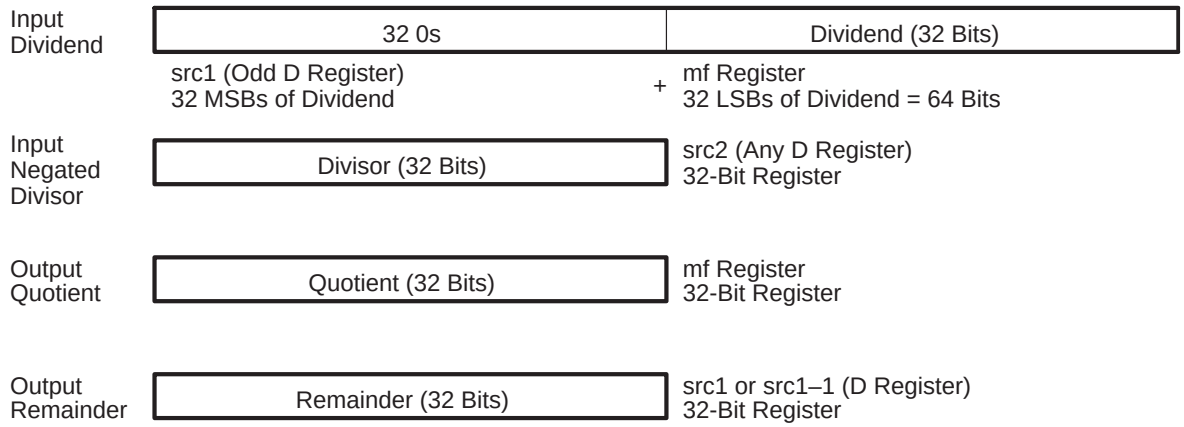


Figure 8–20. 32-Bit Dividend/16-Bit Divisor = 16-Bit Quotient + 16-Bit Remainder in 16 Iterations

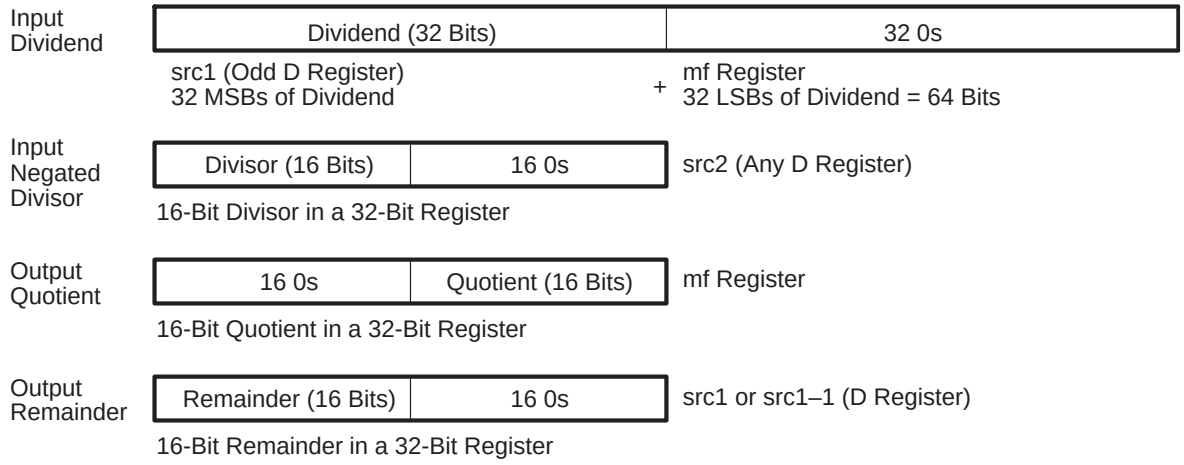
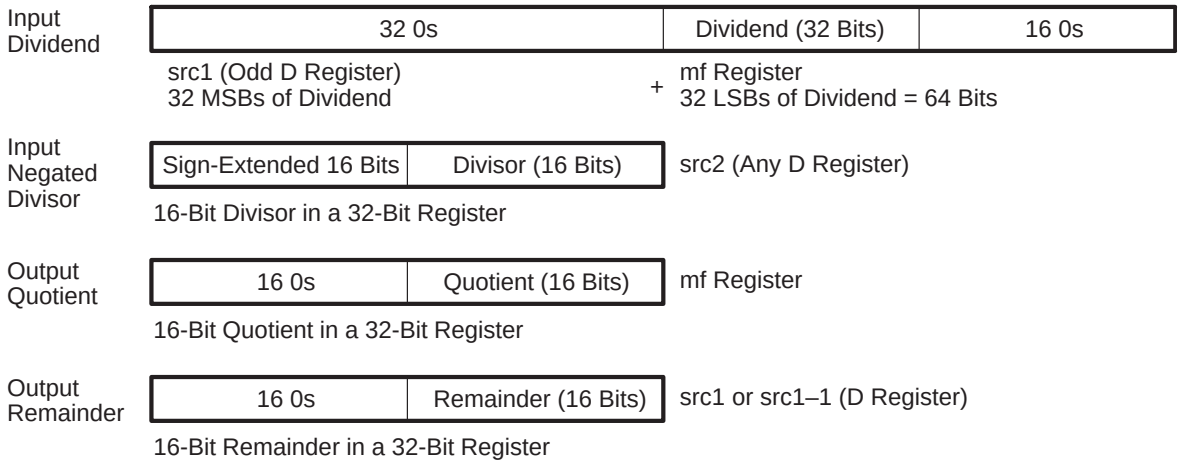


Figure 8–21. 16-Bit Dividend/16-Bit Divisor = 16-Bit Quotient + 16-Bit Remainder in 16 Iterations



8.6.8 Number of Iterations for Divide Iterate

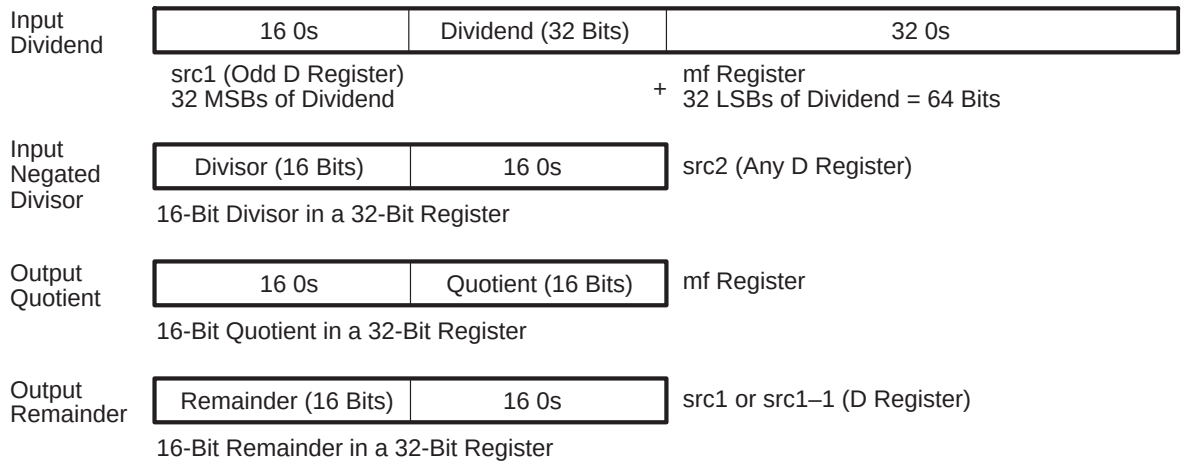
In Figure 8–19 through Figure 8–21, the relationship between data placement and the number of iterations required to produce a result may not be readily apparent. Data placement and number of iterations are directly related.

In general, the number of iterations needed to produce a quotient is equal to the number of shifts required to align the LSB of the dividend with the LSB of the divisor. Keep in mind that the numerical significance of the MSB of the 32-bit divisor has the same value as the MSB of the 64-bit dividend.

For the example shown in Figure 8–21, when the 16-bit divisor is in the least significant half of the src2 register, it requires 16 shifts left for the LSB of the 16-bit dividend to line up to the LSB of the 16-bit divisor when the dividend is placed in the most significant half of the mf register.

An alternative data placement (which also requires only 16 iterations) is shown in Figure 8–22.

Figure 8–22. Alternate Placement 16-Bit Dividend/16-Bit Divisor = 16-Bit Quotient + 16-Bit Remainder in 16 Iterations



8.7 Miscellaneous Operations

You can use the data unit portion of the instruction to specify one of several miscellaneous operations that do not involve the data unit hardware. These miscellaneous operations are:

- dint
- eint
- nop

8.7.1 dint

The dint operation allows you to globally disable interrupts. The syntax for this operation is:

dint

Interrupts are disabled and the global interrupt enable bit in the pc register is reset to zero. Global and/or local transfer(s) can be specified in parallel.

Interrupts (except the non-maskable emulation interrupt) are disabled before the next instruction is fetched.

The dint operation should not be encoded in parallel with a reti3 operation since the reti2 will try to load the global interrupt enable bit at the same time and reti will dominate.

The status register contents are unaffected by this operation.

8.7.2 eint

The eint operation allows you to globally enable interrupts. Individual interrupt sources can then be enabled by their corresponding enable bit in the the inten register. The syntax for this operation is:

eint

The eint operation sets the global interrupt bit, or pc⟦G⟧ bit. Global and/or local transfer(s) can be specified in parallel. When the global interrupt enable bit is zero, all interrupt sources are disabled regardless of the contents of the inten register.

When an instruction containing the eint operation is fetched, the interrupt sources, with their corresponding enable bit set in the inten register, are enabled before the next instruction is fetched.

An instruction that modifies the intflg or inten registers executes in the execute pipeline stage, while the eint operation executes immediately upon being fetched. Therefore, you must insert one delay slot instruction between initializing or modifying the intflg and inten registers and enabling the interrupts. Otherwise, interrupts will be enabled before the desired contents are in the inten and intflg registers.

An example of enabling interrupts with only the task interrupt enabled is shown in Example 8–38.

Example 8–38. Interrupt Enable Sequence

```
inten=1\\14   ; Enable only task interrupt (W set to 0).
intflg=intflg ; Clear interrupt flags.
nop           ; Delay slot between clearing
              ; intflg and globally enabling
              ; interrupts.
eint          ; Globally enable interrupts.
```

The eint operation should not be encoded in parallel with the reti3 instruction, because the reti2 instruction will try to load the global interrupt enable bit at the same time; the reti2 instruction has higher priority.

The status register contents are unaffected by the eint operation.

8.7.3 nop

The nop operation tells the data unit not to perform an operation. The syntax for this operation is:

nop

The data unit performs no operation, in parallel with any global and/or local bus transfer(s). No status bits are modified by a nop instruction.

When only a parallel transfer is specified, the assembler automatically inserts a nop in the data unit portion of the opcode.

8.8 Parallel Transfers

One or two of the following operations can be specified in parallel with a data unit operation in a single PP instruction:

- **Load (**memory-to-register transfer)
- **Store** (register-to-memory transfer)
- **Address Unit Arithmetic** (computed address-to-register transfer)
- **Move** (register-to-register transfer)

These operations are referred to collectively as **parallel transfers**.

A parallel transfer operation falls into one of the following two categories:

- **Global Transfer**

  A parallel transfer performed by the global address unit (this can be a load, store, address unit arithmetic, or move).

- **Local Transfer**

  A parallel transfer performed by the local address unit (this can be a load, store, or address unit arithmetic).

Parallel transfers are described in more detail in the following sections. Table 8–30 lists the topics discussed in this section.

Table 8–30. Discussion of Parallel Transfers

Section	Topic
8.8.1	Addressing Modes
8.8.2	Memory-to-Register Transfers (Loads)
8.8.3	Register-to-Memory Transfers (Stores)
8.8.4	Register-to-Register Transfers (Moves)
8.8.5	Address Unit Arithmetic Operations
8.8.6	Stack Operations

8.8.1 Addressing Modes

The local and global address units both support all 12 addressing modes shown in Table 8–31. The addressing mode is independently specified for each address unit. The modes are used for loads, stores, and address unit arithmetic operations.

All of the addressing modes use indexed addressing. The index can be specified in an index register or directly in the opcode by an immediate offset. Throughout this user's guide, the term offset refers specifically to an immediate index.

Two offset sizes can be used:

- **Short Offset**

  Unsigned 3-bit immediate index for halfword or word accesses; unsigned 4-bit immediate index for byte accesses. A short offset can be used in an addressing mode, regardless of what operations are specified in parallel in the instruction.

- **Long Offset**

  Unsigned 15-bit offset for halfword or word accesses; unsigned 16-bit offset for byte accesses. A long offset can be used only when

  - There is only one parallel transfer,
  - The data unit operation does not use a non-D register, and
  - None of the operations in the instruction is conditional.

  These restrictions apply because the long offset requires the opcode bits normally used to specify the other parallel transfer operation.

An index can be added to or subtracted from the contents of the specified address register to produce a new address. The address is generated either before the address is output to the crossbar (preindexing) or afterwards (postindexing).

- Preindexing modes allow either modifying or not modifying the address register with the new generated address.

- Postindexing modes always modify the address register.

8.8.1.1 Operators for Parallel Transfer

Table 8–31. PP Addressing Modes

Description	Syntax	
	Unscaled Index	**Scaled Index**
Postaddition of index register with modify	*(am++=xn)	*(am++=[xn])
Postsubtraction of index register w/ modify	*(am– –=xn)	*(am––=[xn])
Preaddition of index register	*(am+xn)	*(am+[xn])
Presubtraction of index register	*(am–xn)	*(am–[xn])
Preaddition of index register w/ modify	*(am+=xn)	*(am+=[xn])
Presubtraction of index register w/ modify	*(am–=xn)	*(am–=[xn])
Postaddition of offset with modify	*(am++=offset)	*(am++=[offset])
Postsubtraction of offset w/ modify	*(am– –=offset)	*(am– –=[offset])
Preaddition of offset	*(am+offset)	*(am+[offset])
Presubtraction of offset	*(am–offset)	*(am–[offset])
Preaddition of offset w/ modify	*(am+=offset)	*(am+=[offset])
Preaddition of offset w/ modify	*(am–=offset)	*(am –=[offset])

The following bullets describe the syntax shown in Table 8–31.

- *** (Address indirection)**

  Indicates that generated address is used as a pointer. The contents of the memory location is accessed for the load or store.

  & (Reverse address indirection)

  Used before address indirection (that is, &*) to indicate that the computed address, rather than the contents of a memory location it points to, is transferred to the destination register. This is used for address unit arithmetic operations.

- **am (Address register)**

  Must be a global address register for global transfers or a local address register for local transfers.

  - **global transfer: am** can be a8, a9, a10, a11, a12, a14 (sp), or a15 (zero).
  - **local transfer: am** can be a0, a1, a2, a3, a4, a6 (sp), or a7 (zero).

- **xn (Index register)**

  Must be a global index register for global transfers. Likewise, must be a local index register for local transfers. The contents of the index register can optionally be scaled by the data size (see subsection 8.8.1.4).

  - **global transfer: xn** can be x8,x9,x10.
  - **local transfer: xn** can be x0,x1,x2.

- **offset (Unsigned offset)**

  The offset can optionally be scaled by the data size (see subsection 8.8.1.4).

  - **long offset:** 15 bits (word or halfword access) or 16 bits (byte access)
  - **short offset:** 3 bits (word or halfword access) or 4 bits (byte access)

Table 8–32 shows some addressing shortforms supported by the assembler. Although all of the addressing modes use indexed addressing, an offset of 0 can be used to perform an access to the original address. You can implement immediate addressing by using one of the appropriate read-as-zero address registers (**a15** for the global address unit, **a7** for the local address unit) as the base address with the desired address of the offset in an index register. Preincrement and postincrement or decrement are simply special cases of the addressing modes that use an offset of 1 with scaling by the data size.

Table 8–32. Supported Addressing Shortforms

Description	Syntax	Equivalent Addressing Mode
Indirect	* am	* (am+0)
Indirect Index	* xn	* (zero + xn)
Absolute	* offset	* (zero + offset)
Postincrement	* am++	* (am++= [1])
Postdecrement	* am– –	* (am– –= [1])
Preincrement	* ++am	* (am+= [1])
Predecrement	* – –am	* (am–= [1])

The assembler also supports memory accesses as structure references. When you access an element in a predefined data structure (using the .struct directive), an alternate syntax shown below is supported by the dot (.) operator.

```
*am.element
```

This is equivalent to an access using either preaddition or pre-subtraction of an offset from the address in register **am**. The offset for **element** is calculated by the assembler on the basis of the structure definition. The data size for the access and whether the element of the structure is signed or unsigned are also determined from the structure definition.

8.8.1.2 Byte Addressability

The MVP is a byte-addressable machine with a 32-bit memory address. The PP supports byte, halfword (16-bit), and word (32-bit) memory accesses. By default, word is the data size for a memory access. To specify a halfword or byte access, use a modifier after the assignment operator (=h or =b, respectively). For halfword or byte loads, you can specify either sign extension (default) or zero extension (=uh, =ub).

8.8.1.3 Aligned Access

Memory accesses are always performed to aligned addresses. For word accesses, the two LSBs of the generated address are ignored. For halfword accesses, the LSB of the generated address is ignored. Although the transfer hardware ignores the two or one LSBs for word or halfword accesses, respectively, the address register update (if specified) writes the entire 32 bits of the generated address.

8.8.1.4 Scaled Indexing

An index can optionally be scaled by the data size of the access being performed. This allows data-size independent indices; this is a useful feature for table look-ups and various other applications.

Scaling is indicated by square brackets [ ] around the index register or offset. When scaling is specified, the value in the index register or offset is shifted left according to the data size of the memory access being performed: 2-bit shift left of index for word accesses, 1-bit shift left of index for halfword accesses, and 0-bit shift left of index for byte accesses.

For word or halfword accesses, scaled indexing allows a wider access range for short and long offsets.

For byte accesses, the opcode bit that normally specifies scaling is used as an additional offset bit instead. This is the reason a short offset supports four bits for byte accesses but only three bits for halfword and word accesses. Likewise, a long offset for byte accesses is 16 bits, while for halfword and word accesses, it is only 15 bits. The corresponding immediate index ranges for the different data sizes and scaling options are shown in Table 8–33.

Table 8–33. Immediate Index Range

	Scaled Offset		Unscaled Offset	
Category	**Short**§	**Long**§	**Short**†	**Long**‡
byte	–	–	–15 to +15	–0x8FFF to +0x8FFF
halfword	–14 to +14	–0xFFFE to +0xFFFE	–7 to +7	–0x7FFF to +0x7FFF
word	–28 to +28	–0x1FFFC to +0x1FFFC	–7 to +7	–0x7FFF to +0x7FFF

† 4-bit short offset range possible for byte data size because the scale bit is used as an additional offset bit.

‡ 16-bit long offset range possible for byte data size because the scale bit is used as an additional offset bit.

§ Scaled offset must be a multiple of the data size. For scaled halfword, only even offsets within the specified range can be obtained; for scaled word, only offsets within the specified ranges that are a multiple of 4 can be obtained.

Notes:

- Scaling can be used only to produce post-scaled offsets that are a multiple of the data size (that is, multiple of 1 for byte, multiple of 2 for halfword, multiple of 4 for word).
- You can derive negative ranges by using a positive offset with an addressing mode that subtracts the offset from the base address register. Since the addressing modes allow the offset to be either added or subtracted, signed offsets can be handled by the assembler, provided that the offset magnitude fits into the allowed number of bits for the specified instruction. For example, if the assembly source code specifies a local transfer with an offset of –7, the assembler converts this to an equivalent operation that is supported: pre-subtraction of an offset of 7.

Example 8–39. Assembler Interpretation of Negative Immediate Index

```
d7 = *(a8 += -7)          ;  Becomes *(a8 -= 7).
```

The PP assembler assumes that offsets are specified in byte granularity in the assembly source unless scaling is specified explicitly with square brackets: [ ]. To take advantage of the wider access range for scaled offsets, the PP assembler converts unscaled offsets in the source code to scaled offsets of the equivalent unit of the specified data size granularity if possible.

Example 8–40 shows several instances of offset conversion by the assembler. If the source code specifies an unscaled offset of 0x8000 for a word load, the assembler converts this to a scaled access with an offset of 0x2000. If the desired offset is not a multiple of the data size (for example, an offset of 1 for a word access), it cannot be specified by a scaled access. Thus, an unscaled access is required.

In some cases, it is more natural for you to specify an offset in the source code already in the granularity of data size. In these cases, the scaled offset indicator [offset] should be used.

Example 8–40. Assembler Conversion Operation for Immediate Indices

```
    ;ASSEMBLY SOURCE                 OPERATION ENCODED BY ASSEMBLER
    ;-----------------               --------------------------
          d1 =w *(a2= dba+0x8000)  ; d1 =w *(a2= dba + [0x2000])
          d1 =h *(a2= dba+0x8000)  ; d1 =h *(a2= dba + [0x4000])
error ---> d1 =w *(a2= dba+0x8001)  ; error (0x8001 is not a multiple of 4 so
                                   ; it cannot be specified using a scaled
                                   ; access and it is not in the range of an
                                   ; unscaled long offset).
          d1 =w *(a2= dba+0x801)   ; d1 =w *(a2 = dba + 0x801)
          d1 =w *(a2+[1])          ; d1 =w *(a2 + [1])
          d1 =w *(a2+= –8)         ; d1 =w *(a2–= [2])
```

8.8.1.5 PP-Relative Addressing

In order to run the same code on different PPs, addresses must be calculated in a manner so that a PP can easily access its own local RAM. PP-relative addressing allows the base address for a PP's local data RAM0 (assembler keyword **dba**) or local parameter RAM (assembler keyword **pba**) to be used for address generation. A third keyword, **xba**, is provided for PP-relative addressing in source code that uses the linker for memory allocation. At link time, the linker changes xba to either dba or pba, depending on where the linker allocates space for the symbol.

PP-relative addressing acts exactly like the normal 12 addressing modes except that the base address is dba, pba, or xba instead of an address register. PP-relative addressing is allowed when only one parallel transfer is performed (local or global) or for a local transfer with a parallel move (that is not a field move) in the global address unit.

Table 8–34. PP-Relative Addressing

Description	Syntax
Postaddition of index register with modify	*xba,am=xba + xn
Postsubtraction of index register w/ modify	*xba,am=xba– xn
Preaddition of index register	*(xba+xn)
Presubtraction of index register	*(xba–xn)
Preaddition of index register w/ modify	*(am=xba + xn)
Presubtraction of index register w/ modify	*(am=xba – xn)
Postaddition of offset with modify	*xba,am=xba + offset
Postsubtraction of offset w/ modify	*xba,am=xba – offset
Preaddition of offset	*(xba+offset)
Presubtraction of offset	*(xba–offset)
Preaddition of offset w/ modify	*(am=xba + offset)
Presubtraction of offset w/ modify	*(am=xba – offset)

A generated PP-relative address can modify an address register. Once an address register contains a PP-relative address, subsequent memory accesses are PP-relative.

Example 8–41 shows how to set an address register to a PP's base address for local parameter RAM (LPRAM), local data RAM0 (LRAM0), local data RAM1 (LRAM1), and local data RAM2 (LRAM2). Using PP-relative addressing to initially set up addresses allows the same PP code to run on any PP but have each PP access only its own RAMs.

Example 8–41. PP-Relative Addressing

```
d7 =ub *dba, a8 = dba + 0xfff  ; Read lowest byte of PP's
                               ; LRAM0 and set a8 to address of highest
                               ; byte of PP's LRAM1.
*(a2 = pba + 0x200) = d7       ; a2 set to 0x200 offset from PP-relative
                               ; base address for Parameter RAM and store
                               ; d7.
*(pba + 0xfc) = a2             ; Store address for packet transfer
                               ; parameters pointed to by a2 to the PP-
                               ; relative address for
                               ; the linked-list pointer.
*(a1=dba+0x800) = x1           ; Set a1 to base of LRAM1 and store x1
                               ; there.
d1 = *(a11=dba + 0x8000)       ; Set a11 to base of LRAM2 and load
                               ; contents to d1.
d6 = *(dba + 0xffc)            ; Read highest word of PP's LRAM1.
a2 = &*(a3 = pba + 0x1b8)      ; Load a3 and a2 with the PP's task
                               ; interrupt vector address.
```

8.8.2 Memory-to-Register Transfers (Loads)

A load is a memory-to-register transfer. You can use one of the 12 addressing modes discussed in subsection 8.8.1 to generate the address for the memory access in the address pipeline stage. Then, during the execute stage, the data is read from memory over the global or local data port and written to the destination register.

8.8.2.1 Access Range

A global address unit load is always attempted over the global port. Thus, it can access any of the shared RAMs in a single cycle. If the address on the global port is not in the shared RAMs, a direct external access request is submitted to the transfer controller, and the PP stalls until request has been completed.

When a local transfer is specified without a parallel global transfer, the access is attempted over the global port because it is available and has a wider access range than the local port. This allows any address in the shared RAMs to be accessed in a single cycle.

Local transfers are attempted over the local port if there is a parallel global transfer. As long as the address for the memory access is located in one of the PP's local RAMs, the access can occur in a single cycle. If the address is not in the PP's local RAM, the access cannot be performed over the local port; the PP stalls, and the access is attempted during the following cycle over the global port. Then, if the address is not in the shared RAMs, a direct external access request is submitted to the TC.

8.8.2.2 Conditional Load

A global address unit load can be performed conditionally as specified by any of the 16 condition codes in Table 8–36. It is important to note that when a load is conditional, the memory access and any address register modification occur unconditionally. Only the data write to the destination register occurs conditionally. Thus, if a conditional load is specified from an illegal address, a PP_ERROR interrupt is sent to the MP, even if the condition is not true.

8.8.2.3 Using the Load Syntax

The syntax for the load operation is as follows:

dst = [sign] [size] [⟦ cond ⟧] ∗ addrexp

dst = [sign] [size] [⟦ cond ⟧] ∗ an.element

Note that the ⟦ symbol represents required square brackets. The following text describes the parameters in the load syntax.

- **dst** represents the destination register for the load operation. For a global transfer, the dst operand can be:
  - Any register when no parallel local transfer is specified.
  - Any lower register otherwise

  For a local transfer, the dst operand can be:
  - Any register when there is not a parallel global transfer and the data unit operation does not use a non-D register operand.
  - Any D register otherwise.

  Note that if the destination register is the comm, intflg, call, or br registers, then the data size for the load must be word. (32 bits).

- **sign** specifies how to extend halfword or byte data to 32 bits and can be **s** or **u**:
  - **s** indicates sign extension, which is the default.
  - **u** indicates unsigned (or zero) extension.
- **size** represents the transfer data size and can be one of the following:
  - **w** selects a word (32 bits), which is the default.
  - **h** selects a halfword (16 bits extended to 32).
  - **b** selects a byte (8 bits extended to 32).
- ⟦**cond**⟧ represents any of the 16 supported condition codes that are available only with global loads (see Table 8–36). Write of data to the destination operand is performed only if the condition is true. However, memory accesses and address-register modifies (if specified) are performed regardless of the condition.
- **addrexp** represents any of the 12 supported addressing modes involving a base address (**dba**, **pba**, **xba**, or an address register) and either an index register or immediate offset. You can optionally scale the index by the data size. Addressing modes are described in detail in subsection 8.8.1.
- **an.element** represents an element in a predefined data structure. You can use the dot operator to read from an element in a predefined data structure (defined using the .struct directive) that the base address points to. This is equivalent to loading from ***(an+offset)** where the assembler computes the offset to the specified element from the base address of the structure. The data size for the access and whether to sign or zero-extend is also determined from the structure definition.

  If sign or size is specified in the assembly instruction, it overrides the sign and size of the element in the structure definition, but the byte offset remains the same.

Example 8–42 shows several examples of the load operation.

Example 8–42. Using the Load Operation

```
d7 =b *a0++
x1 =uh *--a0
d3 =  *(a8+x9)
d7 =  *(a0=dba + 0x8000)
d2 =  *(a9=pba + 0x300)
sr =  *sp ++
d7 =  *a2.s_YPitch
x9 =  *(a8+[5])
   ||d7  =  *(a1+[x1])
br =[n] *(a9+[3])
```

8.8.3 Register-to-Memory Transfers (Stores)

A store is a register-to-memory transfer. You can use one of the 12 addressing modes to generate the address for the memory access in the address pipeline stage. Then, during the execute stage, the data in the source register is written to the specified address.

8.8.3.1 Access Range

Global address unit stores are always attempted over the global port. Thus, they can write to any address in the shared RAMs. If the address on the global port is not in the shared RAMs, a direct external access request is submitted to the transfer controller, and the PP stalls until the request has been completed. If the address is on-chip but not within the shared RAMs, this is a fault condition. The MP is notified by the TC that the PP has encountered a fault condition. Meanwhile, the PP remains stalled until it is reset by the MP.

When a local address unit store operation is specified without a parallel global memory access, the access is attempted over the global port because it is available and has a wider access range than the local port. This allows any address in the shared RAMs to be accessed in a single cycle.

A local address unit store is attempted over the local port if a parallel global transfer is specified. Thus, it can access any of the local RAMs in a single cycle. If the address on the local port is not in the PP's local RAMs, the PP stalls and the access is attempted on the following cycle over the global port. Then, if the address is not in the shared RAMs, a DEA request is submitted to the TC.

8.8.3.2 Store with Conditional Source Selection

A global address unit store can be performed with conditional source selection between the odd and even register in a D register pair (d0/d1, d2/d3, d4/d5, or d6/d7) on the basis of the negative condition code (n). If the sr⟦N⟧ bit is 1, the odd register is stored; if N is 0, the even register is stored. Conditional source selection is not supported for condition codes other than n (negative).

If the actual memory access in a store operation must be conditional (for example, when the address may be illegal if the condition is false), then a conditional branch is required. However, if the requirement is not a conditional memory access but a conditional store of a new value to memory, this can be achieved without conditional branching, as shown in Example 8–43.

Example 8–43. Conditional Store Effect

```
d7 = d6 – d5        ; Put in d7 new value that is to
                    ; be written to target address
                    ; if status is negative.
d7 =[nn] *a8        ; Overwrite new value with current
                    ; value at target address if
                    ; status is not-negative.
*a8 = d7            ; New value will be stored if neg-
                    ; ative status, otherwise current
                    ; value will be stored back over
                    ; itself.
```

8.8.3.3 Using the Store Syntax

The syntax for the store operation is as follows:

* addrexp = [size] src [⟦ n ⟧ src − 1]
* an.element = [size] src [⟦ n ⟧ src − 1]

Note that the ⟦ symbol represents required square brackets. The following text describes the operands in the store syntax.

- **addrexp** represents any of the 12 supported addressing modes involving a base address (**dba**, **pba**, **xba**, or an address register) and either an index register or immediate offset. You can optionally scale the index by the data size. Addressing modes are described in detail in subsection 8.8.1.
- **size** represents the transfer data size and can be one of the following:
  - **w** selects a word (32 bits), which is the default.
  - **h** selects a halfword (the least significant 16 bits of the register).
  - **b** selects a byte (the least significant 8 bits of the register).
- **src** represents the source register for the store operation. For a global transfer, the src operand can be:
  - Any register when no parallel local transfer is specified.
  - Any lower register otherwise

  For a local transfer, the src operand can be:
  - Any register when there is not a parallel global transfer and the data unit operation does not use a non-D register operand.
  - Any D register otherwise.

- **src[n]src–1** represents conditional source selection. You can perform store operations using conditional source selection between an odd (src) and even (src–1) register in a D register pair on the basis of the N status bit in the sr register. The D register pairs are:
  - d1[n]d0
  - d3[n]d2
  - d5[n]d4
  - d7[n]d6
- **an.element** represents an element in a predefined data structure. You can use the dot operator to write to an element in a predefined data structure (defined using the .struct directive). This is equivalent to storing an ***(an+offset)** where the assembler computes the offset to the specified element from the base address of the structure. The data size for the access is also determined from the structure definition.

  If the size is specified in the assembly instruction, it overrides the size of the element in the structure definition, but the byte offset remains the same.

Example 8–44 shows examples of the store operation.

Example 8–44. Using the Store Operation

```
*(a0 += 12)= x1
*a8++ =b d7
*a9.sPT_Options = d3
*(a2=dba+0x800) = d3
*(pba + 0xfc) = d1
*++a9 = d7[n]d6
*(a8--=[3]) =h x1
||*--a0 = d2
```

8.8.4 Register-to-Register Transfers (Moves)

A global transfer can specify a move (register-to-register transfer). In addition to normal moves, two types of field moves are supported: field extract moves and field replicate moves. Both normal and field moves can be performed conditionally with any one of the 16 supported condition codes (Table 8–36).

A move specifies a copy of the contents of one register to another register. The write to the destination register occurs during the execute stage of the pipeline. The general syntax for a move operation is:

dst = [g] [⟦ cond ⟧] src

Note that the ⟦ symbol represents required square brackets. The following text describes the operands in the move syntax.

- **dst** can be any PP register. You should note that moves to a15 (read-as-zero register) are reserved for issuing interprocessor commands, as described in Section 3.2.

- **g** attached to the assignment operator in a move operation forces the move to be done by the global unit. By using the expression **dst = src** (without the g modifier), you are leaving it up to the assembler to decide where the move is performed; this operation can be done by either the ALU or the global address unit.

  Note that a move performed by the ALU and a move performed by the global address unit are not equivalent since a move through the ALU sets status.

- ⟦**cond**⟧ represents any of 16 condition codes that specify the condition under which a move is to be performed (see Table 8–36).

- **src** can be any PP register. For field moves, src can be any D register.

The expression **dst = src** without a parallel data unit operation is encoded by the assembler as an ALU operation. If two D-register to D-register moves are coded in a single instruction without the g modifier on either equal sign, then the first one specified will be performed in the ALU. Some examples are shown in Example 8–45.

Example 8–45. Ambiguous Move Syntax

```
d7=d5              ; Encoded as an ALU operation.
                   ; (Affects status).
d7 =g d5           ; Explicit move by the global
                   ; address Unit. (Does not
                   ; affect status).
d1=d2 + d3         ; ALU operation.
      ||d7 = d5    ; Move performed by the global
                   ; address Unit. (does not
                   ; affect status).
```

If you specify a data unit move in parallel with a move performed by the global address unit, a swap operation can be performed. A conditional swap is demonstrated in Example 8–46.

Example 8–46. Conditional Swap

```
a15  =   d6 – d7         ; Compare d6 and d7.
d7   =[lt] d6            ; Swap if d6 is less
     ||d6 =[lt] d7       ; than d7.
```

8.8.4.1 Field Extract Move

Field extract moves allow a byte or halfword to be extracted from a source D register and written right justified (with either sign or zero extension) to a destination register (which can be any PP register). The general syntax for a field extract move is:

dst = [sign] size item [⟦ cond ⟧] src

Notice that the ⟦ symbol represents required square brackets. The following bullets describe the operands in the field extract move syntax.

- **dst** can be any PP register except the pc, call, or br. Note that moves to a15 submit commands as described in Section 3.2.
- **sign** specifies how to extend halfword or byte data to 32 bits and can be **s** or **u**:
  - **s** indicates sign extension, which is the default.
  - **u** indicates unsigned (or zero) extension.
- **size** specifies the size of extracted data from the source register and can be one of the following:
  - **b** selects a byte (8 bits extended to 32 bits by the move operation).
  - **h** selects a halfword (16 bits extended to 32 bits by the move operation).
- **item** indicates which byte or halfword is extracted.
  - **0** represents byte 0 (bits 7–0) or halfword 0 (bits 15–0).
  - **1** represents byte 1 (bits 15–8) or halfword 1 (bits 31–16).
  - **2** represents byte 2 (bits 23–16).
  - **3** represents byte 3 (bits 31–24).
- ⟦**cond**⟧ represents any of the 16 supported condition codes (see Table 8–36). If you use a condition code, the move occurs only if the condition is true.
- **src** selects one of the PP D registers (d0–d7).

Example 8–47 illustrates a field extract move.

Example 8–47. Field Extract Move

```
x0   =    0x0
d0   =    0xD40302E1         ;sets sr bit N to 1
x1   =b0       d0
x8   =b1       d0
x9   =ub3      d0
a8   =b3       d0
a9   =h0       d0
a10  =h1       d0
a11  =uh1      d0
x0   =ub0      [nn] d0     ;    If sr bit N does not
                           ;    equal 1, right justify
                           ;    byte 0 from d0 with
                           ;    zero extension and
                           ;    write to x0.
;; After:                  d0        =    0xD40302E1
;;                         x0        =    0x00000000
;;                         x1        =    0xFFFFFFE1
;;                         x8        =    0x00000002
;;                         x9        =    0x000000D4
;;                         a8        =    0xFFFFFFD4
;;                         a9        =    0x000002E1
;;                         a10       =    0xFFFFD403
;;                         a11       =    0x0000D403
```

8.8.4.2 Field Replicate Move

The rightmost byte (bits 7–0) or halfword (bits 15–0) in a D register can be replicated (four times or twice, respectively) and written to any PP register. The basic syntax for the field replicate move is:

dst = r size [⟦ cond ⟧] src

Note that the ⟦ symbol represents required square brackets. The following text briefly describes the parameters in the field replicate move syntax:

- **dst** can be any PP register except the pc, call, or br. Note that moves to a15 are used to submit commands, as described in Section 3.2.
- **r** indicates the move should be a field replicate move.
- **size** specifies the size of the data extracted from the source register and can be one of the following:
  - **b** selects a byte (8 bits).
  - **h** selects a halfword (16 bits extended to 32).
- ⟦**cond**⟧ selects one of the 16 supported condition codes (see Table 8–36). If you use a condition code, the move occurs only if the condition is true.
- **src** selects one of the PP D registers (d0–d7).

Example 8–48. Field Replicate Moves

```
        d0 =   0x12345678
        x1 =rb[nz] d0
        a3 =rh d0
;; After:                    d0     =      0x12345678
;;                           x1     =      0x78787878
;;                           a3     =      0x56785678
```

8.8.5 Address Unit Arithmetic Operations

You can use the local and global address units to perform general-purpose arithmetic in parallel with the data unit. This is referred to as address unit arithmetic.

For address unit arithmetic, the computed address is written to the destination register instead of being used as an address for a memory access. No memory access is attempted; therefore, the generated address does not have to be a memory address accessible by the PP. The address is computed in the address pipeline stage; the write to the destination register is performed in the execute pipeline stage.

It should be noted that addressing modes that specify modification of the base address register can be used with address unit arithmetic. Thus, the generated address can be used both to modify the base address register (in the address pipeline stage) and to write to the destination register (in the execute pipeline stage).

The syntax for address unit arithmetic is exactly like a load except that an **&** is included before the ***** to indicate that the computed address rather than the value it points to is transferred to the destination register. The general syntax for an address unit arithmetic operation is:

dst = [size] [⟦ cond ⟧] & ∗ addrexp

dst = [size] [⟦ cond ⟧] & ∗ an.element

Note that the ⟦ symbol represents required square brackets. The following text describes the operands in the above syntax.

- **dst** represents the destination register for the address unit arithmetic operation. For a global transfer, the dst operand can be:
  - Any register when no parallel local transfer is specified.
  - Any lower register otherwise

  For a local transfer, the dst operand can be:
  - Any register when there is not a parallel global transfer and the data unit operation does not use a non-D register operand.
  - Any D register otherwise.

  Note that if the destination register is either the call or br registers, then the data size for the load must be word (32 bits).

- **size** determines the number of bits from the result of the address computation that are written to the destination register. The size operand also affects the shift of the index register or offset if the index is scaled. Byte or halfword data sizes cannot be used if the address unit arithmetic destination is the call or br register.

  - **w** represents a write of the entire 32-bit result of the address computation to the destination register. If you use index scaling, the index is shifted left by two.
  - **h** represents a write of the 16 LSBs of the address computation result with sign extension to the destination register. If you use index scaling, the index is shifted left by one.
  - **b** represents a write of the 8 LSBs of the result of the address computation with sign extension to the destination register.

  The default data size for address unit arithmetic is word. However, byte or halfword data size can also be specified unless the destination is the pc (either call or br). For byte or halfword address unit arithmetic, the 8 or 16 LSBs (respectively) of the result of the address unit computation are sign-extended to 32 bits and written to the destination register.

Note:

Sign extension is always performed for byte and halfword address unit arithmetics. Sign extension is not an option.

- 〚**cond**〛 represents any of the 16 supported condition codes (see Table 8–36). They are available only when the address unit arithmetic is performed in the global unit, and no local transfer is done. Writing of data to the destination is performed only if the condition is true. However, address register modification, if specified, is performed regardless of the condition.

- **an.element** represents an element in a predefined data structure. You can use the dot operator to write the address for an element in a predefined data structure to the destination of an address unit arithmetic operation. This is equivalent to using **&*(an+offset)**; however, the offset for the specified element is computed by the assembler on the basis of the structure definition. Note that the data size for the address unit arithmetic is also determined by the structure definition; therefore, use caution with halfword and byte structure elements, due to sign extension.

8.8.5.1 Use With Read-as-Zero Register

Address unit arithmetic can be particularly useful with the read-as-zero registers (a15 or a7). In essence, this becomes equivalent to a load-immediate operation, as shown in Example 8–49. If only an offset or index register is specified, the appropriate read-as-zero register is assumed.

Example 8–49. Load-Immediate Using Address Unit Arithmetic With a15/7

```
d7=&*a15      ; d7  =  0 (using global address unit)
d6=&*a7       ; d6  =  0 (using local address unit)
d5=&*5        ; d5  =  5 (can use either address unit)
d4=&*[5]      ; d4  =  20 (5 shifted by word data
              ;        size)
```

8.8.5.2 Conditional Address Unit Arithmetic

You can specify any of the 16 condition codes in Table 8–36 with a global address unit arithmetic operation. When an address unit arithmetic operation is conditional, the base address register is always modified if modification is specified. Only the transfer of the computed address to the destination register is conditional.

Example 8–50 shows several samples of address unit arithmetic operation.

Example 8–50. Using an Address Unit Arithmetic Operation

```
                        ;;;;;;;;
                        ; Result
                        ;;;;;;;;
d7 = &*(a15)            ;  d7 =  0
x1 = &*(4)              ;  x1 =  4
x1 = &*([4])            ;  x1 = 16
d7 = &*(a9+x8)          ;  d7 = a9 + x8
sr = &*(0x24)           ;  sr = 0x24
d7 = &*(a0+=[10])       ;  d7 = a0 + 40
d3 =[n] &*(16)          ;  If Negative status bit
                        ;  is set to 1,
                        ;  d3 = 16
                        ;  Otherwise,
                        ;  no write to d3 is
                        ;  performed.
```

8.8.6 Stack Operations

The PP's stack pointer address register is used as a system stack pointer during interrupt servicing. The stack pointer is also visible in both the global address unit (as a14) and the local address unit (as a6). The keyword **sp** can be used in source code for the stack pointer. The assembler replaces sp with either a14 or a6 depending on whether it encodes the stack operation as a global transfer or a local transfer.

During the reset sequence (described in Section 9.6, *Reset*), the stack pointer is initialized near the top of the PP's local parameter RAM, at address 0x0100#7F0 (where # = PP number). The stack then grows towards lower addresses. As discussed in subsection 9.5, *Interrupt Processing*, state information is pushed onto the system stack (pointed to by the sp register) when an interrupt is taken. This information is restored during the return-from-interrupt sequence. Pushes are performed with predecrement stores; pops are performed with postincrement loads. Sample push and pop instructions are shown in Example 8–51.

Example 8–51. Example Push and Pop Instructions

```
*--sp = sr          ;    Push Status Register Contents
                    ;    onto the Stack.
sr = *sp++          ;    Pop Previous Status Register
                    ;    Contents from the Stack.
```

It is also common practice to decrement the stack pointer and then store or load elements to/from the stack using addressing modes that do not modify the stack pointer. This allows parallel stack accesses to be specified in the same instruction. The primary purpose of parallel stack operations is to reduce code size; due to contention, performing the two stack accesses uses the same number of cycles as if two instructions were used. However, the reduced code size can lead to less cache servicing and thus produce more efficient code.

Example 8–52. Parallel Stack Operations

```
*(sp–=[5]) = d7
*(sp+[3]) = a0
   ||*(sp+[1]) = d6
*(sp+[4]) = a8
   ||*(sp[2]) = d5
```

When you use parallel stack operations, they should not both modify the stack pointer, because the order in which the stack operations is performed is indeterminate and will lead to unpredictable behavior. Also, parallel stack operations should not access the same location on the stack.

8.9 Conditional Operations

The PP supports two types of conditional operations.

- **Conditional Assignment: =[cond]**

  A condition code controls whether the result of an operation is written to the designated destination register or discarded. If the condition is true, the result is written to the destination register and if it was an ALU operation, status setting is performed; if the condition is false, the result is discarded and no status setting is performed.

- **Conditional Source Selection: (src1[n]src1–1)**

  The negative condition code n determines whether the odd or even D register in a D register pair will be the source for an ALU operation or a store performed by the global address unit. If n = 1, the odd D register is used; if n = 0, the even D register is used.

Conditional operations are specified using bits from the local transfer portion of the opcode, so local transfers cannot be conditional or done in parallel with conditional operations.

Table 8–35 lists the topics discussed in this section.

Table 8–35. Discussion of PP Assembly Language Conditional Operations

Section	Topic
8.9.1	Condition Codes
8.9.2	Conditional Assignment
8.9.3	Conditional Source Selection
8.9.4	Status Bit Protection

8.9.1 Condition Codes

The 16 condition codes are presented in Table 8–36. The column labeled syntax specifies the symbol used in the assembly-language syntax to represent the condition. The condition is described in the center column, and the rightmost column shows the condition as a logical expression involving the N, C, V, and Z status bits of the status register.

Table 8–36. Condition Codes

Syntax	Description	Status Register Bit Combination
u	unconditional (default)	—
p	positive	~N&~Z
ls	lower than or same	~C|Z
hi	higher than	C&~Z
lt	less than	(N&~V)|(~N&V)
le	less than or equal	(N&~V)|(~N&V)|Z
ge	greater than or equal	(N&V)|(~N&~V)
gt	greater than	(N&V&~Z)|(~N&~V&~Z)
hs, c	higher than or same, carry	C
lo, nc	lower than, no carry	~C
eq, z	equal, zero	Z
ne, nz	not equal, not zero	~Z
v	overflow	V
nv	no overflow	~V
n	negative	N
nn	not negative	~N

8.9.2 Conditional Assignment

The assignment of the result of a data unit operation, load, or move to the destination register can be performed conditionally, as specified by one of the 16 condition codes shown in Table 8–36.

The condition code is evaluated according to the contents of the status register at the beginning of the execute stage of the pipeline, or in other words, the status inherited from the previous instructions. If the specified condition is true, the result of the operation is written to the destination register and if it was an ALU operation, status setting is performed. If the specified condition is not true, the result of the operation is not written to the destination register and no status setting is performed. Also, if the destination register is call or br, no status bits are set.

You can use conditional assignment to perform conditional branching, as described in Section 10.6, *Conditional Branches and Calls*, but you can also use it for any data unit operation, global transfer load, or move. Thus, the need for conditional branching and its two associated delay-slot instructions is greatly reduced.

When the data unit specifies two destinations (for example, a multiplier destination and an ALU destination), either both must be conditional, or both must not be conditional. A conditional or unconditional parallel transfer can be specified independently of the data unit. However, when parallel conditional operations are specified within an instruction, they all must be based on the same condition code. Example 8–53 shows some valid combinations of conditional assignment.

Example 8–53. Sample Conditional Assignment Combinations

```
d7 =[z] d6 * d5            ; Write multiply result
                           ; to d7 if zero status
                           ; bit is set (upon entry).
     ||d4 =[z] d3 + d2     ; Conditional write of
                           ; ALU result to d4.
     ||d1 =[z] *a9         ; Conditional load.
d7 = d6 * d5               ; Unconditional Multiply.
     ||d4 = d3 + d2        ; Unconditional ALU.
     ||d1 =[z] *a9         ; Conditional load.
d7 =[p] d6 * d5            ; Write multiply result to
                           ; d7 if positive status.
     ||d4 =[p] d3 + d2     ; Conditional write of
                           ; ALU result to d4.
     ||d1 = *a9            ; load (unconditional).
```

Example 8–54 shows how to perform a conditional register swap by using both an ALU operation and a register-to-register move based on status set by a compare (performed by a subtract) of the two registers. The result of Example 8–54 is essentially a sorting of d6 and d5 with the maximum value returned in d6 and the minimum value returned in d5.

Example 8–54. Conditional Swap

```
d7 = d6 – d5            ; Set status based on
                        ; compare of d6 and d5.
d6 =[lt] d5             ; If d6 is less than d5,
                        ; write d5 to d6.
      ||d5 =[lt] d6     ; If d6 is less than d5,
                        ; write d6 to d5.
```

8.9.3 Conditional Source Selection

You can make a conditional source selection between the odd and even registers in a D register pair according to the negative condition, **n**, for src1 of the ALU and/or the source of a parallel store performed by the global address unit. The D register pairs are:

- d7 and d6
- d5 and d4
- d3 and d2
- d1 and d0

The odd D register is selected if **n** is true; the even D register is selected if **n** is false.

Example 8–55 shows an instruction that uses conditional source selection. For the ALU operation, the second input to the ALU is d7 if the negative condition is true (sr⟦N⟧ = 1) or d6 if the negative condition is false (sr⟦N⟧ = 0).

Likewise for the parallel store, the d3 register is stored if the negative status bit is 1; the d2 register is stored if the negative status bit is 0. The instruction shown in Example 8–55 forms the inner loop for an implementation of a Bresenham line drawing on the PP. This is described in more detail in subsection 15.1.2, *Bresenham Line Drawing*.

Example 8–55. Inner Loop for a Bresenham Line

```
d5 = d5 + d7[n]d6    ; Add d7 or d6 based on
                     ; negative status bit.
||*a8++ = d3[n]d2    ; store appropriate guide
                     ; table value, d3 or d2, based
                     ; on negative.
```

You can specify conditional assignment and conditional source selection in the same instruction, as long as the condition code is negative, as shown in Example 8–56.

Example 8–56. Single Instruction Using Conditional Assignment and Source Selection

```
d7  =[n] d1 * d2
    ||d3=[n] d7 + d6
    ||*a8++= d5[n]d4
```

If the condition code is n, an ALU operation can specify both conditional source selection and conditional assignment. However, the conditional source selection is essentially meaningless because the result will be written and status will be set only if the negative condition is true and therefore only if the odd register is selected.

8.9.4 Status Bit Protection

The individual status bits in the status register—negative (N), carry (C), overflow (V), and zero (Z)—can be protected from modification by the normal status setting of an ALU operation. This is referred to as status protection. Status protection allows status information that is necessary for several operations to be retained.

In the PP opcode, status protection is associated with conditional operations. Status protection is specified in the assembly syntax by a dot and a lowercase list of the status bits to be protected appended to the condition code within the condition brackets of a conditional ALU operation. You can use any combination of the four status bits. The general format for status bit protection is shown below:

dst = ⟦ cond.pro ⟧ ALU operation

Note that the ⟦ symbol represents required square brackets.

- ⟦**cond**⟧ represents any of the 16 condition codes (see Table 8–36).
- **.pro** is a list of the status bits you want to protect in the status register:
  - n (negative)
  - c (carry)
  - v (overflow)
  - z (zero)

Example 8–57 shows a conditional add based on the negative status bit with protection of the carry and overflow status bits.

Example 8–57. Protection of C and V Status Bits

```
d3 =[n.cv] d4 + d5   ; Write ALU result to d3 only
                     ; if the negative status bit
                     ; is set (N=1). Protect the
                     ; carry and overflow status
                     ; bits from being modified.
```

The opcode format for status protection is contained in the conditional parallel transfer formats. Note that even though a conditional format is used, the ALU operation can still be performed unconditionally with =[.ncvz] or =[u.ncvz]. This indicates to the assembler that the write of the ALU output to the destination register is performed unconditionally (regardless of the status bits). Conditional source selection by the ALU or a store and/or a parallel load can still be performed conditionally when =[.ncvz] or =[u.ncvz] is used for the ALU assignment.

In the first instruction in Example 8–58, an unconditional subtract is performed with protection of the zero status bit. In parallel with the unconditional subtract, a load operation is specified with conditional assignment depending on the positive condition code.

In the second instruction, d1 is added to either the d3 or d2 register as specified by the negative status bit, and the result is written unconditionally to d1. The carry status bit is protected from status setting by the ALU operation. In parallel, a load operation is specified with conditional assignment based on the negative status bit.

Example 8–58. Status Protection for Some Unconditional ALU Operations

```
d4 =[u.z] d1 – d2        ;   Subtract with protection
  ||d7=[p]*a8            ;   of zero status bit.
                         ;   Conditional load based on
                         ;   positive condition code.
d1 =[u.c] d1 + d3[n]d2   ;   Add d1 to d3 (if N=1) or
  ||d7=[n]*a8            ;   d2 (if N=0). Result is
                         ;  written unconditionally to
                         ;   d1. Protect the carry
                         ;   status bit.
                         ;   Conditional load based on
                         ;   negative status bit.
```

Example 8–59 shows how status protection can be used. The first instruction compares the current pixel level to the previous lowest pixel level.

In the second instruction, the lowest pixel level is conditionally set to the current pixel level. Status protection is used to retain the status of the compare to conditionally update the coordinates of the low pixel.

Note that if status protection is not used, the status information required to conditionally update the coordinates of the lowest pixel would be lost.

Example 8–59 shows that when status protection is used, the third instruction can perform a conditional update of the coordinates for the lowest pixel level by using the status information from the compare performed in the first instruction.

Example 8–59. Same Status Used by Several Instructions

```
test:          .set   d7
pixel:         .set   d6
low_pixel:     .set   d5
low_pixel_x:   .set   d4
pixel_x:       .set   d3
low_pixel_y:   .set   d2
pixel_y:       .set   d1
       test = pixel – low_pixel    ; Compare pixel with previous low pixel.
       low_pixel =[lt.ncvz] pixel  ; Set low_pixel to level of pixel if it is
                                   ; lower than previous low pixel. Protect
                                   ; compare status.
       low_pixel_x =[lt] pixel_x   ; Set x coordinate of low pixel to x
                                   ; coordinate of pixel if pixel is less
                                   ; than previous low pixel.
      ||low_pixel_y =[lt] pixel_y  ; Set y coordinate of low pixel to y
                                   ; coordinate of pixel if pixel is less
                                   ; than previous low pixel.
```

Status protection can also be used to perform an operation that combines status setting from several operations. In Example 8–60, a return from subroutine is performed if the d1 register is equal to the d2 register or if the d3 register is less than the d4 register.

The first instruction compares the d1 register to the d2 register, setting the zero flag if they are equal. The second instruction compares the d3 register to the d4 register, protecting the zero status bit that contains necessary status information from the first instruction.

Then the third instruction performs a conditional return based on the equal (zero status bit) status information from the first instruction and the less than (based on negative and overflow status bits) status information from the second instruction.

Example 8–60. Using Status Protection for Combinatorial Conditions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;if ( d1 == d2 || d3 < d4) return()
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      d7 =  d1 – d2          ; Sets zero status bit if equal.
      d7 =[u.z] d3 – d4      ; Sets less than status if d3 is less than d4.
      br =[le] iprs          ; Returns if either condition is met.
```

8.9.4.1 Restrictions for Instructions Using Status Protection

- Status protection cannot be specified when a 32-bit immediate data operand is used by the ALU operation.
- Local transfers cannot be performed in parallel with an ALU operation specifying status protection.
- A global transfer in parallel with an ALU operation specifying status protection cannot use a long offset.

8.9.4.2 Status Protection for Software Writes to the Status Register

When the status register is the destination of an ALU operation that specifies protection of status bits, those status bits that are protected are not modified by the software write to the sr and are also not modified by status setting. Status protection is unconditional, even when the ALU operation is conditional.

Example 8–61. Status Protection for Data Unit Operations to the sr Register

```
sr   =[u.ncvz]  d2  + d3   ;nvcz always protected.
sr   =[u.cv]    d2  + d3   ;cv always protected.
                           ;nz written from ALU.
sr   =          d2  + d3   ;ncvz written from ALU.
sr   =[n.ncvz]  d2  + d3   ;ncvz always protected.
sr   =[n.cv]    d2  + d3   ;cv always protected. nz conditionally
                           ;written from ALU.
sr   =[n]       d2  + d3   ;ncvz conditionally written from ALU.
```

When status protection is specified, the protected status bits are also not modified by parallel address unit operations to the sr. For example, if a data unit operation specifies protection of the C status bit, then a parallel load to the sr will not write to the C status bit. Example 8–62 shows you additional examples of status protection.

Note:

Remember that assembler instructions cannot begin in column one.

Example 8–62. Status Protection With Parallel Transfers to the sr Register

```
d1 =[u.ncvz]  d2 + d3 || sr = x1       ;ncvz always protected.
d1 =[u.cv]    d2 + d3 || sr = x1       ;cv always protected.
                                       ;nz written from x1.
d1 =          d2 + d3 || sr = x1       ;ncvz written from x1
d1 =[n.ncvz]  d2 + d3 || sr = x1       ;ncvz always protected.
d1 =[n.cv]    d2 + d3 || sr = x1       ;cv always protected.
                                       ;nz written from x1.
d1 =[n]       d2 + d3 || sr = x1       ;ncvz written from x1
d1 =[u.ncvz]  d2 + d3 || sr =[n] x1    ;ncvz always protected.
d1 =[u.cv]    d2 + d3 || sr =[n] x1    ;cv always protected. nz
                                       ;written from x1 if condition
                                       ;is true.
                                       ;nz = status ALU result if
                                       ;condition is false
d1 =          d2 + d3 || sr =[n] x1    ;ncvz written from x1 if
                                       ;condition true.;ncvz = ALU
                                       ;status if condition false.
```

8.10 Parallel-Operation Combinations

The preceding sections of this chapter described separately the data unit operations and parallel transfers that can be performed. This section describes the combinations of these operations that can be specified in parallel in a single instruction.

To a large extent, the PP's parallel transfers are specified independently of the data unit operation. The specific operation performed by the data unit (whether the data unit operation is a base set ALU, a multiply with a parallel add, or one of the special operations specified in the data unit portion of the opcode) has no impact on the parallel transfers that can be specified in parallel. Any type of parallel transfer (load, store, address unit arithmetic, move, field move, or interprocessor command) can be specified in parallel with any of the supported data unit operations. However, certain operands, long offsets, conditional operations, and PP-relative addressing impose some restrictions on the data unit and parallel transfer operations that can be specified together in the same instruction.

The following sections describe several simple rules and identify the restrictions imposed on parallel operations with respect to a given operation. All of these rules and restrictions are then summarized in a table. Table 8–37 lists the specific topics covered in this section.

Table 8–37. Discussion of PP Assembly Language Parallel Operations

Section	Topic
8.10.1	Understanding the Basic Rules of Parallel Operations
8.10.2	Restrictions With Data Unit Operations
8.10.3	Restrictions With Global Transfers
8.10.4	Restrictions With Local Transfers
8.10.5	Restrictions With Two Parallel Transfers
8.10.6	Summary of Parallel Operations
8.10.7	Programming Exercises

8.10.1 Understanding the Basic Rules of Parallel Operations

The following four subsections describe the restrictions placed on parallel operations from various points of reference. A summary of what is legal is also shown in Table 8–38. It may seem like a large number of restrictions to remember; however, these simple rules will help.

- An unconditional data unit operation that uses only D register operands can be specified in any instruction, regardless of what the specified parallel transfers are.
- If you specify one of the following in an instruction, only one parallel transfer can be specified.
  - Conditional operation (data or global unit)
  - Long offset (local or global unit)
  - Non-D ALU operand
- In order to specify two parallel transfers, none of the following can be used:
  - Long offset
  - Conditional operation
  - Non-D ALU operand
  - Non-D local transfer source or destination
- When the data unit operation and/or a global transfer are conditional, no local transfer can be specified in parallel.
- Only one operation can specify a non-D source and/or destination. If either a data unit operation or a local transfer specify a non-D source or destination, then no global transfer can be specified in parallel.

8.10.2 Restrictions With Data Unit Operations

Restrictions are imposed on the parallel transfers that can be specified in parallel with a data unit operation that uses a non-D register operand, a 32-bit immediate operand, or is conditional. These restrictions are summarized below. Unless otherwise noted, operations are implied to be unconditional and have only D register or 5-bit immediate operands.

❑ Data Unit Operation

For an unconditional data unit operation that does not use a non-D register or 32-bit immediate operand, any combination of parallel transfers can be specified in the same instruction, regardless of which data unit operation being performed.

❑ Conditional Data Unit Operations

When the data unit operation is conditional, no local transfer can be specified in parallel. A global transfer can be specified, provided that it does not use a long offset.

❑ Non-D Data Unit Operations

When a data unit operation uses a non-D register operand for src1 and/or dst, no global transfer can be specified in parallel. A local transfer can be specified, provided that it does not use a long offset and does not specify a non-D register source or destination.

❑ Conditional Non-D Data Unit Operations

When the data unit operation is conditional and has a non-D register operand, no operations can be specified in parallel. A local transfer cannot be specified in parallel, because the data unit operation is conditional; a global transfer cannot be specified in parallel, because the data unit operation uses a non-D operand.

❑ 32-Bit Immediate Data Unit Operations

When the data unit operation uses a 32-bit immediate for source 2, no parallel operations can be specified. The data unit operation can be conditional, but status protection of individual status bits cannot be specified. ALU source1 must be a lower register.

8.10.3 Restrictions With Global Transfers

Anytime a global transfer is specified, a parallel data unit operation cannot use a non-D register or 32-bit immediate operand. A parallel local transfer must use a short offset or index register; it cannot use a long offset. Additional restrictions are imposed on parallel operations if the global transfer uses PP-relative addressing, is conditional, or uses a long offset.

- **Any Global Transfer**

  A parallel local transfer must use a short offset or index register and cannot specify a non-D register source or destination.

- **PP-relative Addressing**

  If a global transfer uses PP-relative addressing, a local transfer cannot be specified in parallel.

- **Conditional**

  If a global transfer is conditional, a local transfer cannot be specified in parallel. A conditional global transfer must use an index register or short immediate offset; it cannot use a long offset.

- **Long Offset**

  If a global transfer uses a long offset, a local transfer cannot be specified in parallel.

8.10.4 Restrictions With Local Transfers

Anytime a local transfer is specified, conditional operations cannot be specified in the instruction. A global transfer in parallel with a local transfer must use a short offset or index register; it cannot use a long offset or PP-relative addressing. If the local transfer uses a long offset or non-D register source or destination; no parallel global transfer can be specified and the data unit operation cannot use a non-D register operand. If the local transfer uses PP-relative addressing, a parallel global transfer can be only a move or an interprocessor command.

- **Any Local Transfer**

  Neither local transfers nor any parallel operations can be conditional. A parallel global transfer must use a short offset or index register and cannot be PP-relative.

- **PP-Relative Addressing**

  When a local transfer uses PP-relative addressing (subsection 8.8.1.5), a parallel global transfer can be only a move (but not a field move) or an interprocessor command; a parallel global transfer cannot be a load, store, address unit arithmetic, or field move.

- **Non-D Local Transfer**

  When a local transfer uses a non-D register source or destination, a parallel global transfer cannot be specified. The data unit operation cannot use a non-D register operand.

8.10.5 Restrictions With Two Parallel Transfers

In order to obtain full utilization of the parallel hardware on the PP, it is desirable to perform a data unit operation in parallel with both a global and local transfer as often as possible. The data unit operation and parallel transfers are subject to some restrictions when both a global and local transfer are specified. They are summarized in Table 8–38.

❑ **Data Unit Operation Restrictions**

- The data unit operation cannot use a non-D or 32-bit immediate operand.
- The data unit operation cannot be conditional.

❑ **Global Transfer Restrictions**

- Only an index register or short offset can be used; a long offset cannot be used.
- The global transfer cannot use PP-relative addressing (subsection 8.8.1.5).
- The global transfer cannot be conditional.
- The global transfer source (for store) or destination (for load or address unit arithmetic) must be a lower register.

❑ **Local Transfer Restrictions**

- Only an index register or short offset can be used; a long offset cannot be used.
- The local transfer cannot specify a non-D register source (for store), or destination (for load or address unit arithmetic).
- Unless the parallel global transfer is a move (not including field moves) or an interprocessor command, the local transfer cannot use PP-relative addressing.

8.10.6 Summary of Parallel Operations

Table 8–38 summarizes the supported parallel operations. It contains a synthesis of all of the restrictions described in the previous sections. Table 8–38 is ordered according to the parallel transfer opcode formats (described in Figure A–1), but it also includes the data unit's base set ALU with a 32-bit immediate format that requires the entire PP opcode.

Table 8–38. Summary of Parallel Transfer Formats

Format	ALU Operands		Cond	Status Protection	Global Transfer				Local Transfer			
					move	Load/Store/AUA			Load/Store/AUA			
	dst1	src1			src → dst	S/D	Index	Rel	S/D	Index	Rel	Port
Double parallel	D	D	No	No	—	Lower	X/short	No	D	X/short	No	Local
Move||local	D	D	No	No	Any → Any	—	—	—	D	X/short	Yes	Local
Field move||local	D	D	No	No	D → Any	—	—	—	D	X/short	No	Local
Global (long offset)	D	D	No	No	—	Any	X/long	Yes	—	—	—	—
Local (long offset)	D	D	No	No	—	—	—	—	Any	X/long	Yes	Global
Non-D DU||local	Any	Any	No	No	—	—	—	—	D	X/short	Yes	Global
Conditional move	D	D	Yes	Yes	Any → Any	—	—	—	—	—	—	—
Conditional field move	D	D	Yes	Yes	D → Any		—	—	—	—	—	—
Conditional global	D	D	Yes	Yes	—	Any	X/short	Yes	—	—	—	—
Conditional non-D DU	Any	Any	Yes	Yes	—	—	—	—	—	—	—	—
32-bit immediate base ALU	Any	Lower	Yes	No	—							

Note: DU = Data unit
AUA = Address unit arithmetic
— = Not applicable

The following bullets describe the columns in Table 8–38:

- dst1

  Indicates the type of register that the destination of the ALU can be. D indicates that the ALU destination must be a D register. Any indicates that the ALU destination can be either a D or non-D register. Note that dst2 (the destination of the multiplier or barrel rotator, which is not shown in the table) must always be a D register.

- src1

  Indicates the type of register that ALU source operand 1 can be. All other data unit source registers must always be D registers.

- Cond

  Indicates whether the data unit operation and/or the global transfer can be conditional.

- Status Protection

  Indicates whether the ALU operation can protect individual status bits from modification by status setting or writes to the sr register.

- src → dst

  Indicates what type of register the source (for a move or field move) can be, followed by what type of register the destination can be.

- S/D

  Indicates the type of register that the source for a store or the destination for a load or address unit arithmetic can be.

- Index

  Indicates whether the index must be an index register or short offset (X/short) or whether it can be an index register or long offset (X/long).

- Rel

  Indicates whether the corresponding operation can use PP-relative addressing.

- Port

  Indicates whether the load or store performed by the local address unit is first attempted over the local port or the global port. If an access is attempted over the local port to an address that is not in the local RAM, the pipeline stalls and the access is retried over the global port when it is free.

Example 8–63 shows an instruction for each of the parallel transfer opcode formats and the 32-bit immediate base set ALU format. Different data unit operations and parallel transfers have been chosen for each instruction to illustrate the variety of operations that can be performed in parallel. The selected data unit operation and parallel transfer operations do not imply any fixed combinations.

Example 8–63. Sample Parallel Operations

```
d5  =   d5 + d3<<3            ; Double parallel
    ||d3 = *a8++
    ||*(a0+[x1]) = d5
d7  =   d6 * d5               ; Move||local
    ||d4 = d4 + d7
    ||x0 = d3
    ||d5 = *a1++
d3  =   (d3&@mf) | (d4&~@mf)  ; Field move||local
    ||x0 =b2 d2
    ||d1 = &*(5)
d6  =   1 + lmo(d2)           ; Global (long offset)
    ||d3 = *(a8= pba + 0x200) ; (with PP-relative
                              ; addressing)
d6  =   d5^d3                 ; Local (long offset)
    ||d1 = *(a0= pba + 0x200) ; (with PP-relative
                              ; addressing)
x2  =   d2 + d1               ; Non-D DU||local
    ||*a3 = d6
d3  =   d4 | d1               ; Cond. global
    ||d2 =[nn] *(a10 + [2])
d3  =[p] d5 + d2              ; Cond. move
    ||x1=[p] d3
d3  =[p] d5 + d2              ; Cond. field move
    ||x1=rh d3
x8  =[z.nz] d2 + 4            ; Cond. non-D DU
br  =   0x20000380            ; 32-bit immediate DU
```

As shown throughout the following sections, Table 8–38 is a valuable reference for determining what operations can be specified in parallel with a given operation or to identify whether a given set of parallel operations in an instruction is legal.

8.10.6.1 Determining Whether an Instruction Is Legal

Table 8–38 can be used to determine if an instruction such as the one shown in Example 8–64 is legal.

Example 8–64. Legal Instruction

```
d7 = d3 & d1
    ||x1 = &*a8++
    ||*(a1+=[2]) = d2
```

The first step is to identify the required opcode format. The instruction in Example 8–64 contains a data unit operation in parallel with both a global transfer and a local transfer.

Three formats allow two parallel transfers to be specified. The instruction in Example 8–64 requires the double parallel format (described in row 1 of Table 8–38) because the global transfer is a load.

The **dst1** and **src1** columns under ALU operands indicate that the ALU destination and ALU data path source 1 must be D registers. The other data unit operands must always be D registers, regardless of the parallel operations performed in the instruction.

One exception is ALU data path source 2 that can be a 5-bit immediate instead of a D register for base set ALU operations. The **Cond** and **Status Protection** columns indicate that the operations cannot be conditional and that the ALU operation cannot specify status protection. These data unit restrictions are all met by the data unit operation in Example 8–64.

The global transfer column contains two subcolumns: one for moves, and one for a load, store, or address unit arithmetic. In this case, the move subcolumn is not applicable because a move cannot be specified using this format.

The load/store/address unit arithmetic subcolumn contains further subcolumns. The **S/D** column indicates that the address unit arithmetic destination in Example 8–64 must be a lower register. Since x1 is a lower register, this restriction is satisfied.

The **Index** column indicates that the index must be either an index register or a short offset; it cannot be a long offset. In Example 8–64, the global transfer postincrements by a scaled 1; therefore, a short offset is sufficient.

The **Rel** column indicates that PP-relative addressing cannot be used by the global transfer. These global transfer restrictions are all satisfied by the global transfer operation in Example 8–64.

The **Local Transfer** column also contains subcolumns for S/D, Index, and Rel. For the double parallel format, these indicate that the specified store in Example 8–64 must use a D register source and an index register or short offset, and it cannot specify PP-relative addressing.

Since the restrictions on the data unit operation, global transfer, and local transfer are all met, the instruction in Example 8–64 is legal.

8.10.6.2 Determining What Other Operations You Can Specify in Parallel

Table 8–38 can also be used to determine what operations can be specified in parallel with the global transfer shown in Example 8–65.

Example 8–65. Global Transfer

```
*(a8 + 0x400) = d1
```

This global transfer uses an offset of 0x400, which requires a long offset. Table 8–38 shows that the global (long offset) is the only format that supports a global transfer that uses a long offset.

For this format, the local transfer column is not applicable indicating that no parallel local transfer can be specified. The columns that describe the parallel data unit operation indicate that a parallel data unit operation cannot be conditional, cannot specify status protection, and must use only D register source and destination operands.

8.10.7 Programming Exercises

In order to write optimized PP code, it is important to utilize the full instruction word whenever possible. This requires the ability to look at an instruction and determine what (if any) other operations can be specified in parallel.

8.10.7.1 Exercise 1: Identifying What to Specify in Parallel

Which part of the instruction word is free in the following cases? Choose none, data unit, global transfer, or local transfer.

1) d1=[nz] d1 ^ d2

2) x1=[n] d1 + d2>>u 4

3) d2=d2 & d1 || a3 = &*(a4 + 0x300)

4) d2=d3 * d4 ||d6=d6 + d2 || d4= &*(a2 +=x0)

Refer to subsection 8.10.7.2 for the answers to this exercise.

8.10.7.2 Answers to Exercise 1

1) **Global transfer**. No local transfer can be specified in this instruction, because the data unit operation is conditional.

2) **None**. No global transfer can be specified, because the data unit operation uses a non-D register operand. No local transfer can be specified, because the data unit operation is conditional.

3) **None**. Because the local transfer uses a long offset, no global transfer can be specified. Also, since the destination of the load is a non-D register, you cannot specify a global transfer.

4) **Global transfer**. The specified data unit operations and local transfer do not require the global transfer portion of the opcode.

8.10.7.3 Exercise 2: Determining Whether an Instruction Is Legal

Using Table 8–38, determine whether the following instruction is a legal combination of parallel operations.

```
d7 = d5 | d3
    ||lrse0 = &*(2)
    ||d3 = *a0++
```

Refer to subsection 8.10.7.4 for the answer to Exercise 2.

8.10.7.4 Answer to Exercise 2

This operation requires the double parallel format because both a global transfer and a local transfer are specified and the global transfer is not a move or field move.

The columns related to the data unit operation specify that the data unit operands must be D registers. This is true for the data unit operation specified in Exercise 2.

Next, the cond and status protection columns indicate that the operation cannot be conditional or specify status protection. This is also true for the given data unit operation.

Next, the load/store/address unit arithmetic S/D column indicates that the address unit arithmetic destination must be a lower register. This condition is not met, because lrse0 is an upper register. Thus, this instruction is an illegal combination of parallel operations.

8.10.7.5 Exercise 3: Identifying Illegal Instructions

All of the following instructions are illegal because two operations require the same portion of the opcode to be specified. Which portion of the opcode is doubled up? Choose between none, data unit, local transfer, and global transfer.

1) a2 = %3 ||d2 = *a9++

2) d1 =[z] d1 + 1 ||*(a11 + 0x100) = d1

3) d1 = d1 + d2 ||d3 = d3\\2 ||d4 = &*(a7 + 0x7000)

Refer to subsection 8.10.7.6 for the answers to this exercise.

8.10.7.6 Answers to Exercise 3

1) **Global Transfer**. The non-D register dst operand used by the data unit operation requires the global transfer portion of the opcode. as does the explicit global transfer. The local transfer is free, however.

2) **Local Transfer**. The local transfer portion of the opcode is required by both the global transfer's long offset and the conditional information for the data unit operation. Only one of these can occur in a single instruction.

3) **Data Unit**. Both the add and the left rotate operation require the data unit. Only one of these can be specified in a single instruction.

8.11 Parallel Write Priority

The same D register can be specified as the destination of two or more parallel operations in a single instruction. In this situation, only one result is actually written to that D register. The result that is written depends on the write priority of the parallel operations and the status for conditional writes. The result of the highest priority operation is normally written to the D register. The results of all other operations specifying that D register destination are normally discarded. If the highest priority operation is conditional and the condition is false, the result of the next highest priority operation is written to the register. The write priority for parallel operations is:

1) **Global Transfer**

2) **Local Transfer**

3) **Data Unit**

   - If a multiply and an ALU operation specify the same destination, the ALU operation is not performed (and no status setting occurs).
   - For EALU||ROTATE, if ALU and rotate destinations are the same, the ALU result has priority.

This fixed write priority provides two significant features:

- First, write priority can be used to force the result of a lower priority operation to be discarded. This is especially useful for ALU operations performed simply for setting status. ALU operations are given a lower priority than both global and local transfers.
- Second, for conditional operations, write priority can be used to specify a single instruction something similar to an if-else statement.

This section discusses these two features of the PP parallel write priority. Table 8–39 lists the topics discussed in this section.

Table 8–39. Discussion of the PP Parallel Write Priority

Section	Topic
8.11.1	Discarding Results Using Fixed Write Priority
8.11.2	Using Conditionals and Fixed Write Priority (If-Else)

8.11.1 Discarding Results Using Fixed Write Priority

When an ALU operation is performed solely for indicating status, the write priority can be used to discard the result, thus eliminating the need to write to a register. If there is a parallel load or move to a D register in the instruction, the result of the ALU operation can be specified to the same destination, causing the result of the ALU operation to be discarded. If no parallel transfer is required, a move of ALU destination register to itself can be specified in parallel with the ALU operation to force the ALU result to be discarded.

Example 8–66. Compare Result Discard

```
d7  =  d6 – d5    ; Use same destination as
    ||d7 = *a8    ; parallel transfer to force
                  ; discard of ALU result.
d7  =  d6 – d5    ;
    ||d7 = d7     ; Parallel move to preserve contents
                  ; in d7.
```

Example 8–67. Discard Local Transfer Write

```
d3   =    *a8++        ; load d3.
  ||d3 = &*(a1–=[7])   ; Update a1 but discard write of
                       ; address unit arithmetic result
                       ; to d3.
```

Although you can discard a write by specifying one of the read-as-zero address registers as the destination of an operation, this imposes restrictions on the operations that can be specified in parallel because a7 and a15 are non-D registers. Therefore, using write priority is more efficient for discarding results than using the zero registers in most cases with the data unit and local address unit.

8.11.2 Using Conditionals and Fixed Write Priority (If-Else)

The fixed write priority can be used with a conditional global transfer to specify a single instruction similar to an if-else statement. Consider the pseudocode instruction shown in Example 8–68.

If cond is true, the global transfer to d7 occurs and the write of the ALU result is discarded (that is, it is not written to d7) because the global transfer write has a higher priority.

If cond is not true, the global transfer to the d7 register does not occur, and the write of the ALU result to d7 can take place. In other words, the global transfer writes to d7 if the condition is true; else if the condition is not true, the ALU operation writes to d7.

Example 8–68. If-Else Statement Using Write Priority

```
;*******************************************************
;    if [cond]
;         d7 = <global transfer>
;    else
;         d7 = <ALU Operation>
;*******************************************************
     d7 = <ALU Operation>
          ||d7 =[cond] <global transfer>
```

Note:

The analogy to an if-else statement is not entirely accurate, because the ALU operation and its associated status setting occur, regardless of the condition; it is the write of the result of the ALU operation that occurs only if the condition is not true. Also, for the global transfer operation, only the data transfer occurs conditionally; if specified, the memory access and optional update of the address register occur, regardless of the condition.

In Example 8–69, write priority is used to update a run length count of zeros (the variable zero_run). The first instruction sets status based on data (the variable). If the data is equal to 0, the zero status bit is set to 1; otherwise, the 0 status bit is set to zero.

The next instruction initializes zero_run to 0 if data is nonzero or increments zero_run by 1 if data is equal to 0. If the zero status bit is not set, the global transfer operation is performed, initializing the zero_run to 0. Because the write by the global transfer operation has priority over the write of the ALU, the result of the ALU operation is discarded (though status is still set by the ALU operation).

If the zero status bit is set, the global transfer is not performed, because the nz (not-zero) condition is false. This allows the ALU operation that increments zero_run by 1 to write its result back to zero_run. This sequence of operations can be placed in a loop to count runs of zeros, initializing the run when nonzeros are encountered.

Example 8–69. Run-Length Update Using Write Priority

```
data:            .set     d6
zero_run:        .set     d7
data = data                          ;     Test data.
zero_run = zero_run + 1              ;     else
                                     ;           increment run length.
    ||zero_run =[nz] a15             ;     If (not zero)
                                     ;           initialize run length to
                                     ;           zero.
```

8.12 Programming Restrictions

This section describes some constraints to keep in mind when programming a PP. Some constraints are a consequence of allowing numerous parallel operations to be independently performed on optimized hardware. This makes different restrictions (or capabilities) available, depending upon which parallel operations are specified.

❑ **Pipeline:**

- Branches and loop register loads always have two delay-slot instructions. For example, a modification of the pc or a loop register has no effect on the next two instructions fetched, since they will have already been fetched when the register is actually modified.
- There should always be a delay-slot instruction between an instruction that modifies/uses an address register or index register as the destination of an operation, and an instruction that uses that same address register or index register for address generation.

❑ **Opcode:**

- PP-relative addressing can be used when only one memory access or address unit arithmetic is specified.
- Registers allow a full 32-bit index, whereas immediate offsets have a more restricted range.
- Local transfers can access only D registers if there is a parallel global transfer or a parallel data unit operation with a non-D register operand.
- Only local transfers can be performed in parallel with data unit operations using non-D register operands.
- An ALU operation can specify a non-D register source and/or destination if no parallel global transfer is specified. All other sources and destinations must be immediate values or D registers.

  When a global transfer is specified in parallel with an ALU operation, the ALU operation cannot use a non-D register source or destination.

- The global port is limited to accessing only the lower registers when a parallel local access is specified.
- Similarly, the second source of a data unit operation with a 32-bit immediate operand must be one of the lower registers.
- Local transfers cannot be performed conditionally.
- Parallel conditional operations must be based on the same condition.
- If a conditional data unit operation or global transfer is specified, then no local transfer can be specified in the instruction.
- Multiply and divi sources and destinations must be D registers.

❑ **Other:**

- Only aligned addresses may be accessed. Nonaligned addresses are truncated to make them align. Note, however, that the full 32-bit address is written back to the address register if the addressing mode specifies address register update.
- For operations involving the bit detection logic (lmo, rmo, lmbc and rmbc), the ALU carry ripple is split at 8 bits. The input to the bit-detection logic and the destination of the ALU operation on the bit-detection result must be D registers.
- Conditional source selection based on negative for an ALU operation or a store performed by the global address unit must be between odd and even registers that form a D register pair. The D register pairs are: d7/d6, d5/d4, d3/d2 or d1/d0. If n is true, the odd register is selected; if n is false, the even register is selected.

Chapter 9

Interrupts and Reset

This chapter explains the PP interrupt sources and the state of the PP at reset.

Topics

9.1	Interrupt-Related Registers	PP: 9-2
9.2	Enabling and Disabling Interrupts	PP: 9-8
9.3	Interrupt Flags	PP: 9-10
9.4	Interrupt Priorities and Vector Addresses	PP: 9-13
9.5	Interrupt Processing	PP: 9-15
9.6	Reset	PP: 9-18

9.1 Interrupt-Related Registers

The PP architecture supports up to 31 interrupt sources. Nine maskable interrupt sources are currently implemented. These interrupts are used for interprocessor communication and for notification of packet transfer request status from the transfer controller. An enabled interrupt condition causes an interrupt service routine to be invoked. Disabled interrupts sources still set their corresponding interrupt flags, but the interrupt service routine is not performed. Polling disabled interrupt flags in software is a useful alternative to interrupt service routines when the required action in response to the interrupt condition does not have to be performed immediately.

Interprocessor commands (discussed in Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis*) are used by a PP to issue message interrupts to other on-chip processors (including itself). The following sections describe interrupt-related registers, global enabling and disabling of interrupts, software clearing and setting of interrupt flags, interrupt vector addresses and priority, interrupt service routines, interprocessor commands, and reset.

These PP registers contain bits related to interrupts:

- **pc register**

  When equal to 0, bit 1 of the pc register, referred to as the G bit, globally disables interrupts. This bit must be set to 1 (using eint) before individual interrupts enabled with the inten register will invoke their interrupt service routines.

- **inten register**

  The inten register is the interrupt-enable register. It contains one enable bit for each interrupt source.

- **intflg register**

  The intflg register is the interrupt-flag register. It contains one flag bit for each interrupt source.

The remainder of this section describes the bits in these registers.

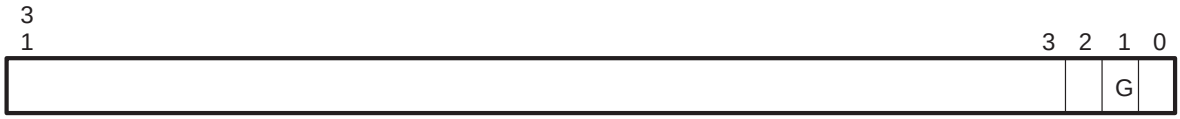
9.1.1 The pc Register

A global interrupt enable bit (G) is located in bit 1 of the pc register.

- When G = 0, interrupt sources are globally disabled, regardless of the values in the inten register. Even though interrupt servicing is disabled, the interrupt flags are still set in the intflg register and can be polled by software.
- When G = 1, interrupt sources are individually enabled according to the contents of the inten register.

As described in Section 9.2, *Enabling and Disabling Interrupts*, the G bit is set to 1 by an eint operation and cleared to 0 by a dint operation. The G bit is not modified by branches or calls performed by data unit operations or by moves to the pc register (br or call). Loads to the pc register (br or call) modify the G bit only if the address is generated using an unscaled immediate-index with bit 1 not equal to 0.

Figure 9–1. pc Register: Global Interrupt Enable Bit



Note: For a complete description of the pc register, see Chapter 7, *Summary of PP Registers*.

9.1.2 The inten Register

The inten register allows selective enabling or disabling of interrupts.

- **E—Enable bits**

  You can enable each interrupt source individually by setting the corresponding enable (E) bit of the inten register to 1. Even when an enable bit for a corresponding interrupt is 0, the corresponding bit in intflg is set when the interrupt condition occurs. However, the interrupt is disabled in the sense that the interrupt is not serviced.

- **W—Write-mode bit**

  Bit 0 of the inten register is used to modify the behavior of software writes to the intflg register as follows:

  - When the W bit = 0, each bit of the intflg register can be cleared to 0 by writing a 1 to it. Writing a 0 to a bit in the intflg register has no effect. This is the normal setting for the W bit.
  - When the W bit = 1, each bit of the intflg register can be set to 1 by writing a 1 to it. Writing a 0 to a bit in the intflg register has no effect. The W bit is usually set to 1 only when you are restoring a task and you want to return the intflg register to its previous state. For details, see subsection 9.3.2, *Saving and Restoring an Interrupt State*.

Figure 9–2. inten and intflg Registers

inten

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r	r	r	r	E	E	E	E	–	–	–	E	E	E	E	–	–	E	–	–	–	–	–	–	–	–	–	–	–	–	–	W

intflg

r	r	r	r	I	I	I	I	–	–	–	I	I	I	I	–	–	I	–	–	–	–	–	–	–	–	–	–	–	–	–	–
				PP3MSG	PP2MSG	PP1MSG	PP0MSG				MPMSG	PTEND	PTERR	PTQ			TASK														

lowest ◄ priority highest

Key:

- E Interrupt enable bit
- I Interrupt request bit
- r Reserved bit for additional MSG interrupts for an MVP with eight PPs
- – Reserved
- W Set or clear mode for writes to intflg

9.1.3 The intflg Register

Pending interrupts are recorded in the intflg register. The intflg register latches interrupt requests until they are specifically cleared by software. The interrupt source priority is indicated from highest to lowest by the right to left order of the associated bits in the intflg register.

Note:

The reserved interrupt flags will produce an interrupt if they are enabled and set to 1. Thus, software should set these reserved bits to 0 after a hardware reset. On current generations of the MVP, if compatibility with future versions of the MVP is desired, these reserved bits **should never be set to 1** after they have been initialized to 0, because future versions may define and use those bits.

- **TASK Interrupt**

  The TASK interrupt is issued by an MP interprocessor command (discussed in Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis*) and causes the PP to load its TASK interrupt vector. Typically, this interrupt is used to make the PP switch tasks under MP control. Nothing prevents this interrupt from being used in other ways. However, for consistency with TI's software libraries, this interrupt should always be used as a TASK interrupt.

- **PTQ—Packet Transfer Queued Interrupt**

  The PTQ interrupt flag is set to 1 if a packet transfer is requested (by a software write to the P bit of the comm register) when a previous packet transfer request is still queued. This allows a packet transfer request that is linked automatically by the next-entry parameter of a previous packet transfer to be submitted without checking that the previous transfer has finished. If the previous PT is still queued, the PTQ interrupt flag will be set to 1. For more details on the PTQ interrupt and a sample PTQ interrupt service routine, see subsection 12.3.2.1, *Submitting a Packet Transfer Request While a Previous Request is Still Queued*.

  Note that you should not implement a software write to a packet transfer request linked-list pointer without first making sure no packet transfer request is already queued.

❑ PTERR—Packet Transfer Error Interrupt

The PTERR interrupt flag is set to 1 if the transfer controller encounters an error condition while executing a packet transfer requested by the PP. These error conditions are defined in Section NO TAG, *Packet Transfer Errors*, in the *MVP Transfer Controller User's Guide.* Two sample error conditions are:

- The packet transfer parameters are not aligned to a 64-byte boundary.
- The packet transfer linked-list pointer isn't pointing to an on-chip address.

❑ PTEND—Packet Transfer End Interrupt

The PTEND flag is set by the TC when it encounters the end of the PP's packet transfer linked list, or when it completes a packet transfer within a linked list that instructs the TC to interrupt the requesting processor when the transfer has been completed. See subsection 12.3.1.2, *PTEND Flag Polling*, and subsection 12.3.2.2, *TC Interrupt of PP Upon Packet Transfer Completion*, respectively, for examples of software polling of the interrupt bit and PTEND interrupt servicing.

❑ MPMSG—MP Message Interrupt

The MPMSG interrupt flag is set to 1 when the MP sends a message interrupt to the PP. This interrupt is typically used by the MP to notify a PP that it has deposited data or a pointer to data that needs to be processed in the PP's local RAM. For further details, see Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis*.

❑ PPnMSG—PPn Message Interrupt

The PPnMSG flag (where n can be 0, 1, 2, or 3 in the TMS320C80) is set when PPn sends a message interrupt to the PP. The PPnMSG interrupts can be used to implement data-passing protocols between processors. For example, if the output data from PP0 serves as the input data for PP1, PP1 must know when its input data has been made available by PP0 for processing. Likewise, PP0 must know when PP1 is ready to receive PP0's output data (to avoid overwriting data that has not been processed yet by PP1).

Note than PPn can send a message interrupt to itself. See Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis* for details on issuing a message interrupt.

9.2 Enabling and Disabling Interrupts

After a hardware reset, inten and intflg are in an unknown state. Thus, they must be initialized before interrupts are enabled. All reserved bits in the intflg register should be set to 0 because they will generate an interrupt if they are enabled and set to 1.

For an interrupt source to be enabled, the global interrupt bit must be set and the corresponding bit in the inten register must also be set. The G bit is set and cleared with the eint and dint instructions, respectively.

9.2.1 Pipeline Considerations

The dint and eint operations become effective before the succeeding instruction is fetched. Thus:

- Interrupts are recognized between an eint instruction and the following instruction.
- Interrupts are not recognized between a dint instruction and the following instruction. Following a dint instruction, interrupts will not be recognized again until they are re-enabled with by an eint instruction.

For more information, see Section 8.7, *Miscellaneous Operations.*

Note:

The instruction that modifies intflg or inten executes in the execute pipeline stage; eint and dint execute immediately upon being fetched. Thus, after modifying intflg or inten, one delay slot instruction should be inserted before interrupts are enabled, as shown in Example 9–1. Otherwise, interrupts will be enabled before the desired contents are in inten and intflg.

Example 9–1. Interrupt Enable Sequence

```
inten = 1\\14        ;  Enable only task interrupt
                     ;  (assumes W set to 0).
intflg = intflg      ;  Clear interrupt flags.
nop                  ;  Delay slot between clearing
                     ;  intflg and globally enabling
                     ;  interrupts.
eint                 ;  Globally enable interrupts.
```

9.3 Interrupt Flags

When the W bit of the inten register is set to a 1, software can set intflg bits only to 1. Writing 0s to bits in the intflg register has no effect. This mode is used for restoring a saved interrupt state.

When the W bit of inten is a 0, writing a 1 to an intflg bit will cause it to be cleared to 0. Writing a 0 has no effect. This allows individual interrupt flags to be cleared without disturbing the state of others.

9.3.1 Clearing Interrupt Flags

An interrupt service routine should clear its associated interrupt flag because this is not done automatically in hardware. If a particular interrupt source is trying to set a flag at the same time as software is trying to clear it, the flag will remain a 1.

In order to clear interrupt flags, the W bit of the inten register must be 0. Example 9–2 shows how to clear all interrupt flags (by moving the value in the intflg register to itself), clear an individual flag, and clear the message interrupt flags (top eight bits of the intflg register).

Example 9–2. Interrupt Flag Clearing, W = 0

```
inten = inten&~%1      ;  Set W=0.
intflg = intflg        ;  Clear all interrupt flags.
                       ;  (1s written to all set flags)
intflg = 0x1\\14       ;  Clear task interrupt flag.
intflg = ~%24          ;  Clear top 8 intflg bits.
```

9.3.2 Saving and Restoring an Interrupt State

The interrupt state can be saved when a new task must be executed on the PP; the original state can be restored after the task has finished. The write mode controlled by the W bit allows this to be done without causing missed interrupts during the saving or restoration.

A simple code example for saving and restoring the interrupt state is shown in Example 9–3. A variety of similar procedures can be produced if only certain interrupt flags must be saved and restored.

Example 9–3.Saving and Restoring Interrupt Status

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   Save Interrupt Status Before switching
;;        to another task.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
dint                ;    Disable interrupts.
*––sp = inten       ;    Push inten on Stack.
*––sp = intflg      ;    Push intflg on Stack.
intflg = intflg     ;    Clear all intflg bits.
                    ;    Interrupts coinciding with the
                    ;    execution of this instruction will
                    ;    be set and should be associated
                    ;    with the new task.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   Begin new task.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
<Instruction>       ;    Execute New Task
     .              ;
     .              ;
<Instruction>       ;    The task should clear its own
                    ;    interrupt flags before returning
                    ;    the previous task.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   Restore Previous Interrupt Status
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
dint                ;    Disable interrupts.
inten = 0x1         ;    Set W bit to 1.
intflg = *sp++      ;    Pop old intflg.
inten = *sp++       ;    Pop old inten.
nop                 ;    Delay slot before enabling
                    ;    interrupts.
eint                ;    Enable interrupts.
```

9.4 Interrupt Priorities and Vector Addresses

When an enabled interrupt flag is set, the corresponding PP interrupt vector is fetched from the PP's local parameter RAM at the addresses shown in Figure 9–3. The logic that generates the interrupt vector address includes the PP number in order to generate unique addresses for each PP. Note that the interrupt priority increases toward lower addresses.

The two LSBs of the interrupt vector determine the state of the global interrupt enable (G) bit and loop inhibit (L) bit upon entry to the interrupt service routine. These two control bits located in the LSBs of the pc register are set to the value of the corresponding bits in the interrupt vector upon entry to an interrupt.

Figure 9–3. Interrupt Vector Locations

Priority	32 Bits	Address (# = PP Number)
Lowest	Reserved for PP7 Message	0x0100 #1FC
	Reserved for PP6 Message	0x0100 #1F8
	Reserved for PP5 Message	0x0100 #1F4
	Reserved for PP4 Message	0x0100 #1F0
	PP3 Message	0x0100 #1EC
	PP2 Message	0x0100 #1E8
	PP1 Message	0x0100 #1E4
	PP0 Message	0x0100 #1E0
	Reserved	0x0100 #1DC
	Reserved	0x0100 #1D8
	Reserved	0x0100 #1D4
	Master Processor Message	0x0100 #1D0
	Packet Transfer End	0x0100 #1CC
	Packet Transfer Error	0x0100 #1C8
	Packet Transfer Queued	0x0100 #1C4
	Reserved	0x0100 #1C0
	Reserved	0x0100 #1BC
	Task Interrupt	0x0100 #1B8
	Reserved	0x0100 #1B4
	Reserved	0x0100 #1B0
	Reserved	0x0100 #1AC
	Reserved	0x0100 #1A8
	Reserved	0x0100 #1A4
	Reserved	0x0100 #1A0
	Reserved	0x0100 #19C
	Reserved	0x0100 #198
	Reserved	0x0100 #194
	Reserved	0x0100 #190
	Reserved	0x0100 #18C
	Reserved	0x0100 #188
	Reserved	0x0100 #184
Highest	Reserved	0x0100 #180

While not enforced by hardware, software convention requires that at the start of an interrupt service routine, interrupts should be globally disabled (G = 0) and looping should not be inhibited (L = 0). In order to satisfy this software convention, the two LSBs of an interrupt vector should always be 0.

Example 9–4 illustrates how to set up the MP message interrupt vector to the service routine MP_MSG_INT.

Example 9–4. Setting up an Interrupt Vector

```
      d7 = MP_MSG_INT
      *(pba + 0x1d0) = d7
       .
       .
       .
MP_MSG_INT:
       .
       .
       .
```

9.5 Interrupt Processing

This section describes what occurs upon entry to an interrupt service routine (ISR) and provides some guidelines for ISRs, including the return from interrupt sequence.

When an interrupt has been requested but has not yet been processed, it is called a pending interrupt. If a pending interrupt is enabled (and no interrupt with a higher priority is also pending), it is accepted on the following cycle. When an interrupt is taken, a sequence of five pseudoinstructions, referred to as PS1–PS5, is injected into the instruction pipeline.

As shown in Figure 9–4, these pseudoinstructions push the pc, ipa, and ipe values and the correct G and L bits required by the interrupt return sequence onto the stack. Additionally, the status register contents are pushed onto the stack so that status can be restored upon return from the interrupt. PS3 loads the PC at its branch address with the interrupt vector corresponding to the interrupt condition being serviced. The two LSBs of the interrupt vector should be 0 to clear the G and L bits (global interrupts disabled and looping not inhibited) in the pc register before the start of the ISR. Execution of the interrupt service routine begins immediately following the completion of PS5.

Figure 9–4. Stack Condition after PS1–PS5

				addr	← Stack Pointer Before Interrupt
PC Values for Return From Interrupt	x	x	0	addr – 4	
IPA Value for Return From Interrupt	x	x	1	addr – 8	
IPE Value for Return From Interrupt	x	G	1	addr – 12	
sr Contents at Time of Interrupt				addr – 16	← Stack Pointer Upon Entry to ISR

9.5.1 Interrupt Service Routine Guidelines

The following are some general guidelines for interrupt service routines (ISRs). Specific ISR code is shown in various sections throughout this user's guide.

- Registers that are used in an interrupt service routine (other than the pc, ipe, ipa, and sr registers, which are pushed onto the stack during the five pseudoinstructions executed automatically during entry to an interrupt) should be explicitly pushed onto the stack before being used and popped off before returning from the interrupt.

  It is sometimes advantageous to distribute the necessary pushes and pops in parallel with other operations in the ISR, when possible, instead of performing them all directly upon entry to or exit from the ISR.

- The interrupt flags are not cleared by hardware. Thus, it is generally recommended that they be cleared by the interrupt service routine.

- The iprs register is not saved by the pseudoinstructions upon entering an interrupt. If an interrupt service routine performs any calls, then iprs should first be pushed onto the stack by the interrupt routine and restored before returning.

- The stack pointer should be returned to its location at the start of the ISR before the return from interrupt sequence.

- It is recommended that ISRs that explicitly enable interrupts (using eint), disable them (using dint) before executing the return from ISR sequence.

- The final four instructions must contain the following four global transfer return from interrupt (reti) operations:

```
reti1
reti2
reti3
reti4
```

9.5.2 Returning From an Interrupt Service Routine

The reti operations restore the status register contents and then pop the stacked values of ipe, ipa, and pc into the branch register code for the pc register (br) in three consecutive instructions. These operations effectively replay the PC history calculated on entry to the interrupt, but the fetches are performed and any cache-misses are serviced. The global interrupt enable bit is restored by reti2 to its value upon entry to the interrupt. The loop inhibit bit is set during all but the last reti operation (reti4) because the PC history saved on the stack (as shown in Figure 9–4) already accounts for looping.

Other operations that do not utilize the global port can be coded in parallel with the reti operations as long as they do not modify the status register.

9.6 Reset

A PP can be reset by either hardware or software (interprocessor command) as follows:

- **Hardware reset**

  Assert a low level on the MVP $\overline{\text{RESET}}$ pin. When hardware reset is asserted, the transfer controller receives a reset signal. The TC then issues reset acknowledges to all the PPs and the MP. The reset acknowledges remain active as long as the $\overline{\text{RESET}}$ pin is active. To guarantee correct recognition of a hardware reset, the $\overline{\text{RESET}}$ pin must be active for at least four MVP clock cycles.

- **Software reset**

  The MP can reset a PP by either sending a reset PP command (R bit set and individual PP designator bit(s) set in the command word) or sending a reset TC command (r bit set and the TC designator bit set in the command word). When the TC receives a reset command from the MP, it resets itself and sends reset acknowledges to all the other processors, causing them to be reset also. For details on interprocessor commands, see Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis*.

9.6.1 Activity During Reset

All resets are managed by the TC. If a PP receives a reset command (submitted by the MP), the PP immediately sets its halt bit and sends a reset request to the TC. The PP is not actually reset until the TC returns a reset acknowledge to the PP. Before returning a reset acknowledge, the TC resets its state information relating to cache services, DEA services, and packet transfer requests for the PP being reset.

These cache services, DEA services, or packet transfer services are abandoned, and all transfers relating to them will cease, once the pipeline has drained. TC activity for the nonresetting processors is not abandoned but is drained from the pipeline temporarily for reordering.

Between sending a reset request to the TC and receiving back a reset acknowledge, the PP clears the halted latch visible to the MP in its PPERROR register. The halted latch goes active only when the reset acknowledge has been received. This allows the MP to recognize when reset has completed. When the MP sends a reset command to a PP, it should wait for the halted signal to go active before unhalting that PP.

Once the PP has received a reset acknowledge from the TC, the following operations are performed:

- The halt latch is set (if it is not already set).
- The H, P, Q, and S packet-transfer request handshake bits in the comm register are cleared to 0.
- The address units cancel any pending accesses.
- Cache requests are canceled.
- The three loop enable (E) bits in the lctl register are cleared.
- Two pseudoinstructions to be executed when the PP is unhalted that save the stack pointer (shifted left by 1 bit) are inserted into the pipeline. Then the interrupt state machine is set so that interrupt pseudoinstructions PS3–PS5 are loaded with the task interrupt vector as the interrupt source.

The above reset sequence is utilized by both hardware and software resets. For a software reset, the above mechanism preserves a maximum amount of program status to allow code debug.

9.6.2 Initial State Following Reset

When reset is completed, the PP remains in a halted state until the MP issues an unhalt command to the PP. When the PP is unhalted after reset, it executes the two pseudoinstructions that are forced into the pipeline by reset. Next, the PP runs through the PS3 to PS5 pseudoinstruction sequence for a task interrupt.

The pc is loaded with the address contained in the task interrupt vector pointer. Note that it is typically the MP's job to store the correct value in the task interrupt vector.

These are the result of the five instructions following reset:

- The previous program status information shown in Figure 9–5 is placed at the top of parameter RAM.
- The stack pointer (sp) is set to 0x0100 #7F0, just after the previous program's status information.
- Code execution begins at the address specified by the task interrupt vector.

Note:

The two LSBs of the task interrupt vector should be 0 in order to set the G bit to 0 so interrupts are globally disabled and set the L bit to 0 so looping is not inhibited.

Figure 9–5. Stack Condition Following Software Reset

Left-shifted sp Value From Before reset	–	G	L	0x0100 #7FC
IPA Value From Before Reset	–	G	L	0x0100 #7F8
IPE Value From Before Reset	–	G	L	0x0100 #7F4
				0x0100 #7F0 ← Stack Pointer

= PP Number

9.6.3 Activity Following Reset

Software reset can be used to gain control of a PP when it has encountered an error condition or refuses to respond to an MP task interrupt. Possible situations requiring a software reset include:

- A PP refuses to respond to an MP task interrupt.
- A PP has fetched an illegal opcode.
- A PP submitted a cache request or DEA that faulted.

An instruction-cache reset is performed by reset only when specifically requested in the command word. Software reset preserves the PP state information. The only pieces of program status lost to debug are the current PC value and the lctl E bits. The PC value is typically not important, because it corresponds to an instruction fetched after the occurrence of the error condition that is being investigated.

Chapter 10

Understanding Branches and Calls

This chapter discusses branches and calls and how they differ.

Topics

10.1	Software Branching	PP: 10-2
10.2	Understanding the Difference Between Branches and Calls	PP: 10-3
10.3	Branch Delay Slots	PP: 10-4
10.4	Subroutine Call and Return Sequence	PP: 10-8
10.5	Absolute Versus Relative Branching	PP: 10-11
10.6	Conditional Branches and Calls	PP: 10-13
10.7	Branch Tables	PP: 10-15
10.8	Two-Input ALU Operations With pc Destination	PP: 10-17

10.1 Software Branching

Software branching is performed on the PP by specifying one of the pc register write codes (br or call) as the destination of an ALU operation, load, or move. ALU operations with the pc register as the destination can involve any Boolean or arithmetic combination of two inputs. However, branches are typically specified in one of the following ways:

- An absolute branch

  ```
  br = address
  ```

- A relative branch

  ```
  br = ipe + offset
  ```

- A load from the branch table

  ```
  br = *BranchTbl.element
  ```

- A move from a register

  ```
  br = iprs
  ```

The following sections discuss issues related to software branching.

10.2 Understanding the Difference Between Branches and Calls

The PP provides two different register codes for the pc register:

- Register code **call** is used for making subroutine calls (a branch with a subsequent return).
- Register code **br** is used for branches that do not have an associated return.

When **call** is specified as the destination of an operation, two actions occur:

- The subroutine start address is written to the pc register, and
- The subroutine return address is written to the iprs (instruction pointer, return from subroutine) register.

The standard call and return sequence using iprs is detailed in Section 10.4.

When **br** is specified as the destination of an operation, only the write to the pc register occurs; the iprs register is not modified. In all other regards (for example, delay slots, etc.), calls and branches behave the same. In the following sections, discussions concerning branches also apply to calls unless a specific distinction is noted.

10.3 Branch Delay Slots

After an instruction that specifies a branch or a call is fetched, two more instructions are fetched (this is normal pipeline behavior) before the branch is executed. These two instructions are referred to as **delay slot** instructions. The PP (unlike the MVP's master processor) does not provide an annul option for these branch delay slot instructions.

Typically, the branch delay slot instructions will be the next two instructions in memory. However, this may not be the case if the branch occupies a branch delay slot for a previous branch or is a loop end address.

Example 10–1 illustrates the delay slot instructions for the branch to LABEL, assuming that the branch does not occur in the delay slot of a branch and that neither the branch nor its first delay slot instruction are a loop end. Thus, the next two instructions (<Instruction1> and <Instruction2>) are the delay slot instructions.

Branch delay slot instructions can perform useful operations. When the branch must be taken before more useful work can be done, you can fill the two delay slots with nop instructions.

Example 10–1. Branch Delay Slot Instructions

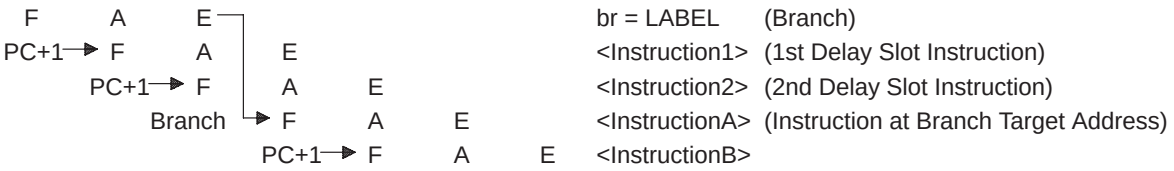
```
   br = LABEL
   <Instruction1>  ;  Delay Slot1 Instruction
   <Instruction2>  ;  Delay Slot2 Instruction
   .
   .
   .

LABEL: <InstructionA>
       <InstructionB>
   .
   .
```

The program flow for Example 10–1 is illustrated in Figure 10–1. This figure shows the overlap between pipeline stages (FAE) of the branch and its delay slot instructions. Progression from left to right corresponds to pipeline stage advances. Progression from top to bottom corresponds to the instruction sequence that is fetched and executed. PC+1→F indicates that the fetched instruction is determined by an increment of the program counter from the previous instruction. This is the case for the two branch delay slot instructions.

Direct writes to the pc register are prioritized over the program counter increment. Thus, when the branch is executed, the pc is set to the target address for the branch.

The arrow from the execute stage of the branch instruction to the next fetch stage indicates that the next instruction is determined by the branch (instead of the program counter increment). Thus, <InstructionA> is the next instruction fetched after the branch is executed (that is, after the two branch delay slot instructions). After the branch occurs, the program counter resumes incrementing, and <InstructionB> is fetched.

Figure 10–1. Software Branch Program Flow



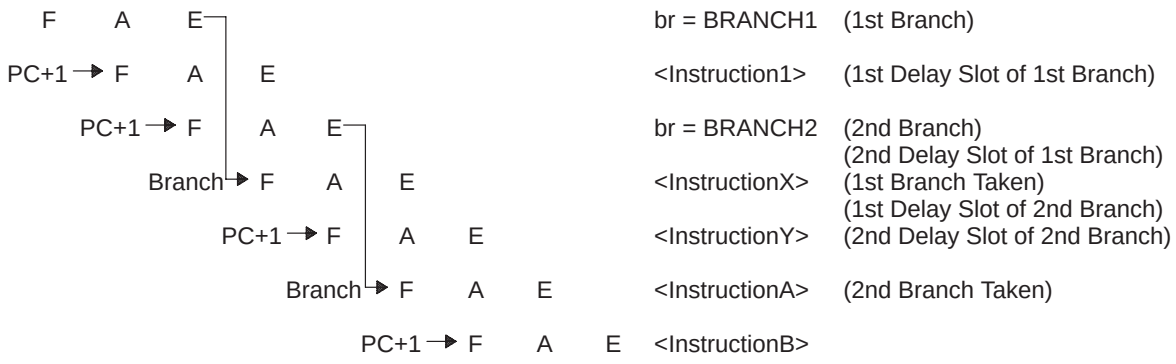
If a branch is specified in the first or second delay slot instruction of a branch, the first branch is taken and 1 or 2 instructions, respectively, are fetched at the target address for the first branch before the second branch target instruction is fetched. In Example 10–2, a branch is specified in the second delay slot instruction of a branch.

Example 10–2. A Branch in the Second Delay Slot of a Branch

```
   br = BRANCH1
   <Instruction1>
   br = BRANCH2
   <Instruction2>
   <Instruction3>
          .
          .
          .

BRANCH1:
   <InstructionX>
   <InstructionY>
          .
          .
          .

BRANCH2:
   <InstructionA>
   <InstructionB>
          .
          .
          .
```

The program flow for Example 10–2 is illustrated in Figure 10–2. Note that the delay slot instructions for a branch are not always the next two instructions in memory. As shown in Figure 10–2, the delay slot instructions for the second branch are the first two instructions at the target address of the first branch (<InstructionX> and <InstructionY>)—not the next two instructions in memory after the instruction specifying the second branch (<Instruction2> and <Instruction3>).

Figure 10–2. Branch in Delay Slot of a Branch



10.4 Subroutine Call and Return Sequence

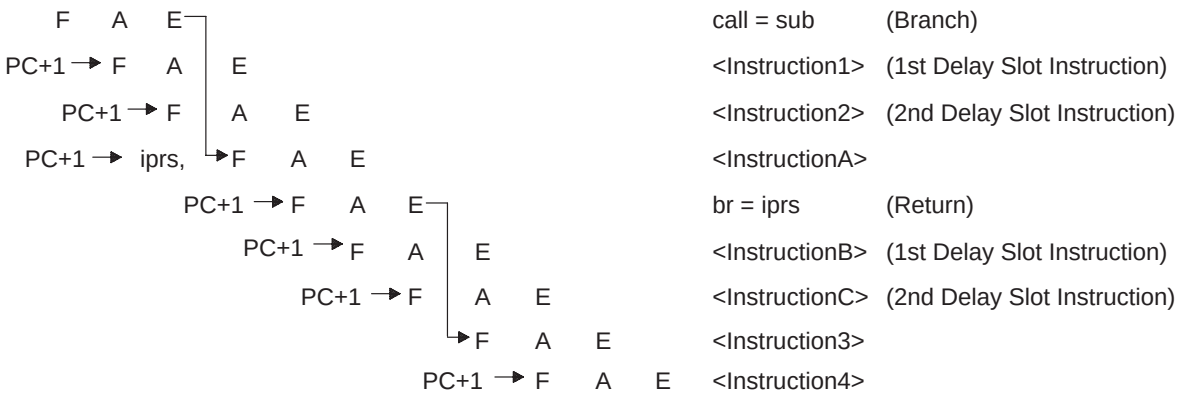
A subroutine call is performed by writing the subroutine address (typically specified by a label) to the call register code for the pc. When a call is executed, not only is the subroutine address written to the pc register, but also the return from subroutine address is written to the iprs register. The return address is determined by PC+1 (the instruction sequentially following the second delay slot instruction) at the time the branch is executed. The return address written to iprs is PC+1, even if the next instruction fetched (had the branch not occurred) would have been different (for example, the second delay slot instruction was a loop end).

Example 10–3 illustrates a call and return sequence. The resulting program flow shown in Figure 10–3 assumes that the call is not in a branch delay slot and that neither the call nor its first delay slot instruction are a loop end. At the time the call is executed, iprs is loaded with the current PC+1. This is the address for <Instruction3>. The return from subroutine is performed by moving the return address contained in the iprs register to br (the branch register code for the pc).

Example 10–3. Call and Return Sequence

```
      call = sub          ;  pc = subroutine address
                          ;     & iprs = return address
      <Instruction1>      ;  1st delay slot instruction.
      <Instruction2>      ;  2nd delay slot instruction.
back: <Instruction3>      ;  Instruction fetched upon
                          ;     returning from the
                          ;     subroutine.
      <Instruction4>      ;
          .
          .

sub:  <InstructionA>      ;  Start of Subroutine.
      br = iprs           ;  Return to back
      <InstructionB>      ;  Return delay slot1
      <InstructionC>      ;  Return delay slot2
```

Figure 10–3. Software Call and Return Program Flow



As shown in Example 10–4, a typical use for call delay slot instructions is to to push registers onto the stack. Likewise, return delay slot instructions can be used to pop these registers off the stack.

Example 10–4. Use of Delay Slot Instructions

```
        call = sub      ;  pc = subroutine address
                        ;  & iprs = return address
        *--sp = d7      ;  1st delay slot instruction.
        *--sp = d6      ;  2nd delay slot instruction.
back:   <Instruction3>  ;  Instruction fetched upon
                        ;  returning from the subroutine.
        <Instruction4>  ;
            .
            .

sub:    <InstructionA>  ;  Start of Subroutine.
            .
            .
            .
        br = iprs       ; Return to back
        d6 = *sp++      ; Return delay slot1
        d7 = *sp++      ; Return delay slot2
```

10.4.1 Nested Calls

When you implement a call within a call, you must save the iprs value of the first call before making the second call; this avoids overwriting the first call's return address.

In Example 10–5, the following events occur to ensure that the iprs value is saved:

1) **main** calls **sub1**, causing the return address to be written to the iprs register.
2) **sub1**, in turn, pushes iprs onto the stack before performing a call to **sub2**.
3) When the call to **sub2** is executed, the contents of iprs are overwritten with the return address from **sub2**.
4) The return from **sub2** is performed by moving iprs to br (the branch register code for the pc register).
5) The return from **sub1** to **main** is performed by popping the stacked return address directly to br. It is not necessary to pop this value to iprs first.

Example 10–5. Saving the iprs Value Before a Nested Call

```
main:     .
          .
      call = sub1     ;  Return address for
                      ;  sub1 written to iprs.
      <Instruction1>  ;  1st Delay Slot Instruction
      <Instruction2>  ;  2nd Delay Slot Instruction
sub1_ret:
      <Instruction3>
          .
          .
sub1: <InstructionA>
          .
          .
      *--sp = iprs    ;  Push return from sub1
                      ;  address onto stack.
      call = sub2     ;  Return address for sub2
                      ;  overwrites return address
                      ;  for sub1 in iprs.
      <InstructionB>  ;  1st Delay Slot Instruction
      <InstructionC>  ;  2nd Delay Slot Instruction
sub2_ret:
      <InstructionD>
          .
          .
      br = *sp++      ;  Pop sub1 return.
      <InstructionE>  ;  1st Delay Slot Instruction
      <InstructionF>  ;  2nd Delay Slot Instruction

sub2:     .
          .
      br = iprs       ;  Return to sub2.
      <InstructionY>  ;  1st Delay Slot Instruction
      <InstructionZ>  ;  2nd Delay Slot Instruction
```

10.5 Absolute Versus Relative Branching

Branches (and calls) can be specified in either absolute or relative fashion, as described in the following sections.

10.5.1 Absolute Branch

To perform an absolute branch, write the desired new PC address to the pc register. This is typically done by using a label that is resolved at link time.

Example 10–6. Absolute Call

```
main:     .
          .
      call = sublabel
      <Instruction1>     ;  delay slot1
      <Instruction2>     ;  delay slot2
return:   .
          .

sublabel: .
          .
      br = iprs
      <InstructionA>     ;  delay slot1
      <InstructionB>     ;  delay slot2
```

10.5.2 Relative Branches

Relative branches specify an offset from the current program counter location. On the PP, relative branches should be specified relative to the ipe (instruction pointer, execute stage) register. This is because the ipe register value at the time a call or branch is executed corresponds to the value of the PC at the time the instruction specifying the branch was fetched. The pc register cannot be reliably used to call or branch from because a call, branch, loop, or interrupt may have modified the value of the PC to something other than the desired relative branch base.

Note:

The three LSBs of any immediate or register value added to the ipe register should always be 0 to prevent undesirable carries from the G and L control bits (located in the two MSBs of the pc register) into the PC value.

In Example 10–7, a relative subroutine call is performed. The $ symbol is equivalent to a label for the instruction being executed. Assuming **sublabel** is within the current section of the current file, the assembler can evaluate the expression (**sublabel – $**) and produce the correct displacement between the address of the call instruction being executed and the address of the desired subroutine. This value is then added at runtime to the ipe register, which contains the address of the call instruction being executed.

When the branch offset is +/– 3 instructions, using a relative branch makes it possible to specify the branch with a 5-bit immediate. This, in turn, lets you specify a local transfer in parallel with the branch.

Example 10–7. Relative Call

```
main:       .
            .
        call = ipe + (sublabel – $)
        <Instruction1>        ;  delay slot1
        <Instruction2>        ;  delay slot2
return:
        <Instruction3>
            .
            .

sublabel:   .
            .
        br = iprs
        <InstructionA>        ;  delay slot1
        <InstructionB>        ;  delay slot2
```

10.6 Conditional Branches and Calls

Branches and calls can be performed conditionally just like any other ALU operation, move, or load. Thus, branches can occur conditionally, as specified by any of the 16 supported condition codes (see Table 8–36, *Condition Codes*).

Example 10–8 illustrates how a conditional branch is used to poll the comm register's Q bit until it is 0. As long as the Q bit is nonzero (indicating that a packet transfer request is queued), a branch back to the start of the poll sequence (poll) is performed. When the Q bit becomes 0, the conditional branch is not taken, and therefore, the poll loop is exited. At that point, the comm register's P bit can be set to 1 to submit a new packet transfer request (see subsection 12.2.3, *Step 3: Issuing a Packet Transfer Request to the TC*, for more information about setting the P bit).

ALU operations that write to the pc register do not modify status. This special case for ALU status saving prevents status from being corrupted by branches or calls to routines that require the status information. Also, different branch target addresses can be specified on the basis of different conditions.

Example 10–8. Conditional Branch

```
       a15 = comm & 0x1\\29  ;  Test Q bit.

poll:  br =[nz] poll         ;  Continue polling Q as long
                             ;   as Q=1 (Zero status bit
                             ;   = 0).
       nop

       a15 = comm & 0x1\\29  ;  Test Q bit.

       comm =comm | 0x1\\28  ;  Submit PTR (when poll loop
                             ;   is exited).
```

The code in Example 10–8 can be compacted by moving the instruction that sets the P bit into the polling loop. In Example 10–9, the first delay slot instruction of the conditional branch conditionally sets the P bit according to the opposite condition of the branch (that is, if the Q bit is 0). Because the branch operation does not modify status, the zero status bit is correctly maintained with respect to the previous test of the Q bit (see subsection 12.3.1.1, *Q Bit Polling*, for more information about polling the Q bit).

The two conditional operations in Example 10–9 are thus mutually exclusive; either the branch is performed, indicating that a packet transfer request is still queued, or the conditional branch is not taken and a new packet transfer request is submitted.

Example 10–9. Conditional Branch

```
       a15 = comm & 0x1\\29     ;  Test Q bit.

poll:  br =[nz] ipe             ;  Continue polling Q as
                                ;   long as Q = 1 (Zero
                                ;   status bit = 0).
       comm =[z] comm | 0x1\\28 ;  Submit PTR if Q=0 (Zero
                                ;   status bit = 1).
       a15 = comm & 0x1\\29     ;  Test Q bit.
```

10.7 Branch Tables

Branches specified by an ALU operation require a 32-bit immediate source operand (unless the branch is +/– 3 instructions). This precludes any parallel transfers from being specified in the instruction. If a branch to a certain location is performed frequently, such as a conditional branch inside a loop, an efficient technique is to determine the address for the desired branch and store the 32-bit address in a branch table located in the shared RAMs.

In Example 10–10, a branch table is set up in the PP's local parameter RAM. Two operations are required initially to set up each entry in the branch table. Then, each time a branch or call is required to one of target addresses specified in the branch table, it can be performed by a load from the branch table. This allows a data unit operation and/or a local transfer to be specified in parallel with the branch.

Notes:

1) Loads to the pc register must be word data size; halfword and byte loads to the pc register are not allowed.

2) Loads to the pc register from memory should not use an unscaled immediate-offset. The return from interrupt (reti) operations use unscaled immediate offsets for loads to the pc register to indicate special modification of the G and L control bits.

Example 10–10. Branch Table

```
Branch_Tbl_Offset: .set   0x400
Ga_Branch_Tbl:     .set   a10

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Branch Table Set-Up
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   d7 = Sub1             ;  Determine 1st branch address.
   *(Ga_Branch_Tbl = pba + [Branch_Tbl_Offset]) = d7
                         ;  Set branch table base address
                         ;   and store first entry.
   d7 = Branch1
                         ;  Determine 2nd branch address.
   *(Ga_Branch_Tbl + [1]) = d7
                         ;  Store 2nd branch table entry.
      .
      .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; End of branch table set-up
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      .
      .
   <Data Unit Op>
   ||br = *(Ga_Branch_Tbl + [1])  ;  Branch to Branch1
                                  ;   using a load from
                                  ;   the Branch Table.
      <||Local Transfer>

   <Instruction>                  ;  1st delay slot
                                  ;   instruction
   <Instruction>                  ;  2nd delay slot
                                  ;   instruction
      .
      .

Branch1:
      .
      .

Sub1:
      .
      .
```

10.8 Two-Input ALU Operations With pc Destination

The pc register can be specified as the destination of any arithmetic or Boolean ALU operation involving two inputs (base set ALU operation class 7 with the B port input ignored). Note that the inputs cannot be routed through the barrel rotator, mask generator, expander, or bit-detection logic.

Table 10–1 lists the two-input arithmetic and two-input Boolean ALU operations that can written to the pc register (either **br** for branches or **call** for subroutine calls saving a return address).

Table 10–1. Supported ALU Operations to the pc

Two-Input Arithmetic

```
br = src1 – src2         ; allows br = ipe – offset
br = src1 + src2         ; allows br = ipe + offset
```

Two-Input Boolean

```
br = src1
br = ~src1
br = src2                ; allows br = address
br = ~src2


br = src1 | src2
br = ~src1 | src2
br = src1 | ~src2
br = ~src1 | ~src2


br = src1 & src2
br = ~src1 & src2
br = src1 & ~src2
br = ~src1 & ~src2


br = src1 ^ src2
br = src1 ^ ~src2
```

Note: src1 can be any register. src2 can be any D register or an immediate.

Chapter 11

Understanding the Loop Control Logic

The program flow control unit includes three hardware loop controllers that support up to three simultaneous loops. Since each PP instruction performs so many actions in parallel, key loops often require very few instructions. Thus, overhead is greatly reduced by using hardware looping.

Topics

11.1	Looping	PP: 11-2
11.2	Basic Loop Control	PP: 11-3
11.3	Setting Up the Loop Control Registers	PP: 11-6
11.4	Nested Loops	PP: 11-14
11.5	Uses of Loop Control Flexibility	PP: 11-19

11.1 Looping

The program flow control unit contains three comparators and associated loop control hardware that support up to three levels of zero-overhead looping. Looping is considered zero-overhead because it occurs in parallel with the normal single-cycle instruction execution so that no looping-related instructions are required within the loop. A loop count decrement and conditional branch to the loop start address occur automatically in hardware each time a loop end is matched. Thus, repetitive operations can be coded in tight loops with no associated loop control overhead.

The loop control logic operates in the fetch pipeline stage. When a loop end address is matched, the corresponding loop start address is loaded to the PC, and that instruction is fetched on the cycle that immediately follows (with zero delay). In contrast, software branches occur in the execute stage of the pipeline. Thus, two delay-slot instructions are fetched between an instruction specifying a software branch and the write to the pc register of the branch target address. Hardware looping greatly reduces the amount of software branching required and therefore makes code more efficient.

In addition to supporting three levels of nested zero-overhead loops, the PP loop control logic supports a variety of features including multiple loop ends associated with the same loop count, hardware branching, and conditional hardware branching.

11.2 Basic Loop Control

This section introduces the basic loop-control mechanisms and describes how they are used to provide looping capability.

11.2.1 Loop Control Registers

Looping on the PP is controlled primarily by the values in the following registers, which are defined further in Chapter 7, *Summary of PP Registers*.

- **Loop End Registers: le2, le1, le0**

  Each loop end register points to the last instruction in a loop. During each instruction fetch, the loop end register is compared to the pc to detect the loop end. When the loop end register matches the pc (when the last instruction in a loop is being fetched), the corresponding loop hardware action is invoked.

- **Loop Start Registers: ls2, ls1, ls0**

  Each loop start register points to the first instruction in a loop. The loop start register is copied over the pc when the looping hardware wants to branch back to the beginning of the loop. This is done in the cycle during which the pc matches the associated loop end register.

- **Loop Counter Registers: lc2, lc1, lc0**

  Each loop count register contains a count of the number of times to copy the associated loop start register over to the pc (the number of times to branch to the starting instruction of the loop). The loop count is decremented each time the associated loop end register matches the pc and the associated loop start register is copied over the pc. Since the loop count is the number of times to branch, it is the number of times to do the loop, minus one.

❑ Loop Reload Registers: lr2, lr1, lr0

Each loop reload register contains an initialization value for the associated loop count register. This reinitialization takes place after the last time through the loop. When the associated loop count register is zero at the end of the loop, instead of decrementing the loop counter, the loop counter is loaded from the reload register. This prepares the loop counter for the next time the loop is entered.

Note:

Writes to a loop reload register automatically write to the loop count register as well, but the converse is not true. This mechanism assists in faster loop initialization.

❑ Loop Control Register: lctl

The loop control register contains three similar 4-bit fields, each field relating to a loop end register. One bit in each field enables the comparator to the address in the loop end register. The other three bits designate the associated loop counter, which can be none, lc0, or the loop counter with the same number as the loop end register.

11.2.2 Loop End Comparison

Each time an instruction is fetched, the PC is compared with the address contained in each loop end register that is enabled (as discussed in subsection 11.3.1.3). When a match is found, the instruction address in the associated loop start register (ls2 for le2, ls1 for le1, and ls0 for le0) is loaded into the PC. Also, the count in the loop counter register, and assigned in the loop control (lctl) register (as discussed in subsection 11.3.1.3) is decremented.

11.2.3 Loop Count Decrement/Reload

A loop counter is decremented each time that a loop end associated with it is encountered, unless the loop count is 0. lc2 and lc1 can be associated only with le2 and le1, respectively. lc0 can be associated with any, some, or all of the loop end registers.

When the PC matches an enabled loop end address and the associated loop count is zero (before decrement), the PC is not loaded with the loop start address and the loop count is not decremented. Instead, the PC increments by 1, thus exiting the loop, and the associated loop counter is loaded with the contents of the loop reload register.

11.3 Setting Up the Loop Control Registers

Looping requires setting up the loop control registers. These registers can be written individually (for general case loops) or by using one of two shortform loop initialization register codes (for special case loops).

11.3.1 General Loop Control Initialization

Loop control values can be initialized by individually loading all of the loop control registers, as described in this section. Since the loop registers are loaded by software in the execute pipeline stage and used by the loop control hardware in the fetch stage, there should always be at least two instructions between the loading of any loop register and the loop end address at which that register will be used. (It is, however, permissible to modify other loop registers not related to that specific loop end. For example, le2 can be modified in the two instructions before le1 is encountered, provided that the loop end address associated with le2 occurs at least two instructions later).

As shown in Example 11–1, initializing a loop by individually setting the loop control registers requires the following steps:

1) Set the loop end register to the address of the last instruction in the loop. This determines where the loop back occurs.

2) Set the loop start register to the address of the first instruction in the loop. This determines the address that is looped back to.

3) Set the loop reload register to the number of loop backs to be performed each time the loop is entered (the number of times to do the loop, minus one). The same value is also written automatically to the loop counter register.

4) Optionally set the loop counter register (if the desired initial loop counter value is different than the loop reload register value) to the number of loop backs for the first time the loop is entered.

5) Set bits in the lctl register to enable the loop end, and designate the associated loop counter. A loop end should not be enabled (in lctl) until after the correct loop end address has been set up.

Example 11–1. Loop Set Up

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Loop Set Up
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   le0 = Loop_End0        ; Set loop end address.
   ls0 = Loop_Start0      ; Set loop start address.
   lr0 = 99               ; Perform loop instructions
                          ; 99+1 times (loop back to
                          ; ls0 99 times).
   lctl = 0x9             ; Enable le0 and associate
                          ;  it with lc0.

   <Instruction1>         ; 1st Delay Slot Instruction
   <Instruction2>         ; 2nd Delay Slot Instruction
             .
             .
             .
Loop_Start0: .
             .
             .
Loop_End0:   .
```

11.3.1.1 Initializing Loop End and Start Addresses

Loop end and start addresses can be made code-position independent by computing them as offsets with respect to the ipe register, as shown in Example 11–2 (for a general discussion of PC-relative address computations, see subsection 10.5.2, *Relative Branches*).

Example 11–2. Code-Position Independent Loop End Specification

```
    le0 = ipe + (loopend_label – $)
        .
        .
        .
loopend_label:
        .
        .
```

11.3.1.2 Initializing Loop Reload and Loop Counter Values

When you specify a write to a loop reload register (lr*n*), a write to the associated loop counter register (lc*n*) is also performed automatically. If the desired lc*n* value is the same as the lrn value, then only lr*n* needs to be loaded by software. **If you want different values in lc*n* and lr*n*, you must set lrn first.** Otherwise, writing the lrn value will overwrite the desired lc*n* value.

The value in the loop counter register determines the number of times that the hardware branch to the loop start address is performed. Hence, the value written to the loop reload and/or loop counter register should be one less than the number of times the instructions in the loop should be performed. For example, if a loop operates on one pixel per pass through the loop and it needs to be performed on 100 pixels, the loop counter should be set initially to 99. Since the loop counter is decremented each time a loop back is performed, the loop counter constantly reflects the number of remaining passes through the loop (not including the current one).

11.3.1.3 Enabling Loop Ends and Designating the Associated Loop Counter

The loop control register (lctl) shown in Figure 11–1 enables the comparison between the PC and each individual loop end and associates a loop counter with each loop end. lctl contains three similar 4-bit fields, each field relating to a loop end register. The three enable (E) bits activate the corresponding loop end register comparators. Looping is completely disabled by setting these three bits to 0. When a PP is reset, all lctl E bits are cleared. An lctl E bit should not be set to 1 until the corresponding loop end register has been set.

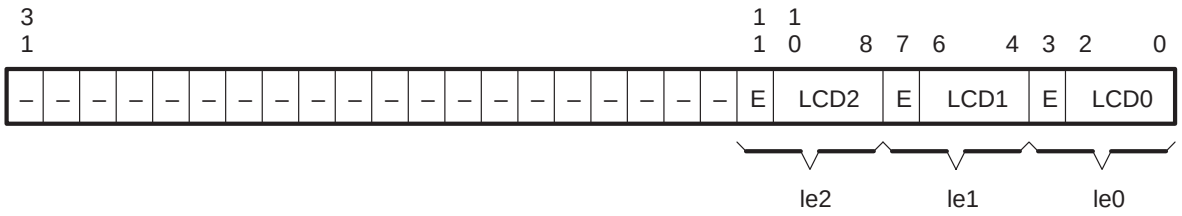
The three LCD*n* fields, each consisting of three bits, designate which loop counter should be decremented when the loop end register is successfully compared against the program counter. The only permitted loop counter assignments are:

- **le0**: lc0 or none
- **le1**: lc1, lc0, or none
- **le2**: lc2, lc0, or none

All other combinations are reserved. If lc0 is designated for two or three loop end registers, multiple ends to the same loop can be implemented (see subsection 11.5.1, *Multiple Ends Associated WIth the Same Loop Counter*).

If the LCD*n* field is 0, no loop counter is checked, decremented, or reloaded, and the loop start copy always occurs. Thus, the loop control logic can be used to perform hardware branches (subsection 11.5.2, *Hardware Branching*) without having to maintain a nonzero loop count.

Figure 11–1. lctl Register



E — Loop end enable

LCD*n* — Loop counter designator for loop end

000 — None. Always load loop start into PC

001 — lc0

010 — lc1

011 — lc2

1XX — Reserved

11.3.2 Shortform Loop Control Initialization

The overhead associated with individually setting all of the loop control registers to initialize a loop that is executed many times is almost insignificant. However, the overhead becomes significant for small short-lived loops. To support faster loop initialization, two shortform methods are provided for two special cases of loop setup performed immediately before entry to the loop.

11.3.2.1 Shortform for a Single-Instruction Loop

Special register codes (lrse2–lrse0) support fast initialization of single-instruction loops. Writing a loop count value to an lrse*n* (where *n* = the loop register number) register performs the following:

- Sets lr*n* (and lc*n*) to the desired count.
- Sets ls*n* and le*n* to the address of the instruction sequentially following the second delay-slot instruction (that is, PC+1).

  Note that a single instruction loop is possible with le*n* equal to ls*n*.
- Sets the corresponding E bit in lctl, enabling le*n*. Sets the loop counter designator to lc*n*.

These operations all occur in a single cycle during the execute pipeline stage. There are two delay-slot instructions between fetching the instruction that writes to lrse*n* and the start/end address of the loop. The loop start and loop end addresses are calculated in relation to the PC value of the second delay-slot instruction. If one of the two instructions fetched immediately **before** the shortform loop initialization operation is a branch, the loop start and loop end are calculated in relation to the PC value after the branch is executed. Shortform loop initialization should therefore be used with care within the delay-slot instructions of a branch.

In Example 11–3, assuming that the instruction that writes to lrse*n* is not in the delay slot of a branch and that neither it nor its first delay-slot instruction (<Instruction1>) occurs at a loop end, the loop start and loop end addresses for the loop are set to the third subsequent instruction in memory (<Instruction3>).

The code shown in Example 11–3 sets up the loop control logic associated with le2 to perform <Instruction3> four times. The value 3 written to lrse2 indicates the number of times to loop back to the start of the loop, <Instruction3>, as illustrated in Figure 11–2. The fourth time <Instruction3> is fetched, lc2 is 0 and therefore, the loop back does not occur. Instead, the PC is incremented by one, le2 is reloaded with 3 from lr2, and <Instruction4> becomes the next instruction fetched.

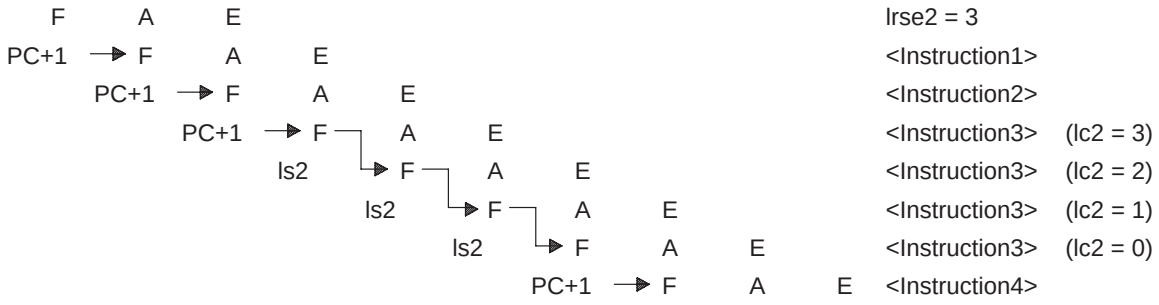
Note:

When editing code between a shortform loop initialization operation and the desired loop start address, be careful to preserve exactly two delay-slot instructions.

Example 11–3. Single-Instruction Loop Shortform Initialization

```
        lrse2 = 3          ; Perform instructions in the
                           ; loop 3+1 times.
        <Instruction1>     ; Delay Slot1 Instruction
        <Instruction2>     ; Delay Slot2 Instruction
loop2:  <Instruction3>     ; Single-Instruction Loop.
        <Instruction4>     ;
```

Figure 11–2. Program Flow for Example 11–3



11.3.2.2 Shortform Initialization for Multi-Instruction Loop

Special register codes (lrs2–lrs0) support fast initialization of multi-instruction loops. Fast multi-instruction loop set-up is identical to single-instruction loop set-up except that the associated loop end register is not modified when writing to the lrs*n* register; it must be initialized in a separate instruction. Writing a loop count value to an lrs*n* (where *n* = the loop register number) register has this effect:

1) Sets lr*n* (and lc*n*) to the desired count.

2) Sets ls*n* (loops start address) to point to the instruction sequentially following the second delay-slot instruction.

3) Sets the corresponding E bit in lctl, thus enabling le*n*. Sets the loop counter designator to lc*n*.

These operations all occur in a single cycle during the execute pipeline stage. There are two delay-slot instructions between the fetching of the instruction that writes to lrs*n* and the start of the loop. The loop start address is calculated from the PC value of the second delay-slot instruction. If one of the two instructions fetched immediately **before** the shortform loop initialization operation is a branch, the loop start is calculated in relation to the PC value after the branch is executed. Shortform loop initialization should therefore be used with care within the delay-slot instructions of a branch.

In Example 11–4, assuming that the instruction that writes to lrs*n* is not in the delay slot of a branch and that neither it nor its first delay-slot instruction (<Instruction1>) occurs at a loop end; the loop start address for the loop is set to the third subsequent instruction in memory (<Instruction3>). The instruction sequence shown in Example 11–4 sets up the multi-instruction loop starting at <Instruction3> to be performed four times.

Example 11–4. Multi-Instruction Loop Shortform Initialization

```
                le1 = loop_end1
                lrs1 = 3            ; Perform instructions in
                                    ;  the loop 3+1 times.
                <Instruction1>      ; Delay Slot1 Instruction
                <Instruction2>      ; Delay Slot2 Instruction
loop_start1:    <Instruction3>      ; 1st Instruction in
                                    ;  loop.
                   .
                   .
loop_end1:      <InstructionN>      ; Last Instruction in
                                    ;  loop.
```

Note that for the multi-instruction loop shortform initialization, the loop end address is loaded separately. It is recommended for safe coding that loop ends never be enabled (by setting lctl directly or with lrs*n*) before the associated loop end register has been set, so no undesired loop-related branches are taken.

Note:

When editing code between a shortform loop initialization operation and the desired loop start address, be careful to preserve exactly two delay-slot instructions.

11.4 Nested Loops

The PP supports up to three levels of nested loops in hardware. Nested loops are frequently useful in image processing. Example 11–5 shows how to initialize three levels of nested looping. Note that shortform loop initialization is used for all three loops, even though there is more than one delay slot between the shortform register assignment and the corresponding loop start. This is done by assigning the desired address to the loop start register before the loop end register is ever encountered. In this example, the outer loop could perform the operations that set up an 8×8 block of data to be processed.

This may include

- Verifying that the block of data to be processed by the middle and inner loops is on-chip by checking that a packet transfer request has completed.
- Submitting a packet transfer request to:
  - Transfer on-chip the block of data that will be processed by the next pass through the nested loops.
  - Transfer off-chip the block of data that was processed by the previous pass through the nested loops.

The middle loop could be the row loop, plus handle special boundary conditions for the first and last pixel in each of the eight rows within an 8×8 block, as well as the address increment from the last pixel within the 8×8 block for one row of data to the first pixel within the block on the next row. Finally, the inner loop would perform the operations that are repeated for each of the six inner pixels within the same row of the block of data.

Example 11–5. Three Levels of Nested Looping

```
loop_setup:
   le1 = middle_end
   lrs1 = 7
   ls1 = middle      ; Loop back to the start of the
                     ; middle loop 7 times per pass of
                     ; the outer loop thus processing
                     ; each of the 8 rows in an 8x8
                     ; block.

   le2 = inner_end
   lrs2 = 5
   ls2 = inner       ; Loop back to the start of the
                     ; inner loop 5 times per pass
                     ; of the middle loop thus
                     ; processing the 6 inner pixels
                     ; in a row of the 8x8 block of
                     ; of data.
   le0 = outer_end
   lrs0 = 5          ; Loop back to the start of the
                     ; outer loop 5 times thus
                     ; processing 6 8x8 blocks.

<delay slot instruction 1>
<delay slot instruction 2>

outer:      .        ; Start of outer loop.
            .
            .
middle:     .        ; Start of middle loop.
            .
            .
inner:      .        ; Start of inner loop.
            .
            .
inner_end:  .        ; Inner loop end.
            .
            .
middle_end: .        ; Middle loop end.
            .
outer_end:  .        ; Outer loop end.
            .
            .
```

11.4.1 Loop End Priority

Loop ends (and/or loop starts) can be coincident. When two or three loops end at the same address, the loop end registers are prioritized from le2 (highest) to le0 (lowest). Thus, le2, le1, and le0 correspond to the inner, middle, and outer loops, respectively. The loop start register associated with the highest priority matched loop end register with a nonzero loop count is loaded into the PC, and the associated loop counter is decremented. Also, the zero-value loop counters of higher priority, PC-matching loop end registers are reloaded from their associated loop reload registers.

In Example 11–6, the outer, middle, and inner loops have the same loop end address, and lctl = 0xBA9. Due to the prioritization scheme, the following events occur when the PC equals the loop end address (that is, the instruction at the loop end address is fetched):

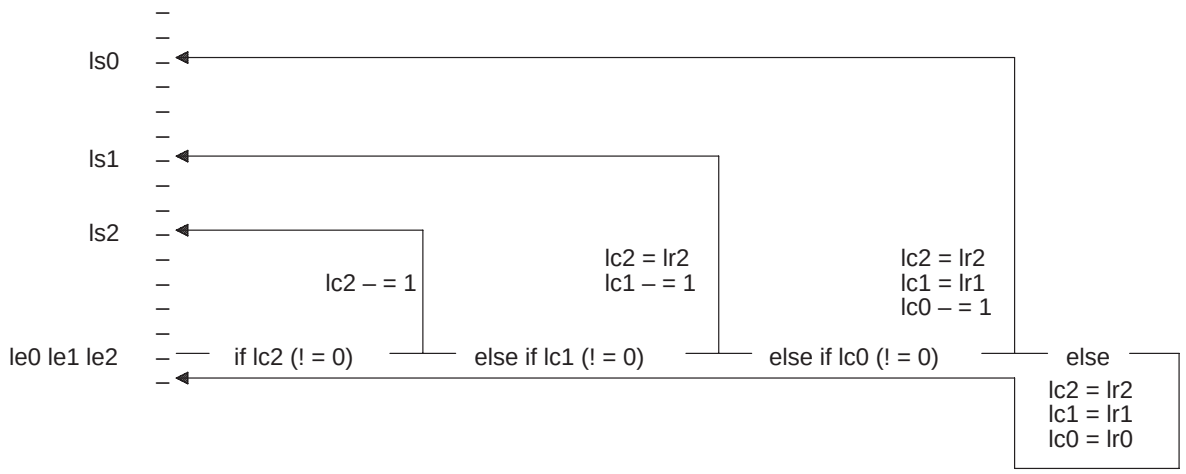
- When lc2 ≠ 0,
  - PC is loaded with ls2 (repeat inner loop)
  - lc2 is decremented
- When lc2 = 0, and lc1 ≠ 0,
  - PC is loaded with ls1 (repeat middle loop)
  - lc2 = lr2
  - lc1 is decremented
- When lc2 = 0, lc1 = 0, and lc0 ≠ 0,
  - PC is loaded with ls0 (repeat outer loop)
  - lc2 = lr2
  - lc1 = lr1
  - lc0 is decremented
- When lc2 = 0, lc1 = 0, and lc0 = 0,
  - PC is incremented by 1 (loop is exited)
  - lc2 = lr2
  - lc1 = lr1
  - lc0 = lr0

Example 11–6. Three Levels of Nested Looping With Same Loop End

```
loop_setup:
   le1 = middle_end
   lrs1 = 7
   ls1 = middle      ; Loop back to the start of the
                     ; middle loop 7 times per pass of
                     ; the outer loop thus processing
                     ; each of the 8 rows in an 8x8
                     ; block.

   le2 = inner_end
   lrs2 = 5
   ls2 = inner       ; Loop back to the start of the
                     ; inner loop 5 times per pass
                     ; of the middle loop thus
                     ; processing the 6 inner pixels
                     ; in a row of the 8x8 block of
                     ; of data.
   le0 = outer_end
   lrs0 = 5          ; Loop back to the start of the
                     ; outer loop 5 times thus
                     ; processing 6 8x8 blocks.

<delay slot instruction 1>
<delay slot instruction 2>

outer:         .     ; Start of outer loop.
            .
            .
middle:     .     ; Start of middle loop.
            .
            .
inner:         .     ; Start of inner loop.
            .
inner_end:
middle_end:
outer_end:  .     ; All 3 loops have same end.
            .
            .
```

The sequence of program flow for Example 11–6 is illustrated in Figure 11–3.

Figure 11–3. Three Loops Ending at Same Address



11.5 Uses of Loop Control Flexibility

The PP loop control logic flexibility can be used for a variety of functions, such as speeding up critical tight loops. This section discusses multiple loop ends associated with the same loop counter, hardware branching, conditional software branching and conditional hardware branching.

11.5.1 Multiple Ends Associated With the Same Loop Counter

The loop control logic allows multiple loop ends to be associated with the same loop counter. This is useful for data-dependent algorithms.

Example 11–7 illustrates how two loop end addresses associated with the same loop counter can be used for performing variable length code (VLC) table look-ups (TLUs). The first step is to perform a TLU on the next eight bits of data. The returned table value indicates whether these eight bits contain a full VLC.

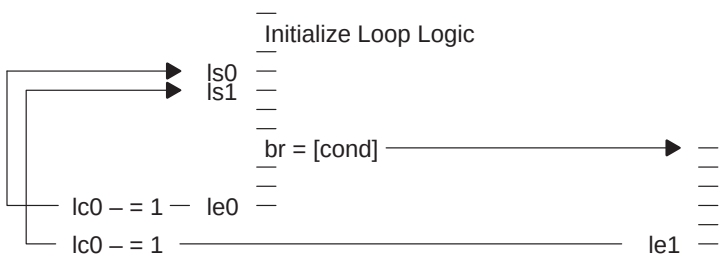
If they do, the routine progresses to le0, at which point lc0 is decremented and the PC is set to ls0. If a full VLC is not contained in the first eight bits, a branch is taken to a routine that performs a second TLU on the next eight bits. The last instruction of the routine that performs the second TLU is set up as le1, which decrements lc0.

The flexibility of the loop control logic is shown in Example 11–7 where the loop back from le1 returns to a different address than le0. With this looping set up, each time a VLC is successfully identified (after either one or two TLUs), lc0 is decremented Figure 11–4).

Example 11–7. Bitstream Decoding With Two Ends to Same Loop

```
      le1 = Second_TLU_end
      ls1 = Second_TLU_ret
      le0 = First_TLU_end
      lrs0 = 63             ; Number of VLCs to look up is
                            ;  64.
      lctl = 0x99           ; Enable le1 and le0. Associate
                            ;  both with lc0.
      <Instruction1>        ; Delay Slot2 instruction

   First_TLU:
      <Instruction2>
   Second_TLU_ret:
      <Instruction3>
      <Instruction4>
      <Instruction5>
      <Instruction6>
      br =[cond] Second_TLU ; Branch to second TLU routine.
      <Instruction8>
      <Instruction9>
   First_TLU_end:
      <Instruction10>

         .
         .
         .

   Second_TLU:
      <InstructionA>
         .
         .
   Second_TLU_end:
      <InstructionZ>
```

Figure 11–4. Two Ends to Same Loop



11.5.2 Hardware Branching

The PP's loop-control logic makes is possible to perform hardware branching. When a loop end is encountered and the corresponding loop counter is nonzero, the PC essentially branches to the address specified in the associated loop start register. In addition, the lctl register can be set up so that the hardware branch is always performed and no loop counter is decremented when the loop end is encountered.

Once the loop end register has been loaded and enabled, a branch occurs each time the instruction at the associated address is fetched; no delays occur between the loop end (pseudobranch request) and the actual branch. Additionally, no instruction is required to request the branch after the looping hardware is configured. Thus, the advantage of hardware branching over software branching is that it requires essentially zero overhead (other than the initial set up of the loop control registers).

Often, hardware branching is associated with a second hardware branch that essentially performs a return. A hardware branch followed by a hardware return typically occurs within an outer loop that initializes the generic aspects of an operation.

For example, most of the coding of various PIXBLT functions is the same, with the exception of the specific pixel-processing inner loop (which is typically two or three instructions). By using hardware branching to enter and return from the pixel-processing subroutine, you can keep code size to a minimum because no delay slots are incurred and no branch instructions are required.

Example 11–8 shows how to set up a hardware branch and hardware return for a generic PIXBLT outer loop. Specific PIXBLT operators such as AND are coded in subroutines.

Hardware branching lets you avoid the instructions required for software branching and their associated delay-slot instructions. Instead, roughly nine instructions are required initially to set up the three sets of loop control registers. If the pixel processing operation is performed on more than a couple of pixels, the loop control initialization (that is required only once) becomes relatively inexpensive, compared to software branching.

Example 11–8. PIXBLT Example With Fast Subroutine Call

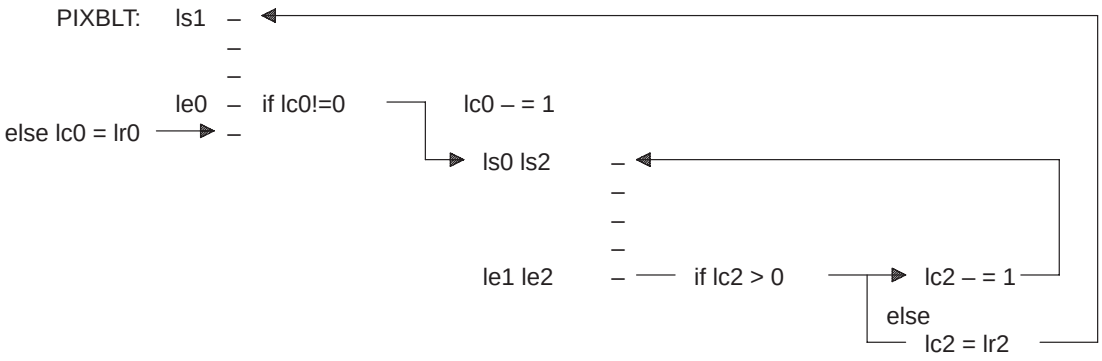
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Set up of loop control registers for a hardware branch
;; and hardware return for desired pixel processing
;; subroutine.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   le0    =  subcall
   ls0    =  xor
   lc0    =  height – 1
   le1    =  xore
   ls1    =  subret
   le2    =  xore
   ls2    =  xor
   lc2    =  width/size – 1
   lctl   =  0xB89

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; PIXBLT Outer Loop.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PIXBLT:
subret:                     ; Zero-delay return address.
      .
      .
subcall:                    ; Zero–delay branch occurs.
   d3 = *a8++               ; Load src2.
      ||d2 = *a0++          ; Load src1.
      .
      .
      .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; AND Pixel Processing Subroutine
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
and:
   d4 = d2 & d3             ; src1 AND src2
    ||d3 = *a8++            ; Load src2
    ||d2 = *a0++            ; Load src1
ande:
   *a9++ = d4               ; Store processed pixel.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; XOR Pixel Processing Subroutine
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
xor:
   d4 =d2 ^ d3              ; src1 XOR src2
    ||d3 = *a8++            ; Load src2
    ||d2 = *a0++            ; Load src1
xore:
   *a9++ = d4               ; Store processed pixel.
      .
      .
      .
```

In Example 11–8, le0 is set up to cause a hardware branch to a pixel-processing subroutine. This hardware branch occurs as long as lc0 is nonzero. ls0 is set to the start address of the desired pixel-processing subroutine.

Once in the pixel-processing subroutine, le2 (which is set to the last instruction of the subroutine) causes the program to loop in the subroutine until lc2 is zero. le1, which is lower priority than le2, is also set to the last instruction of the subroutine. When lc2 is 0 and the last instruction of the subroutine is fetched, a hardware branch back to the PIXBLT outer loop occurs. ls1 is set to the beginning of the PIXBLT outer loop. No loop counters are associated with le1. This allows the hardware return to be implemented without having to maintain a nonzero loop count. The associated loop start register will always be loaded; therefore, the hardware return will always occur.

Figure 11–5 illustrates the program flow for Example 11–8.

Figure 11–5. Program Flow for PIXBLT Fast Subroutine Call



11.5.3 Conditional Software Branching

Software branches (and calls) have priority over looping when they are coincident (the loop end address is the second delay-slot instruction of a branch). Loop counters are not decremented or reloaded during a cycle in which the PC is loaded by a software branch (the loop logic is effectively disabled for that cycle). This allows a software exit from a loop upon a condition becoming true.

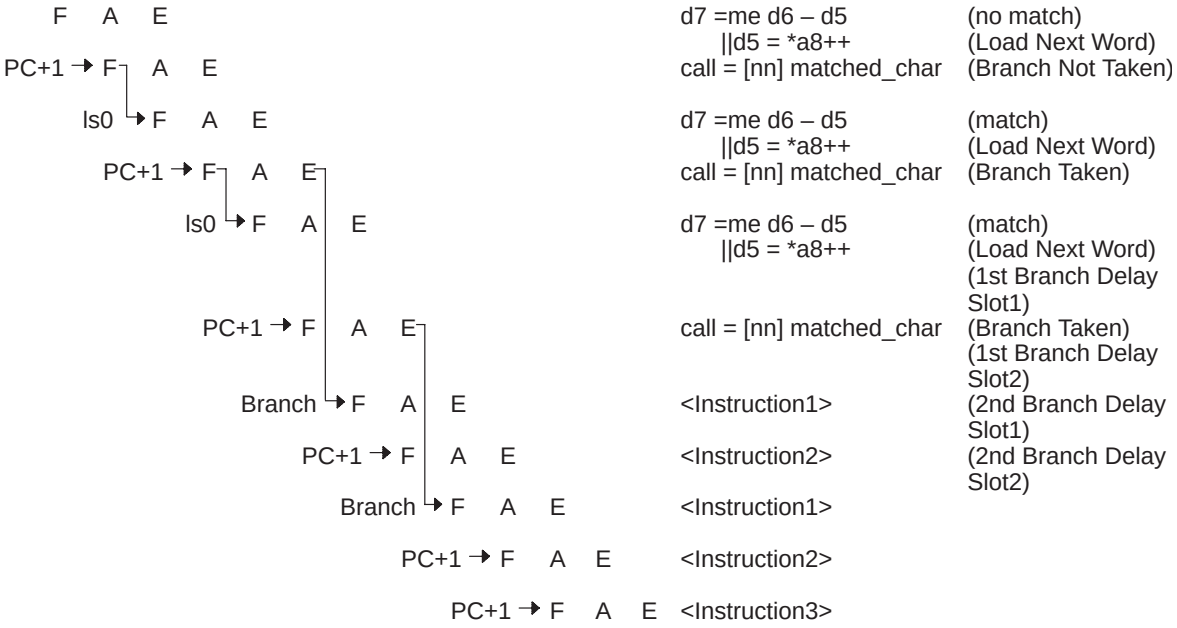
Conditional software branches out of single- or two-instruction loops should be used carefully because the conditional branch occurs in its own delay slot. Example 11–9 uses a conditional branch to exit a loop if a character is matched. Four characters are checked with a multiple-byte compare that saves the split-ALU zero comparisons to the mf register. For multiple arithmetic, the negative status bit records the NOR of the flags saved to mf. Therefore, a single conditional branch can be used to exit the loop if any of the four bytes matches the desired character.

Example 11–9. Character Search Using Conditional Software Branch Exit

```
sr = 4                  ; Set Asize for byte multiple
                        ; arithmetic and MSS for setting mf
                        ; bits according to zero compares.
loop0: d7 =me d6–d5 ;Test 4 bytes for character match.
   ||d5 = *a8++         ; Load next word of 4 bytes.
loop0e:   call =[nn] matched_char
                        ; Exit if character was matched.
       .
       .
       .

matched_char:
   <Instruction1>
   <Instruction2>
   <Instruction3>
       .
       .
```

Figure 11–6 illustrates that in Example 11–9, the second delay-slot instruction of the conditional branch coincides with the loop end. Also, the conditional branch is in its own second delay slot; therefore, if the condition remains true (after the first time it is met), the branch may actually occur twice.

Figure 11–6. Conditional Software Branch in its Own Delay Slot



11.5.4 Conditional Hardware Branching

Conditional hardware branching can be implemented by loading a value into the designated loop counter at least two instructions before a loop end address is encountered.

When the instruction at a loop end address is fetched, the associated loop start address is loaded into the PC by the loop control logic only if the designated loop counter value is nonzero. Thus, conditional hardware branching can be implemented by writing to a loop counter either with a nonzero value (in which case the loop is taken when the instruction at the loop end address is fetched) or with zero (in which case the loop is not taken when the loop end address is fetched).

Note that software writes to a loop counter dominate over the loop count decrement associated with a loop end address.

The character search example shown in the previous section can be optimized by using conditional hardware branching to exit the loop when a character match is found instead of by using conditional software branching.

In Example 11–10, the character search example is implemented with a single-instruction loop. In each cycle, four bytes are compared against the character that is to be matched. The zero compares for each of the split-ALU results are written to the mf register.

If no match is found, all of the zero compares will be false, so a zero value is stored in mf. If a match is found, at least one of the zero compares will be true, so a nonzero value will be stored in mf. In the next cycle, this value will be written to lc2 (the loop counter associated with le2).

Two delay-slot instructions are fetched before the execute stage of the write to lc2 with the nonzero value occurs; then the character search inner loop is exited.

Example 11–10. Character Search Using Conditional Hardware Branch

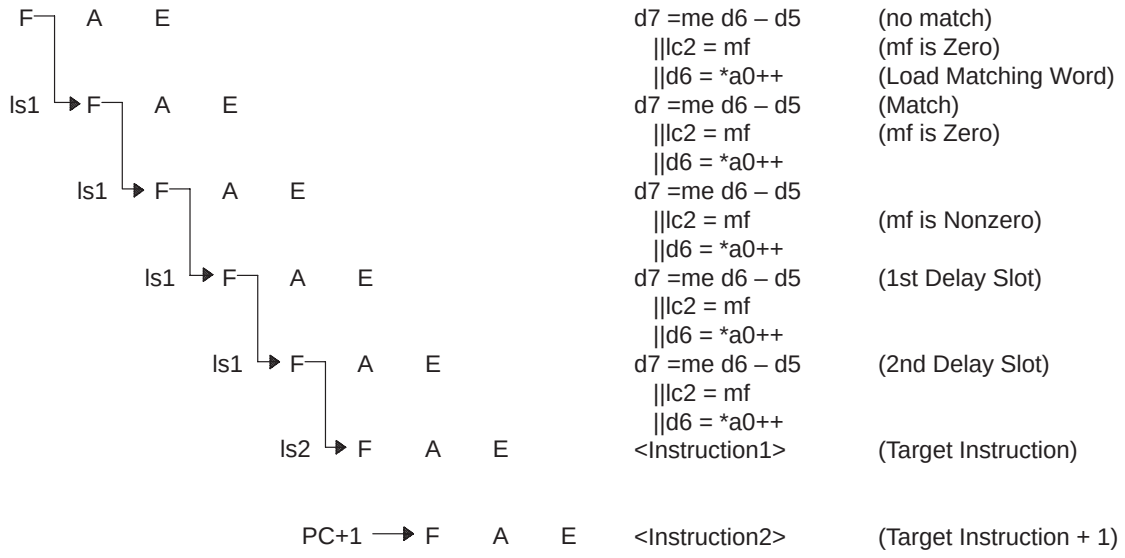
```
   sr = 4          ;  Set Asize for byte multiple
                   ;     arithmetic and MSS for setting mf
                   ;     bits according to zero compares.
   d5 = 0xF0F0F0F0 ;  Put in reg. Packed Character
                   ;     that is being searched.
   le2 = loop2_end
                   ;  Set address for conditional
                   ;     hardware branch out of the
                   ;     character search loop.
   lrs2 = 0        ;  Zero loop count (lc2) associated
                   ;     with conditional branch and
                   ;     enable le2. Note that ls2
                   ;     still needs to be set to the
                   ;     correct target address for
                   ;     the conditional hardware branch.
   lrse1 = 511     ;  Set loop1 to search up
                   ;     to 512 words (2k bytes).
   ls2 = matched_char
                   ;  Correct the target address for
                   ;     the conditional hardware branch.
   mf = 0          ;  prezero mf
      ||d6 = *(a0=dba)
                   ;  Load first word to be searched
                   ;     after setting the pointer.

loop1_start:
loop1_end:
loop2_end:
   d7 =me d6 – d5  ;  Compare 4 bytes to the
                   ;     character being searched for.
      ||lc2 = mf   ;  Set lc2 to value of the multiple
                   ;     zero flags for previous compare.
                   ;     If match is found (mf has
                   ;     set zero flags), then after 2
                   ;     delay–slot instructions, the pc
                   ;     will be set to ls2 thus exiting
                   ;     the search inner loop.
      ||d6 = *a0++ ;  Load next word.

      .
      .
      .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; After the character search loop is exited by the
;; conditional hardware branch, the data pointer needs
;; to be decremented by 4 words to reload the first word
;; that matched the character to determine which byte
;; matched.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

matched_char:
   <Instruction1>
   <Instruction2>
      .
      .
```

Figure 11–7 shows that after a word with a matching character is loaded, the character search loop is executed four more times:

- Two times because of latency. The mf move into lc2 occurs two cycles after the data is loaded.
- Two pipeline delay-slot instructions. Two additional instructions have already been fetched when the mf move to lc2 is executed.

To identify the exact byte that matched, the pointer must be decremented by four, and the compare must be re-executed.

Figure 11–7. Conditional Hardware Branch out of a Single-Instruction Loop



Chapter 12

Packet Transfers

This chapter describes the format of packet transfers, their linked-list addressing, the polling and submitting of packet transfer requests to the TC, and the determination of when the data transfer has completed. The chapter concludes with a description and examples of the two broad classes of packet transfer source and/or destination transfer modes:

- **Dimensioned transfers**, which are typically used for imaging algorithms, and
- **Guided transfers**, which are typically used for graphics.

For more detailed information on packet transfers, see Chapter NO TAG, *Packet Transfers*, in the *MVP Transfer Controller User's Guide*.

Topics

12.1	Packet Transfer Request Protocol	PP:12-2
12.2	Setting Up and Requesting a Packet Transfer	PP:12-3
12.3	Waiting for Packet Transfer Request Completion	PP:12-9
12.4	Dimensioned Transfers	PP:12-16
12.5	Guided Transfers	PP:12-27

12.1 Packet Transfer Request Protocol

Direct external access (DEA) for loads and stores between PP registers and off-chip memory is supported. However, it requires many cycles to perform one single-word access to off-chip memory. Since DEAs are not efficient when more than one or two words are required, the primary model for PP data access uses packet transfer requests, as follows:

1) When the PP requires data that is off-chip, it issues a request for data called a packet transfer request. (Alternatively, the master processor (MP) can issue a packet transfer request for a PP.) The transfer controller (TC) services the packet transfer request and transfers the data to the PP's shared RAM. This process requires more setup than a DEA request, but requires the setup only once for each data request.

2) The PP processes the data as needed by using the shared RAMs, which can be accessed in a single cycle over the MVP crossbar.

3) Output data generated by the PP is transferred from the shared RAMs to off-chip memory by the transfer controller in response to another packet transfer request.

The discussion of packet transfer request protocol between the PP and the TC is divided into two major topics:

- ❑ Setting up the desired packet transfer parameters and issuing a packet transfer request (Section 12.2, *Setting Up and Requesting a Packet Transfer*), and
- ❑ Verifying completion of an issued packet transfer request (Section 12.3, *Waiting for Packet Transfer Request Completion*).

For more detailed information about packet transfer protocol, see Chapter NO TAG, *Packet Transfers*, in the *MVP Transfer Controller User's Guide*.

12.2 Setting Up and Requesting a Packet Transfer

Making a packet transfer request consists of the following steps:

Step 1: Set up the packet transfer parameters

Step 2: Set the linked-list pointer in memory to the desired packet transfer parameters

Step 3: Issue the packet transfer request to the TC

12.2.1 Step 1: Setting Up the Packet Transfer Parameters

Packet transfers are defined by a structure of parameters that must be set up in one of the shared RAMs (typically in parameter RAM), starting at a 64-byte-aligned address, before the packet transfer request is issued to the TC.

These parameters specify:

- The source and destination transfer modes,
- The quantity, location, and structure of the data that is transferred, and
- Several optional features.

The exact layout of the parameter block depends on all these factors. Some common layouts and their explanations are found in Section 12.4, *Dimensioned Transfers*, and Section 12.5, *Guided Transfers*. For detailed information about setting packet transfer parameters, see Section NO TAG, *Setting Packet Transfer Parameters*, in the *MVP Transfer Controller User's Guide.*

An include file, packetpp.i, is provided with the MVP tools to facilitate access to the parameters of packet transfers. The file (shown in Appendix B, *packetpp.i Include File*) is used in the some of the examples in this chapter. Note that the structure of packet transfers is fixed by the hardware; the packetpp.i file's easy mechanism for accessing parameters contrasts with many UNIX include files that allow underlying hardware to change and still maintain the same source code software interface.

12.2.2 Step 2: Setting the Linked-List Pointer

Packet transfers use a linked-list structure. Each processor can have a number of different linked lists stored in memory. However, each processor can have only one linked list issued to the TC at a time.

The starting packet of a linked list of packets to be queued is specified by the linked-list pointer, which is at a dedicated location in each processor's parameter RAM (pba + 0x00FC or 0x0100#0FC, where # is the PP number). Each individual packet transfer contains a pointer to the next entry in the linked list. After the TC services a packet transfer request, it takes the pointer to the next linked-list entry from the packet transfer request and copies it to its linked-list pointer. This allows the TC to service other processor's packet transfer requests and return to service the next request in the current linked list.

Each packet transfer contains a stop bit in the PT options field that indicates whether it is the last packet in the list. The linked-list pointer in the final packet transfer of a linked list typically points to the first packet transfer for the next linked list of packet transfers that will be issued.

Before issuing a packet transfer request, the linked-list pointer must contain the address of the first packet transfer in the linked list of packet transfers. There are only two safe ways to set the linked list pointer:

- ❑ Ensure first that the processor's previous packet transfer request is no longer queued (see Section 12.3, *Waiting for Packet Transfer Request Completion*). The PP can then modify the linked-list pointer as desired. Or,
- ❑ Allow the final packet transfer of a linked list to set the linked-list pointer by using its next-entry address field to link to the next linked list of packet transfers.

Note:

Do not set the linked-list pointer value when a packet transfer is queued. To do so would risk overwriting the pointer to a previously issued packet transfer request that is not yet loaded because the TC is servicing packet transfers and/or caches for other processors.

12.2.3 Step 3: Issuing a Packet Transfer Request to the TC

To issue a packet transfer request to the TC, set the P bit in the comm register, as shown in Example 12–1. This places the processor's packet transfer request on the TC's queue of actions waiting to be serviced. The P bit and other packet transfer handshake signals and their protocol are discussed in subsection 12.2.4.

Example 12–1. Issuing a Packet Transfer Request: Setting the P Bit

```
comm = comm | 1<<28   ; Set bit 28 (P bit) in the comm
                      ; register.
```

or

```
comm = comm | 1\\PT_SubmitShift  ;if packetpp.i is included
```

12.2.4 Packet Transfer Handshake Signals

The packet transfer handshake signals are the means of communication between each PP and the TC. They are contained in the four MSBs (H, S, Q, and P) of the comm register. These bits and related interrupt bits are shown in Figure 12–1.

The P bit, as discussed subsection 12.2.3, places a packet transfer linked list in the TC's queue. This corresponds to issuing a packet transfer request to the TC. The P bit should not be set at the same time that a suspend request is made (by setting the S bit). When a suspend request is made by the TC, the PP hardware copies the Q bit to the P bit to indicate whether the packet request was still active (had not been completed) when the suspend request was made.

The Q bit indicates whether or not the packet transfer linked list is queued (and it has not been fully serviced yet). As discussed in Section 12.3, *Waiting for Packet Transfer Request Completion*, this bit can be tested to check whether a linked list of packet transfer requests has been serviced.

Note:

The Q bit is not set in the cycle immediately following the cycle in which the P bit is set. The Q bit is set after a delay of one instruction (one delay slot).

Note that if Q = 1, this does not necessarily indicate that the desired packet transfer information has been loaded into the TC's packet transfer mechanism. If the TC is busy servicing requests from other processors, there could be some delay from when the Q bit is set to 1, and when the TC actually loads the packet transfer parameters and begins servicing the request.

The H bit indicates the priority of the packet transfer request. A PP can set its packet transfer request at either a low (H = 0) or high (H = 1) priority.

- A **low-priority** for a packet transfer gives PP accesses to shared RAM priority over the packet transfer's accesses to the shared RAM.
- A **high-priority** for a packet transfer gives the packet transfer's accesses to shared RAMs priority over PP accesses to shared RAM.

The H bit can also be used to prioritize a particular packet transfer request over packet transfer requests submitted by other processors.

The packet transfer priority can be changed dynamically by setting or clearing the H bit. The TC monitors the status of the H bit continuously, so changes to the H bit changes the priority of the packet transfers currently submitted by the PP.

The S bit is used to suspend a queued packet transfer request.

The protocol for manipulation of the packet transfer handshake bits in the comm register is shown in Figure 12–2. The H, S, Q, and P bits are all cleared to 0 when a software or hardware reset occurs.

Figure 12–1. Packet Request Handshake Signals

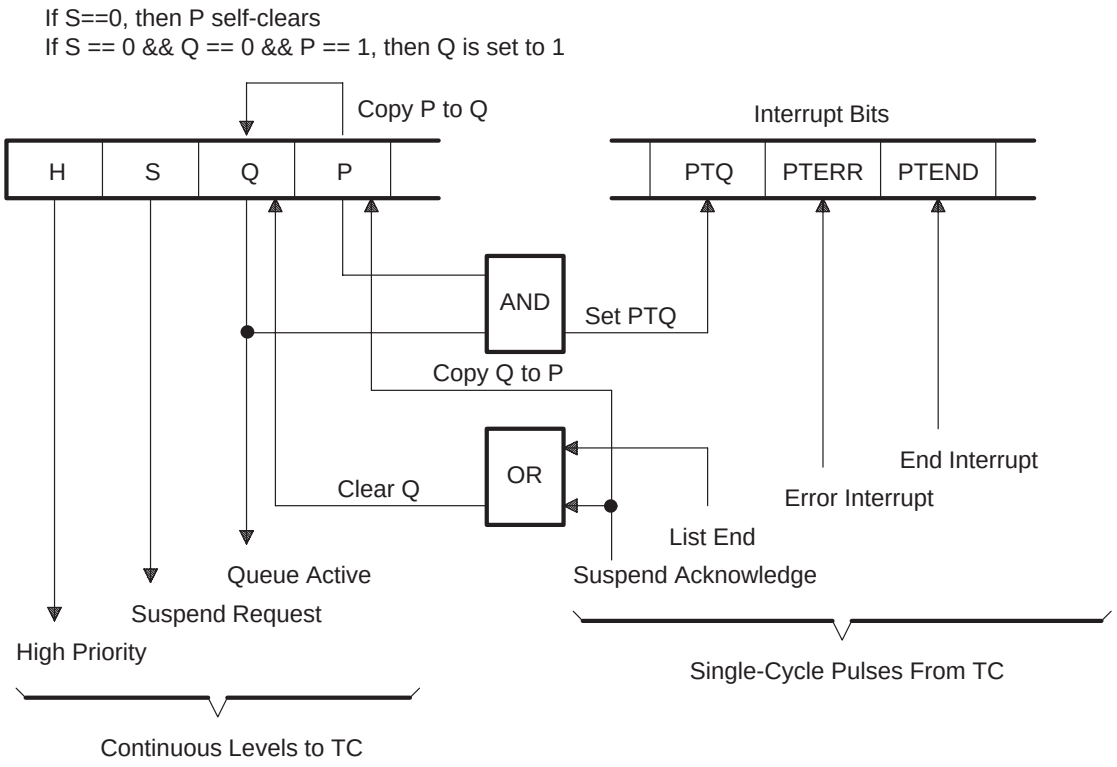


Figure 12–2. Packet Transfer Request Protocol

S	Q	P		
(a) A packet transfer request is submitted when no packet transfer request is queued.				
0	0	0	–	Nothing is active.
0	0	1	–	Software sets P.
				Single copy operation occurs if S = 0 and Q = 0.
0	1	0	–	P is copied into Q after one delay slot. P is cleared to 0. Queue is now active.
				TC sets Q to 0 after request is serviced (to either normal or error completion).
0	0	0	–	Nothing is active.
(b) A packet transfer request is submitted when a packet transfer request is already queued.				
0	1	0	–	Queue is active.
0	1	1	–	Packet transfer request queued.
				P is autocleared because S = 0.
0	1	0	–	Queue is still active. Processor receives PTQ interrupt.
(c) A suspend request is submitted when a packet transfer request is queued.				
0	1	0	–	Queue is active.
1	1	0	–	Software sets S (suspend request).
				TC copies Q to P and clears Q.
1	0	1	–	Linked list suspended. Queue was still active.
0	0	0	–	Software saves register, then sets S and P to 0.
(d) A packet transfer completes at the same time that a suspend request is submitted.				
0	1	0	–	Queue is active.
s/w	TC			TC completes service as software requests suspension.
1	0	0	–	Linked list suspended. Queue was completed. (TC will not issue a suspend acknowledge, so the Q bit is not copied to the P bit.)

12.3 Waiting for Packet Transfer Request Completion

In several situations, you must ensure that a previous packet transfer request has completed before proceeding. These include:

- When entering a routine that processes data brought into the shared RAMs by a packet transfer.
- When writing processed data to an output buffer that may contain prior data that is being written out to off-chip memory by a packet transfer.
- When writing to the linked-list address pointer.

There are two ways to verify that a packet transfer request has been serviced:

- Polling the Q bit or PTEND interrupt flag directly, or
- Using interrupt service routines associated with the PTQ and PTEND flags.

Combinations of polling and interrupt service routines are also possible.

12.3.1 Polling

Completion of a linked list of packet transfers can be checked by polling the Q bit of the comm register. You can check completion of one packet transfer within a linked list of packet transfers by setting the interrupt-when-finished (I) bit in the packet transfer options field and by polling the PTEND flag in the intflg register.

12.3.1.1 Q Bit Polling

The most straightforward way to test for completion of a linked list of packet transfers is to poll the queued packet transfer bit (Q) in the comm register. Example 12–2 shows a simple four-cycle routine that continues polling the Q bit until it is 0, which indicates that no packet transfers are queued for the given PP.

You can reduce this four-cycle polling loop to three cycles by performing the conditional branch on the basis of the polling result of the previous pass through the loop. The initial test is implemented before entry to the polling loop (retaining four instructions total). In some cases in which the zero status flag is known to be clear (set to 0) upon entry to the polling loop, the initial test can be omitted, thereby reducing the total number of instructions to three. See Example 12–3.

Note:

There is one delay slot between when a packet transfer is submitted by setting the P bit and when the Q bit is set. Thus, the first poll of the Q bit should not be performed on the cycle immediately following submission of a packet transfer.

Example 12–2. Polling for Completed Packet Transfer Request

```
Poll:  a15 = comm & 0x1<<29  ; Test Q bit (comm register bit 29)
       br =[nz] Poll         ; Continue in Poll loop as long as the Packet
                             ; Transfer Request is queued.
       nop                   ; Branch delay slot 1.
       nop                   ; Branch delay slot 2.
```

Example 12–3. Three-Cycle Polling for Completed Packet Transfer Request

```
              a15 = comm & 1<<29 ; Initial test of Q bit.
                                 ; NOTE:  This could be omitted if
                                 ; it is known that Z status bit
                                 ; is not currently set to 1.
       Poll3: br =[nz] Poll3     ; Continue in Poll loop as long
                                 ; as Packet Transfer Request is
                                 ; queued.
              nop                ; Branch delay slot 1.
              a15 = comm & 1<<29 ; Test for next pass.
```

12.3.1.2 PTEND Flag Polling

The PTEND interrupt flag (bit 19 of intflg register) is set each time a linked list of packet transfers is completed. It is also set when an individual packet transfer completes if the I bit (uppercase i, bit 28) of the packet transfer options field is set to 1 (see subsection 12.3.2.2). Thus, an alternative to polling the Q bit of the comm register is to poll the PTEND flag.

The advantage of polling the PTEND flag rather than the Q bit is that you can identify the completion of an individual packet transfer in the packet transfer linked list. The Q bit allows you to identify the completion of only the final packet transfer in a linked list.

The I bit in the packet transfer options field can act as a flag when it is not safe for the PP to continue execution beyond a certain point in its code until a specific packet transfer request in the packet transfer linked list completes.

Note:

A timing hazard is involved when the I bit is set in the packet transfer options of more than one packet transfer in a linked list of packet transfers. Since no count is kept of the number of times that the PTEND flag is set, you cannot distinguish either subsequent packet transfer completions in a linked list or the end of the complete linked list by checking the PTEND flag. Moreover, the other transfers could complete and set the PTEND flag before the first PTEND has been serviced and cleared.

12.3.2 Interrupts

Two interrupt flags provide an efficient alternative to polling in some situations:

- PTQ—packet transfer queued
- PTEND—packed transfer ended

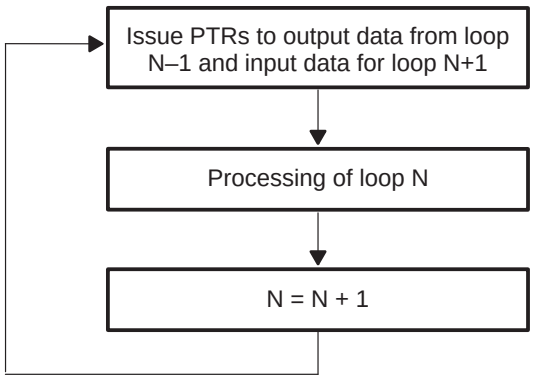
12.3.2.1 Submitting a Packet Transfer Request While a Previous Request Is Still Queued

The PTQ (packet transfer queued) flag (bit 17 of intflg register) becomes set if software writes a 1 to the P (packet transfer request) bit of the comm register when the Q (queue active) bit is already a 1. The PTQ interrupt can be used as an alternative to polling the Q bit before issuing a packet transfer request.

The PTQ interrupt is very useful due to the nature of the data flow for many of the targeted applications of the MVP. A typical flow of packet transfer requests for PP processing is shown in Figure 12–3. A linked list of packet transfers is issued to bring in the data for pass n+1 through the processing loop and transfer out the output data from pass n–1 from the processing loop before entering pass n of the processing loop.

Note that both input and output require double buffering. In a well-balanced routine, the processing loop requires somewhat more time than the servicing of the packet transfer requests by the TC. In a typical imaging routine, try to balance the load so that more than 90% of the time, the previous packet transfer request is already complete before the next one is ready to be submitted.

Figure 12–3. Typical Packet Transfer Request/Processing Loop Flow



Note: PTRs = packet transfer requests

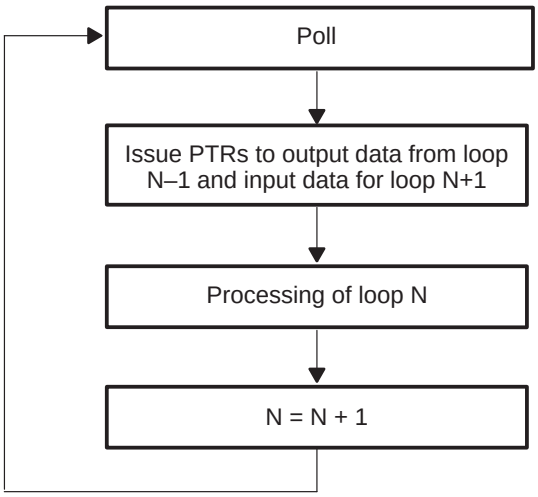
When you use the PTQ interrupt method of submitting packet transfers, you should set the linked list pointer to the next linked list of packet transfers by using the next-entry address of the final packet in the previous linked list of packet transfers, rather than overwriting the linked-list pointer directly. If you write directly to the linked-list pointer without first making certain that there is no previous packet in the queue, a timing hazard may exist: the pointer for the previous packet transfer request may be overwritten before it has been read by the TC. See subsection 12.3.2.1 for more information.

Cache misses can occur, other processors can submit packet transfer requests, and the order of packet transfer service is round-robin-based; these situations make it impossible to know beforehand exactly when a packet transfer will complete.

The PTQ flag allows you, while issuing each packet transfer request, to verify that the previous request has completed. If the previous packet transfer request is still queued, the PTQ flag is set, and the interrupt service routine is taken if its corresponding interrupt enable bit is set.

When the PTQ interrupt service routine executes, it requires more instructions and generally more cycles than direct polling, but it executes only when required. Direct polling, on the other hand, requires one pass through the polling loop even if the packet transfer request has already been serviced. This is shown in Figure 12–4.

Figure 12–4. Packet Transfer Request/Processing Flow With Polling



Note: PTRs = packet transfer requests

Example 12–4 shows a basic PTQ interrupt service routine. The pointer to this routine is located in the parameter RAM of the corresponding PP at address pba + 0x01C4 or 0x0100 #1C4 (# = PP number).

The interrupt service routine polls the Q bit of the comm register until it is equal to 0. When Q is equal to 0, this indicates that the previous packet transfer request has completed and is no longer queued.

At this point, the new packet transfer request can be resubmitted. Next, the PTQ interrupt flag is cleared. Finally, the standard return-from-interrupt sequence restores the status register, ipe, ipa, and pc values, respectively.

Note:

When a packet transfer request is submitted (by setting the P bit) with a packet transfer request already queued, three more instructions are executed before the PTQ interrupt is taken. Therefore, when you rely on PTQ interrupt to ensure completion of a prior packet transfer request, be sure that the three instructions following the instruction that issues the next packet transfer request do not require the previous packet transfer request to have been completed.

Example 12–4. Packet Transfer Queued Interrupt Service Routine

```
prbiv:
      a15 = comm & 1<<29        ; Test for queued packet
                                ; transfer request (PTR).

; Spin Until previous PTR is no longer queued.

poll: br =[nz] poll             ; Spin if PTR is still
                                ; queued.
      comm =[z] comm | 1<<28    ; Resubmit new PTR if
                                ; previous PTR is no longer
                                ; queued.
      a15 = comm & 1<<29        ; Test for queued PTR.

; After exiting spin loop ...

      intflg = 1<<17            ; Clear PTQ flag.

; Standard return–from–interrupt operation sequence

      reti1
      reti2
      reti3
      reti4
```

12.3.2.2 TC Interrupt of PP Upon Packet Transfer Completion

The packet transfer end (PTEND) interrupt flag (bit 19 of intflg) is set by the TC when one of the following conditions occur:

- The TC encounters the end of the PP's linked list
- The TC completes a packet transfer that instructs the TC to interrupt the requesting processor upon the transfer's completion.

If the I bit (uppercase i, bit 28) of the packet transfer options field is set to 1, it causes an interrupt to be sent to the processor that initiated the packet transfer, as soon as that transfer on the linked list has finished. This interrupt indicates only that the given entry (transfer) has completed, not that the linked list is done.

This interrupt allows the requesting processor to be flagged when a particular point in the linked list has been reached. **Note that PTEND can reliably detect only one set I bit in a linked list.** Since no count is kept of the number of times that the PTEND flag is set, a particular iteration cannot be distinguished by the PTEND flag. This is due to the fact that other packet transfers may compete and can set the PTEND flag before the first PTEND has been serviced and cleared.

It is conceivable that you could use a PTEND interrupt service routine to switch tasks on the PP until the packet transfer that is pending for the current task is completed. However, this is not likely to be practical, because the overhead for task switching is typically higher than the penalty of letting the PP wait in a polling loop until the packet transfer is completed. A more likely use for this interrupt is for the PP to send a message to the MP when a packet transfer is finished to signal that a region of memory (or resource) is available.

12.4 Dimensioned Transfers

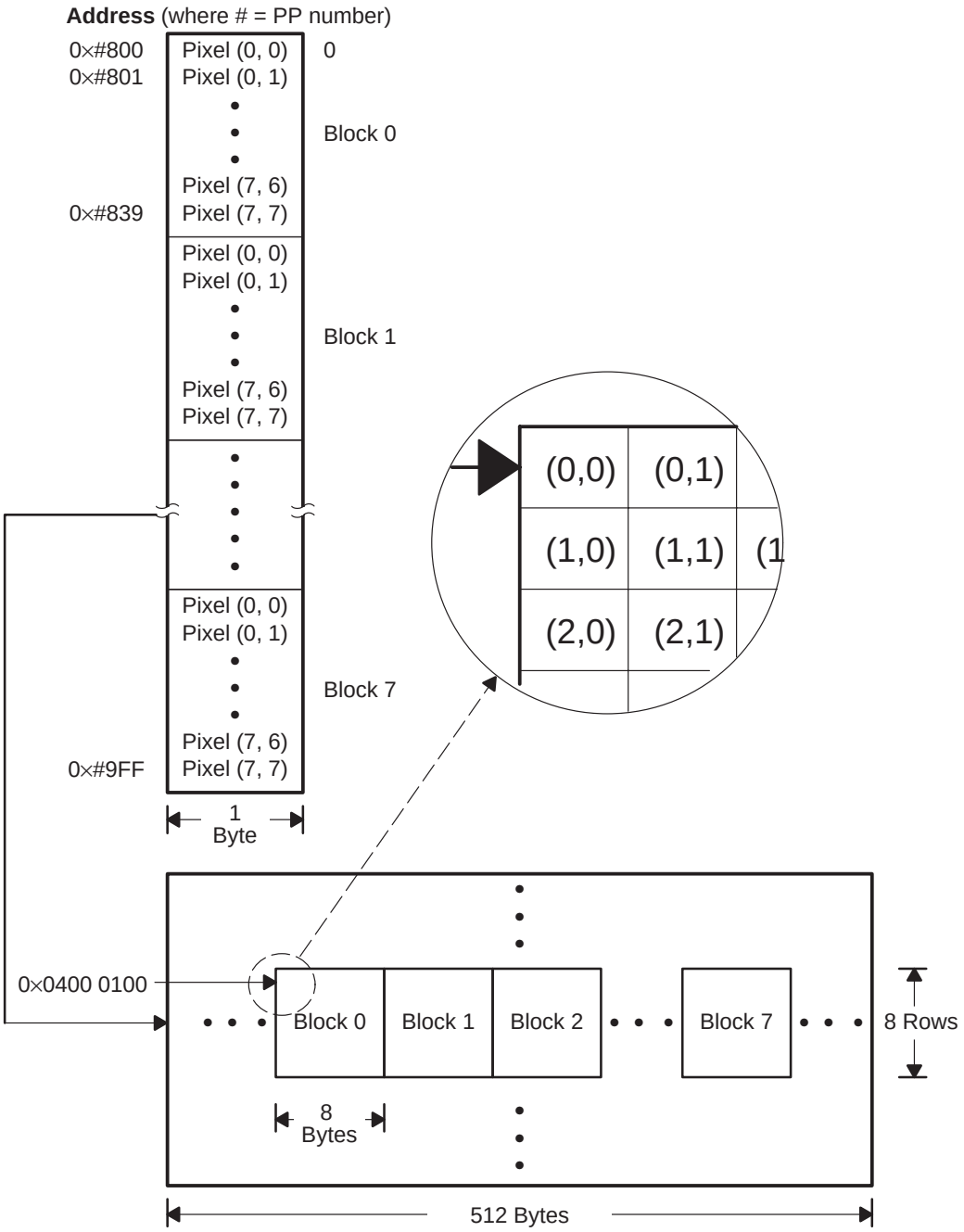
Dimensioned transfers describe sources and/or destinations that may be a simple contiguous linear sequence of data bytes or up to three dimensions of such regions. Data items along the first dimension are always one byte apart. The distance between entries in the second dimension is defined by the packet transfer parameter called B pitch. Likewise, the distance between entries in the third dimension is defined by the C pitch packet transfer parameter.

Dimensioned transfers are well-suited for imaging algorithms. Image-processing algorithms often process an entire image by rows, by columns, or by $N \times M$ blocks. This section describes how to set up a packet transfer of multiple 8×8 blocks of data from the shared RAMs to off-chip memory. A second example illustrates how to use linked packet transfers to output or input a partial or entire row of image data.

12.4.1 Dimensioned-to-Dimensioned Transfer Example

This subsection demonstrates how to set up a dimensioned-to-dimensioned packet transfer to transfer eight 8×8 blocks of byte-sized pixels, which have been processed by the PP, back to off-chip memory. The 8×8 blocks of data are stored linearly in the shared RAMs but must be output to off-chip memory with the associated pitches of the image, as shown in Figure 12–5. Although in this example the 8×8 blocks are contiguous in off-chip memory, this is not a requirement.

Figure 12–5. Transfer From 1-D to 2-D Space



12.4.1.1 Setup of Packet Transfer Parameters

The parameter template for a dimensioned-source-to-dimensioned-destination packet transfer is shown in Figure 12–6. Although for this example both the source and destination transfer modes are dimensioned, this is not a requirement. Many combinations of source and destination operating modes are available, as detailed in Appendix NO TAG, *Packet Transfer Parameter Tables*, in the *MVP Transfer Controller User's Guide*.

Figure 12–6. Dimensioned src and dst Packet Transfer Parameters (Big-Endian Format)

Byte Address	63 … 32	31 … 0		
base+56	Reserved			
base+48	Reserved			
base+40	src C Pitch		dst C Pitch	
base+32	src B Pitch		dst B Pitch	
base+24	src C Count		dst C Count	
base+16	src B Count	src A Count	dst B Count	dst A Count
base+8	src Start Address		dst Start Address	
base	Next Entry Address		PT Options	

In order to transfer the eight contiguous one-dimensional 64-byte sequences from the on-chip RAM to their corresponding two-dimensional 8×8 block in external memory, the packet request parameters for the template in Figure 12–6 are set up as follows:

- **Next Entry Address.** The content of the next entry address field is written to the linked-list pointer upon completion of the packet request. Since for this example the same packet request is performed repeatedly, the next entry address should point back to the same packet transfer.
- **PT Options.** The 32-bit packet transfer options field allows many variations in the way that data is transferred. For this example, both the src operating mode (bits 14–12) and the dst operating mode (bits 6–4) select the dimensioned transfer mode. Other bits in the packet transfer options can be used to select special access modes and automatic updates to the source and/or destination start address.
- **src Start Address.** The source start address points to the shared RAMs of the first pixel in Block 0 (see Figure 12–5). For this example, this is the base address for PP's local RAM1 (dba +0x0800). In a real application, the source start address might be passed as an argument to the packet transfer setup routine or loaded from memory.

- **dst Start Address.** The destination start address points to the destination location in off-chip memory of the first pixel in Block 0 (see Figure 12–5). For this example, that address is 0x0400 0100. In a real application, the destination location is typically passed as an argument to the packet transfer setup routine or loaded from memory.
- **src A Count.** The src A count is 512 because eight blocks of 64 byte-sized pixels must be transferred.
- **src B Count.** This is 0 and indicates that the second dimension is disabled for the source.
- **dst A Count.** For off-chip memory, the A count is 8.
- **dst B Count.** The 16-bit B count is the number of steps that should occur within the second dimension. For this example, the B count is 7, which corresponds to the number of rows in a block, minus one.
- **src C Count.** This is 0 and indicates that the third dimension is disabled for the source.
- **dst C Count.** This specifies the number of steps that should occur in the third dimension of the destination. For this example, this equals 7, which is the number of blocks, minus one.
- **src B Pitch.** This is don’t care because the second dimension isn’t active.
- **dst B Pitch.** This 32-bit field specifies the pitch of the second dimension of the destination. In this example, it is 512, which corresponds to the address delta between the first pixel of adjacent rows within each block.
- **src C Pitch.** This is don’t care because the third dimension isn’t active.
- **dst C Pitch.** This 32-bit field specifies the pitch of the second dimension of the destination. For this example, the C pitch is 8, which is the off-chip memory address delta between the upper left corner pixels in adjacent 8×8 blocks.

Example 12–5 shows PP code that sets up the packet transfer parameters described in the preceding text. The same setup code is shown in Example 12–6 and uses the packet transfer structure defined in the include file packetpp.i.

Example 12–5. PP Packet Transfer Parameters Set-Up Code

```
d7 = 0x1\\31                    ; Prepare PT_Options .
  ||*(a8 = pba + 0x200) = a8    ; Set Next–Entry Address to itself.
*(a8+[1])     = d7              ; Dimensioned–to–Dimensioned transfer
                                ; with stop bit set.
d7 = &*(dba + 0x800)
*(a8+[2])     = d7              ; src Start Address = dba + 0x800
d7 = 0x4000100
d7 = 0x1\\10                    ; d7 = 512
  ||*(a8+[3]) = d7              ; dst Start Address = 0x4000100
*(a8+[4])     = d7              ; src B Count = 0, A Count = 512.
d7 = 0x70008
*(a8+[5])     = d7              ; dst B Count = 7, A Count = 8.
d7 = 7
  ||*(a8+[6]) = a15             ; Source C Count = 0.
d7 = 0x1\\10                    ; d7 = 512
  ||*(a8+[7]) = d7              ; dst C Count = 7.

; NOTE: Since 2nd and 3rd Dimension for the source transfer aren't
; active, they do not need to be set.

d7 = 8
  ||*(a8+[9]) = d7              ; dst B Pitch = 512.
*(a8+[11])    = d7              ; dst C Pitch = 8.
```

It is recommended that you set up the packet transfers by using structures such as those shown in Example 12–6. This makes code easier to read and also minimizes errors caused by accidentally using the wrong offset.

The packetpp.i file is shown in Appendix B, *packetpp.i Include File*. It contains two structures used for setting up packet transfers (in big-endian format). The first structure, with sPT_ or ePT_ prefixes, is used in Example 12–6. It defines the base or origin of the structure at the start of the packet transfer parameters.

This is convenient when the base address of the packet transfer parameters must be written to the linked-list pointer and/or is written to the next entry address parameter of the packet transfer. A second packet transfer structure defined in packetpp.i uses the .access directive to set the base of the structure in the middle of the packet transfer parameters. This allows any packet transfer parameter to be accessed via a short (3-bit) offset for tighter packing of code.

Example 12–6. PP Packet Transfer Parameters Setup Code (Using Structures)

```
   .include "packetpp.i"

d7 = 0x1\\31                    ; Prepare PT_Options.
  ||*(a8 = pba + 0x200)  = a8   ; Set Next-Entry Address to itself.
*a8.sPT_Options    = d7         ; Dimensioned-to-Dimensioned transfer
                                ; with stop bit set.
d7 = &*(dba + 0x800)
*a8.sPT_SrcStartAddress = d7    ; src Start Address = dba + 0x800
d7 = 0x4000100
d7 = 0x1\\10                    ; d7 = 512
 ||*a8.sPT_DstStartAddress = d7 ; dst Start Address = 0x4000100
*a8.sPT_SrcBACount = d7         ; src C Count = 0, A Count = 512.
d7 = 0x70008
*a8.sPT_DstBACount = d7         ; dst B Count = 7, A Count = 8.
d7 = 7
 ||*a8.sPT_SrcCCount  = a15     ; Source C Count = 0.
d7 = 0x1\\10                    ; d7 = 512
 ||*a8.sPT_DstCCount  = d7      ; dst C Count = 7.

; NOTE: Since 2nd and 3rd Dimension for the source transfer
; aren't active, they do not need to be set.

d7 = 8
 ||*a8.ePT_DstBPitch  = d7      ; dst B Pitch = 512.
*a8.ePT_DstCPitch  = d7         ; dst C Pitch = 8.
```

12.4.2 Using the Double-Buffer Transfer Model

In general, you should double-buffer data so that packet transfers between off-chip memory and the MVP's on-chip RAMs occur in parallel with the PP's data processing and without causing excessive crossbar contention. Double-buffering allows both the TC and PP to operate at maximum efficiency.

In typical double-buffering, you assign one of the shared RAMs to the TC for an output/input process and another shared RAM or two to the PP's compute process. This allows the result of the previous compute process to be output to off-chip memory and the next block of data in off-chip memory to be input to the on-chip shared RAM assigned to the TC; the PP concurrently processes the current data in other RAM(s).

At the conclusion of this stage, the RAM assignments switch so that the RAM that contains the newly processed output data is assigned to the TC and the RAM that contains the newly loaded data is assigned to the PP for processing.

An example of double-buffered processing is shown in subsection 12.4.3.

12.4.3 Row-Based Processing: Robert's Edge Detection Example

By transferring entire or partial rows, you can efficiently manage various image-processing algorithms that operate on small blocks (kernels). This subsection shows how this is done for the Robert's Edge Detection, which uses a 2×2 block.

Since Robert's Edge Detection operates on 2×2 blocks of pixels, data from two rows is required to compute each output pixel. In order to reduce the number of packet transfer requests that must be issued, two rows of image data are always maintained in the shared RAMs. While data from two rows is processed by the PP, the TC, in parallel with the PP, first outputs a processed row of data and then inputs a new row of data. This process is summarized as follows:

- Initially, two entire rows (or partial rows, if the row width is greater than 1K bytes) are brought into the shared RAMs.
- While the first edge-enhanced output row is computed, the third row in the image is input to the shared RAMs.
- Next, while the second output row is computed, the first edge-enhanced row of pixels is transferred to off-chip memory, and the fourth row in the image is brought into the shared RAMs.

This represents essentially a pipelining of three stages to achieve maximum performance: transferring a row of data from off-chip memory into the shared RAMs, processing on the row of data, and transferring an output row of data from the shared RAMs to off-chip memory. These three stages are referred to as input, compute, and output in Figure 12–7.

Figure 12–7. Input, Compute, Output Pipeline Stages

Time	0	1	2	3	. . .	N–2	N–1	N	N+1
Input	1,2	3	4	. . .		N			
Compute		1	2	. . .		N–2	N–1,N		
Output			1	. . .		N–3	N–2	N–1,N	

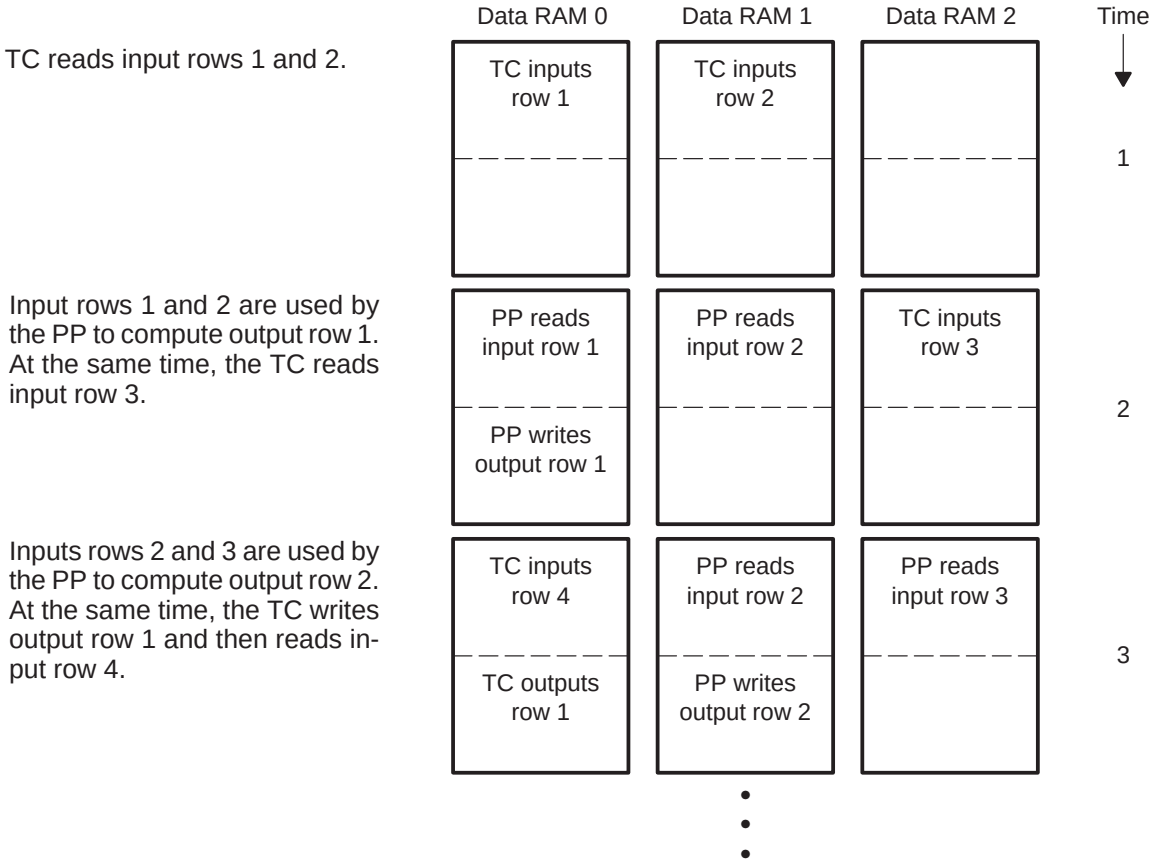
For this example, the output packet transfer for row n–1 and the input packet transfer for row n+2 can be linked together so that when the output packet transfer is completed, the input packet transfer is serviced automatically.

Memory is allocated as shown in Figure 12–8 to eliminate the possibility of contention between TC crossbar accesses and PP crossbar accesses. One of the three local data RAMs is always dedicated exclusively to the TC.

The linked output and input packet transfers are written to the upper and lower halves, respectively, of the same RAM module. Since the two packet transfers are sequential (that is, they do not overlap in time), they do not contend with each other. The PP accesses the same RAM for both row N input data and row N output data. However, for this algorithm, these accesses are not required to occur in parallel; therefore, no contention occurs.

In this example, the input and output areas are in different parts of a RAM. However, if the input data is no longer needed, the output data could write over the input data, increasing the maximum row size that will fit into a RAM.

Figure 12–8. Sample Memory Allocation (Double Buffering)



For kernels requiring more rows of data (for example, Sobel Edge Detection requires three rows), the number of input buffers required increases, and the Robert's Edge Detection method of data management becomes less feasible. At some point, it becomes more practical to deal with blocks or patches of data instead of rows of data.

The description of rows could just as easily apply to columns of data. However, it is more efficient, when possible, to transfer rows of data because a row of data typically resides in the same page of off-chip memory; a column of data is almost certainly not contained in a single page.

12.5 Guided Transfers

Guided transfers (source and/or destination) are those in which the address sequence is guided by an on-chip table, rather than calculated solely from the packet transfer parameters.

A base address is contained in the packet transfer parameters. Actual addresses are generated from this base address with either an offset or delta table (located in the shared RAMs), depending on the type of guided transfer indicated by the PT options.

Guided transfers are often better-suited than dimensioned transfers for graphics and image analysis algorithms because the required data is frequently not contiguous in memory and there is no fixed address delta between accesses. Guided transfers are typically used for such things as line draw operations (with optional features like antialiasing and patterning) and trapezoid fills.

The main requirement for guided transfers is that the guided path must be established before the transfer is performed. Therefore, the PPs (or MP) must compute the guide table before submitting the packet transfer request.

Often, in practice, the processor that creates the guide tables and the TC can be active simultaneously. The processor that created the guide table can submit a guided transfer request and then go on to prepare the next guide table while waiting for an interrupt to indicate the previous transfer's completion.

The actual data processing or analysis then begins on the portion of data in the on-chip RAM, while the next segment is being transferred in by the TC. Once the required processing has been performed, the same guide table that was used for transferring the data on-chip is often used to perform a guided transfer of the processed data back to off-chip memory.

There are two classes of guided transfers:

- In **fixed-patch** guided transfers, the first and second dimensions are determined by the packet transfer parameters (much like a dimensioned transfer), but the third dimension is guided from entries in the on-chip table.
- **Variable-patch** guided transfers are similar to fixed-patch, except that the sizes of the first and second dimensions are specified in the guide table instead of in the packet transfer parameters. Variable-patch guided transfers are effective for functions such as a trapezoidal fill.

The following subsections illustrate how to use a fixed-patch delta-guided transfer to draw a random line. For more details on guided transfers, refer to Section NO TAG, *Guided Transfers*, in the *MVP Transfer Controller User's Guide*.

12.5.1 Line Draw Using Fill-With-Value

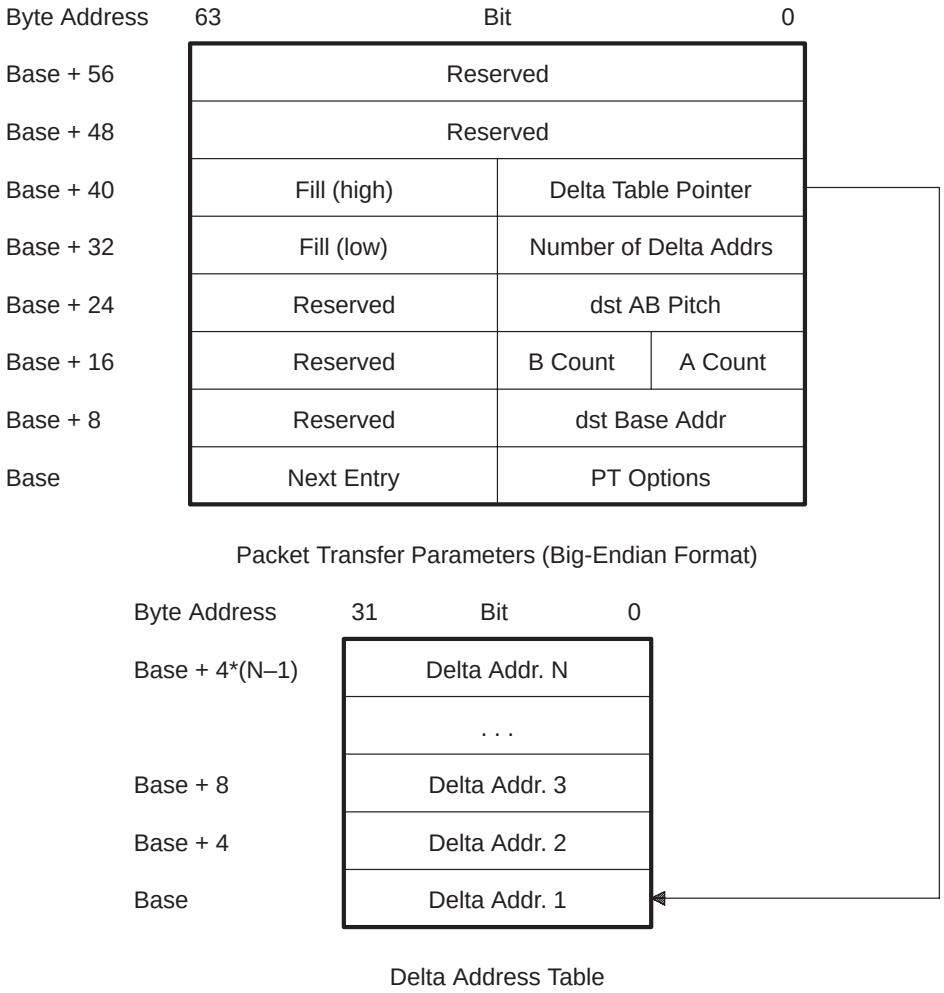
For simply drawing a random line with a fixed color in an image, the TC can write to the appropriate pixels in off-chip memory without ever bringing the pixels into the MVP. This is done with a special packet transfer feature referred to as fill-with-value. (See subsection 15.1.3.1, *Solid Fills*, for an example of a fill-with-value source packet transfer.)

The following steps are required to unconditionally draw a random line using the Bresenham technique:

- Generate a table of delta address (DA) values
- Update the packet transfer parameter template
  - Number of guide-table entries
  - Pointer to guide-table
  - PT options (fill-with-value to fixed-patch delta-guided)
  - Starting address for the line
  - Fill value
- Poll until the PP does not have a queued packet transfer request
- Issue the packet transfer request to the TC (once the TC has no queued packet transfer request for this PP)

A fill-with-value to fixed-patch delta-guided packet transfer, as shown in Figure 12–9, writes a fixed fill value to each output pixel in the requested line. Special cases for horizontal, vertical, and diagonal lines may optionally use the fill-with-value to dimensioned-packet transfer form to avoid the delta address calculations.

Figure 12–9. Fill-With-Value to Fixed-Patched Delta-Guided Packet Transfer



12.5.1.1 Fixed-Patch Delta-Guided Address Usage

The packet transfer in Figure 12–9 writes the fill value to pixel addresses as shown here:

```
Address(pixel 1) = delta address 1 + Base Address
Address(pixel 2) = delta address 2 + Address(pixel 1)
Address(pixel 3) = delta address 3 + Address(pixel 2)
                 . . .
Address(pixel N) = delta address N + Address(pixel N-1)
```

12.5.1.2 Pixel Fill Value

The fill value that is used to overwrite the old pixel value must be replicated to use all 64 bits in the packet transfer parameter words.

- If the pixel is a byte, the fill value is repeated eight times.
- If the pixel is a halfword, the fill value is repeated four times.
- If the pixel is a word, the fill value is repeated two times.

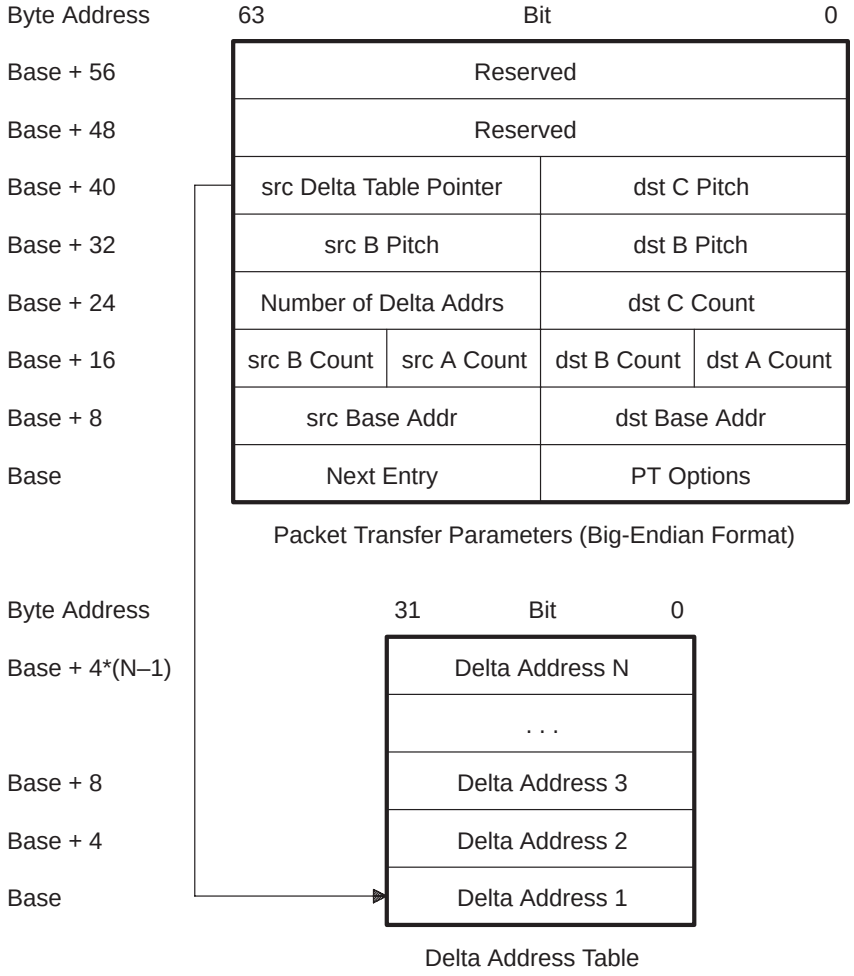
12.5.2 Line Draw With Pixel Processing

The previous example of fill-with-value to fixed-patch delta-guided packet transfer drew lines of solid color. A more general line draw would use a fixed-patch delta-guided source and a dimensioned destination in the shared RAMs (see Figure 12–10) to bring the pixels linearly into the MVP. Pixel-processing operations are performed on the pixels in the line. Then the processed line can be transferred back to off-chip memory.

A general line-draw sequence is illustrated below:

1) Determine the source input delta-guided address values.
2) Store the parameters for the fixed-patch delta-guided to dimensioned packet transfer.
3) Issue the input packet transfer request to TC.
4) Operate on the pixel data after it becomes available in the shared RAMs.
5) Toggle the bit in PT options to reverse source and destination parameters.
6) Issue the output packet transfer request to the TC.

Figure 12–10. Fixed-Patched Delta-Guided Source to Dimensioned Destination Packet Transfer



Chapter 13

Advanced EALU

This chapter describes the bit fields in d0 that define the EALU function code (the A bit and the EALU_FC field), the bits that determine the carry-in behavior and sign extension (CISE), and a bit that controls in the input to the C port of the ALU. The d0 fields that specify options along the multiplier data path are described in subsection 8.5.2, *Multiplying in Parallel With an EALU Operation*.

As described Section 8.4, *The Extended Set of ALU Operations (EALU)*, the assembler can derive the 32-bit constant required in the d0 register for you. However, the descriptions of the EALU_FC and CISE bit fields in this chapter will give you a better understanding of the operations available with an EALU.

Topics

13.1 The EALU Function Code . . . . . PP: 13-2

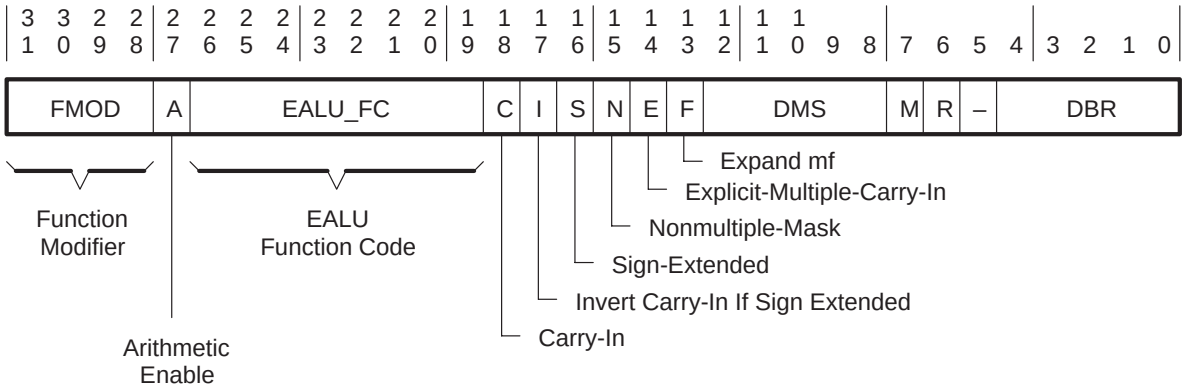
13.2 EALU Carry-In and Sign Extension . . . . . PP: 13-7

13.1 The EALU Function Code

The A bit and the EALU_FC field, shown in Figure 13–1, specify the function performed by the ALU for an EALU operation.

- **FMOD.** The FMOD field contains an arithmetic function modifier, as described in subsection 8.3.4, *ALU Function Modifiers*. In ealu operations, function modifiers can apply to Boolean operations in addition to arithmetic operations. Multiple arithmetic, bit detection, and other modifications to the ALU data path are available with FMODs.
- **Arithmetic Enable.** The A bit enables (A = 1) or disables (A = 0) the carry-generates in the ALU and therefore determines whether the performed EALU is an arithmetic or bitwise Boolean operation.
- **Extended ALU Function Code.** Bits 26–19 specify the starting value for the ALU function code for EALU operations. If an ealuf operation is specified, the ALU function code is the 1s complement of these bits. These bits will become the ALU's F7–F0 signals but may be modified enroute as a result of the chosen FMOD code or the S bit.

Figure 13–1. d0 Fields Specifying the EALU Operation



Example 13–1 contains the basic equations that define the operation of the PP ALU. The F0–F7 function signals in the equation are specified directly in the EALU_FC field in the d0 register.

The propagate term, when set, causes a carry-in to a bit to become a carry-out from the bit. The propagate term for a PP ALU operation is defined by a sum of Boolean products, shown in Table 13–1, which are individually either enabled or disabled by a corresponding ALU function code.

The generate term is what distinguishes Boolean operations from arithmetic operations. The effect of the arithmetic (A) bit is to either enable or disable the effect of generate on the carry-out. When the generate term is enabled (A = 1) and is true, then a carry-out is generated, regardless of the carry-in.

For EALU operations, these eight function codes are derived from bits 26–19 of d0. If an ealuf operation is executed, the 1s complement of each of bits 26–19 is formed before the operation proceeds to the FMOD/S-bit logic that may further alter its value. The following two sections describe how these functions codes are used to set up Boolean and arithmetic operations.

Example 13–1. Basic Equations for the PP ALU

```
Propagate = F0&(~A&~B~&~C) | F1&(A&~B&~C) | F2&(~A&B&~C) | F3&(A&B&~C)
            F4&(~A&~B~&C)  | F5&(A&~B&C)  | F6&(~A&B&C)  | F7&(A&B&C)

Generate  = ~Propagate & A & (F0&~B&~C | F2&B&~C | F4&~B&C | F6&B&C)
          = A & (F0&~F1&~B&~C | F2&~F3&B&~C | F4&~F5&~B&C | F6&~F7&B&C)

Result    = Propagate ^ carry-in

carry-out = (Propagate & carry-in) | (Generate & Arithmetic)
```

Table 13–1. The ALU's Boolean Products

d0 Bit	Opcode Bit	ALU Function Signal	Products
26	58	F7	A & B & C
25	57	F6	~A & B & C
24	56	F5	A & ~B & C
23	55	F4	~A & ~B & C
22	54	F3	A & B & ~C
21	53	F2	~A & B & ~C
20	52	F1	A & ~B & ~C
19	51	F0	~A & ~B & ~C

13.1.1 ALU Boolean Operations

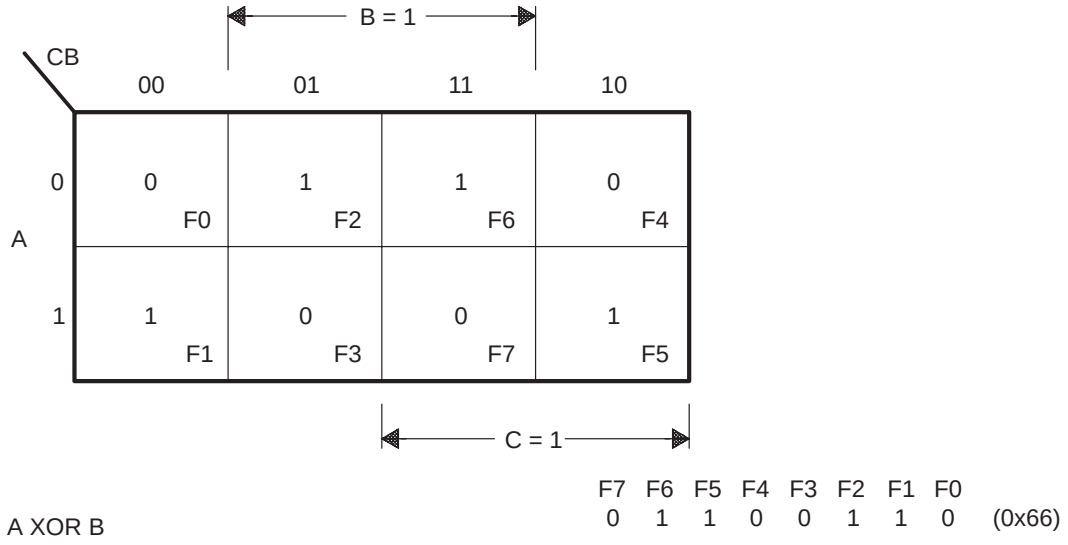
The ALU performs Boolean operations when the arithmetic enable bit (A) is equal to 0. Boolean operations are described completely by the eight function control signals labeled F7–F0 and shown in Table 13–1. The ALU function is defined by the sum of the products defined by the control signals (F0|F1|F2|F3|F4|F5|F6|F7); therefore, it is possible to specify all 256 logical combinations of three inputs, as well as all sixteen two-input Booleans.

Table 13–1 can be used to derive the coding for the desired Boolean operation. For example, consider the simple case in which the desired operation is a single three-input AND expression such as A&~B&C. To specify this operation to the ALU, set F5 to 1 and all other function signals to 0. Consequently, the required function code in the EALU_FC field in d0 is 0x20.

To derive the ALU function code for more complicated Boolean expressions, you can easily determine the required function signal values by using a Karnaugh map. For example, the truth table for A ∧ B (A XOR B) is shown in Figure 13–2.

Note that each block in the truth table corresponds directly to one of the ALU function codes. (for example, CBA = 110 corresponds to F6, etc.). Therefore, from the truth table in Figure 13–2, you can read directly that the EALU code in bits 26–19 is 0x66.

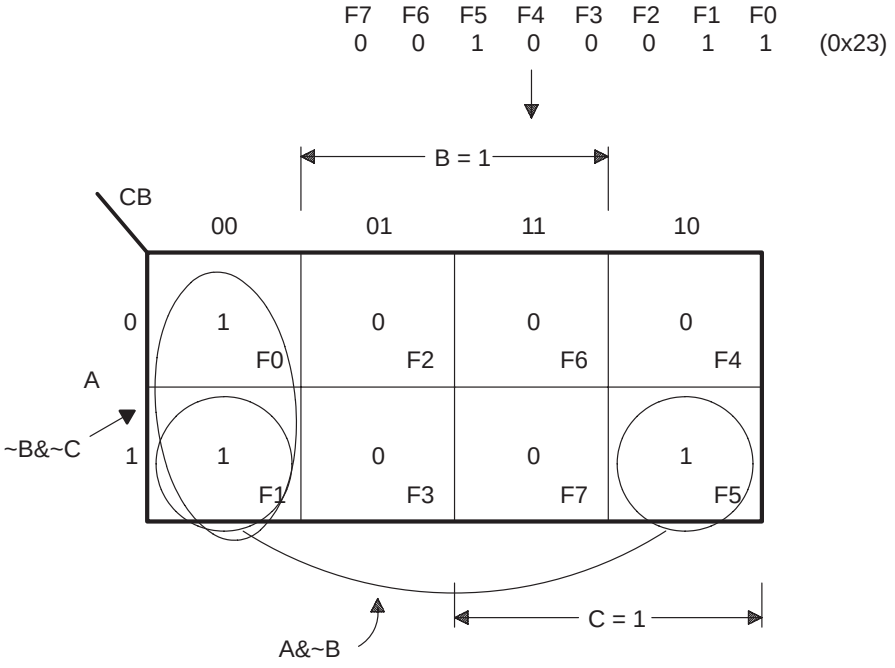
Figure 13–2. Truth Table Derivation of A XOR B Function Coding



Karnaugh maps are also useful for determining the function performed by a given function code, as shown in Figure 13–3.

The CISE bits described in Section 13.2 apply for Boolean operations, as well as arithmetics. For a standard Boolean operation, the C bit that determines the carry-in to bit 0 should be 0. However, it is possible to use the C bit to specify a carry-in to the LSB of a Boolean operation.

Figure 13–3. Determining Boolean Operation Performed by Function Code 0x23



Performed Operation Is:

A&~ B | ~B&~C

13.1.2 EALU Arithmetic Operations

When A = 1, the ALU carry-generate is enabled, and the EALU operation is arithmetic, rather than from Boolean. For arithmetic operations, just as for Boolean, there are 256 function code combinations. In Section 8.4, *The Extended Set of ALU Operations (EALU)*, the function codes for the some of the arithmetic operations were shown but not explained in detail. This section describes how to determine the ALU function code for arithmetic operations.

For a simple add, the propagate is an XOR of two inputs. The generate term is the AND of both inputs. The function A + B is defined by the equations in Example 13–2.

To set up the PP ALU for an A + B operation, the EALU_FC field in d0 should be set to 0x66 (for A ∧ B, as shown in Figure 13–2). The C bit, which determines the carry-in to bit 0, should be set to 0.

In the ALU, a subtract is performed by adding the 2s complement of one of the inputs to the other input. To obtain the 2s complement, one of the inputs is inverted and then a carry is forced into the LSB to obtain the 2s complement from the inverted input. Thus, the function A – B is defined as shown in Example 13–3.

The EALU_FC field in d0 is set to 0x99 (A ∧ ~B), and the C bit must be set to 1 to force a carry-in into bit 0.

Example 13–2. A+B ALU Terms

```
Propagate = A ^ B
Generate  = A & B
Result    = Propagate ^ carry–in
carry–out = Generate | (Propagate & carry–in)
```

Example 13–3. A–B ALU Terms

```
Propagate = A ^ ~B
Generate  = A & ~B
Result    = Propagate ^ carry–in
carry–out = Generate | (Propagate & carry–in)
```

13.2 EALU Carry-In and Sign Extension

For the base set of ALU operations, Booleans have a carry-in of 0, adds have a carry-in of 0, and subtracts have a carry-in of 1 into bit 0.

Base set ALU operations that involve sign extension or absolute value are performed by special logic associated with certain ALU functions that manipulate the ALU function according to the sign bit of the input to the barrel rotator.

For EALU operations, you configure carry-in, sign extension, and absolute value directly by using the CISE bits in d0 (see Figure 13–1). The function of each of these bits is described in the following subsections.

13.2.1 Carry-In Starting Value (C Bit)

For EALU operations, the C bit contains the starting value for the ALU's bit 0 carry-in. The C bit is equally applicable to Booleans and arithmetic operations. However, for Booleans, it is normally 0, and for arithmetic operations, it is typically 0 for adds and 1 for subtracts. The effect of setting C to a 1 for a subtract operation is to convert a 1s complement input into the corresponding 2s complement number. This allows the ALU to perform a subtract operation by adding the 2s complement of one of the inputs.

The C bit can effectively add a 1 to the result of an add, or subtract a 1 from the result of a subtract. For example, if the function code in EALU_FC is 0x66, as described above for the function A+B, but the C bit is set to 1 instead of 0, the operation A+B+1 is performed by the EALU (without requiring an operand of 1 to be specified). Likewise, if you use the EALU function setup for A–B, as described above, but set the C bit to 0 instead of 1, the operation A–B–1 is performed by the EALU.

13.2.2 Invert Carry-In (I Bit)

The invert carry-in bit is used in conjunction with the S bit (described in the next section) to specify operations involving the absolute value of the input to the B port. When both I and S are set to 1, if the sign bit of the input to the barrel rotator is set (the number is negative), then the carry-in starting value specified by the C bit is inverted and the lower four ALU function code bits are also inverted. Example 13–4 shows how this behavior allows absolute value operations to be performed.

Example 13–4. Using I and S Bits for A – |B|

```
EALU Set Up for A + (B&C | –B&~C)
---------------------------------

EALU: 0x69         A ^ (B&C | ~B&~C)
C:  1
I:  1
S:  1
E:  0
F:  0
Input to C port:   0x0

If input to barrel rotator is non–negative:

   A + (B&C | –B&~C)  →  A–B

  else (i.e., input to barrel rotator is negative):

   EALU: inverted to 0x66   A^B
   C:    inverted to 0
                         →  A+B

Thus, this EALU set–up acts like the following
 which is equivalent to A–|B|:

   if (B>=0)
    A–B
   else
    A+B
```

13.2.3 Sign-Extend Selection (S Bit)

The S bit supports operations involving a signed shift right. It can also be used in conjunction with the I bit to specify absolute value operations, as described in subsection 13.2.2.

If either the E bit or the A bit is 0, and the S bit and the sign of the data going to the barrel rotator are both 1, then the four LSBs (22–19) of the ALU code extracted from d0 are inverted. Example 13–5 shows how this behavior can be used to specify a signed right shift operation.

Example 13–5. Using S Bit for Signed Right Shift

```
d0 is configured as follows:

EALU Set Up for A + (B&C)
-------------------------
FMOD:  one of the function modifier codes for %!
A:     1
EALU:  0x6a      A^(B&C)
C:     0
I:     0
S:     1
E:     0
F:     0

If the input to the barrel rotator is negative, the four LSBs of function code are
inverted:

   0x6a –> 0x65     A^(B|~C)
     which for a carry–in of 0 is equal to
             A+(B|~C)

To perform a signed right shift, the input to the C port must be a mask derived
from the barrel rotate amount. Assuming a MPY||EALU is being performed, the barrel
rotate amount is specified in d0 and the C port input is %d0. Thus, the operation
acts like:

If input to barrel rotator is nonnegative,

   A + (src\\d0 & %!d0)    ; The & %!d0 operation masks off the MSBs containing
                           ; bits that wrapped around from the LSBs
 else
    A + (src\\d0 | ~%!d0)  ; The | ~%!d0 operation puts 1s in the MSBs containing
                           ; bits that wrapped around from the LSBs
```

13.2.4 Explicit-Multiple Carry-In (E Bit)

The E bit allows carry-ins for multiple arithmetic to be determined at runtime. This, in turn, supports specifying either an add or subtract independently for each portion of the ALU according to the input data to the C port.

If the E and A bits are both 1, and the FMOD code is not cin, then the effects of the S, I, and C bits are annulled. The carry-in information into the LSB of each piece of the ALU is, instead, made equal to the LSB of the corresponding piece of the C port XORed with the ALU's F0 signal. Example 13–6 illustrates how to perform a subtract in the upper halfword and an add in the lower halfword by using the E bit.

Note:

The E bit should not be set to 1 when a bit-detection operation is specified by the function modifier code in d0.

If multiple operation is not selected by the FMOD code, the carry-in to bit 0 of the 32-bit ALU is C port bit 0 XORed with F0.

Example 13–6. Using E Bit for Mixed Multiple Operations

```
EALU Set Up for A + (B&C | –B&~C)
---------------------------------

FMOD:  one of the multiple arithmetic options
A:     1
EALU:  0xd9         A ^ (B&C | ~B&~C)
C:     don't care
I:     don't care
S:     don't care
E:     1
F:     0

Assuming the input to the C port is %16 (0x0000 FFFF),

       carry–in to bit 0 is:
          C port bit 0 ^ F0  = 1 ^ 1 = 0 (correct carry–in for add)
       carry–in to bit 16 is:
          C port bit 16 ^ F0 = 0 ^ 1 = 1 (correct carry–in for subtract)

This is equivalent to

       A + (B&%16 | –B&~%16)

which is equivalent to

       A+B   for lower halfword
       A–B   for upper halfword
```

13.2.5 Expanded Multiple Flag Option (F Bit)

The F bit chooses between the output of the mask generator (F = 0) and the expanded mf register (F = 1) as the input to the ALU's C port. This option supports using the expander in EALU operations with either the parallel multiply or parallel rotate. The F bit is ignored in the EALU||ROTATE case when src4 goes directly to the C port of the ALU instead of through the mask generator.

13.2.6 Summary of Carry-In Determination

The carry-in for an EALU operation is determined by the FMOD code and the CISE bits. If the FMOD code is set for cin, then the CISE bits are ignored, and the carry-in to the ALU is the carry bit in the status register.

Here is a summary of the EALU carry-in behavior:

- If FMOD code is cin, then the carry-in to bit 0 of the ALU is taken directly from the C bit in the status register without modification. (ealuf does not invert the value.)
- If FMOD code is not cin and either the E bit or A bit is 0:
  - The carry-in starting value is specified by the C bit in d0.
  - For ealuf operations, the carry-in starting value is inverted on the way into the ALU.
  - If the S and I bits are 1 and the sign bit of the input to the barrel rotator is 1, the carry-in starting value (or the inverse of the carry-in starting value for ealuf operations) is inverted to form the final carry-in input to the ALU.
  - If a multiple operation is specified by the FMOD code, the carry-in value supplied to bit 0 of the ALU is used as the carry-in to each piece of the ALU.
- If FMOD code is not cin and the E and A bits are both 1:
  - The carry-in into the LSB of each piece of the ALU is the LSB of the corresponding piece of the C port XORed with the ALU's F0 signal (bit 19) derived from d0.
  - If multiple operation isn't specified by the FMOD code, the carry-in to bit 0 of the ALU is C port bit 0 XORed with F0.

Table 13–2 describes the carry-in and inversion of the four LSBs of the ALU function code (used for sign extension). It also lists the possible combinations of the settings for the d0 bits that affect carry-in and the sign of the barrel rotator input.

Table 13–2. EALU Carry-In and Sign-Extension Specification

Opcode	d0 Bit Settings						Rotator Input Sign	Carry-In	Invert Value of Extracted d0 (22–19)?
	cin FMOD	A	E	F	S	I			
ealu	no	X	0	0	0	X	X	d0(C)	no
ealu	no	X	0	0	1	0	0	d0(C)	no
ealu	no	X	0	0	1	0	1	d0(C)	yes
ealu	no	X	0	0	1	1	0	d0(C)	no
ealu	no	X	0	0	1	1	1	!d0(C)	yes
ealu	no	0	1	0	0	X	X	d0(C)	no
ealu	no	0	1	0	1	0	0	d0(C)	no
ealu	no	0	1	0	1	0	1	d0(C)	yes
ealu	no	0	1	0	1	1	0	d0(C)	no
ealu	no	0	1	0	1	1	1	!d0(C)	yes
ealu	no	1	1	0	X	X	X	C(n)^F0	no
ealu	yes	X	X	0	0	X	X	sr(C)	no
ealu	yes	X	X	0	1	0	0	sr(C)	no
ealu	yes	X	X	0	1	0	1	sr(C)	yes
ealu	yes	X	X	0	1	1	0	sr(C)	no
ealu	yes	X	X	0	1	1	1	sr(C)	yes
ealuf	no	X	0	0	0	X	X	!d0(C)	yes
ealuf	no	X	0	0	1	0	0	!d0(C)	yes
ealuf	no	X	0	0	1	0	1	!d0(C)	no
ealuf	no	X	0	0	1	1	0	!d0(C)	yes
ealuf	no	X	0	0	1	1	1	d0(C)	no
ealuf	no	0	1	0	0	X	X	!d0(C)	yes
ealuf	no	0	1	0	1	0	0	!d0(C)	yes
ealuf	no	0	1	0	1	0	1	!d0(C)	no
ealuf	no	0	1	0	1	1	0	!d0(C)	yes
ealuf	no	0	1	0	1	1	1	d0(C)	no
ealuf	no	1	1	0	X	X	X	C(n)^F0	yes
ealuf	yes	X	X	0	0	X	X	sr(C)	yes
ealuf	yes	X	X	0	1	0	0	sr(C)	yes
ealuf	yes	X	X	0	1	0	1	sr(C)	no
ealuf	yes	X	X	0	1	1	0	sr(C)	yes
ealuf	yes	X	X	0	1	1	1	sr(C)	no

Notes on columns in Table 13–2:

- **Opcode**—Indicates whether the opcode specifies ealu or ealuf.
  - ealu—ALU function based on bits 26–19 of d0.
  - ealuf—ALU function based on the 1s complement of bits 26–19 of d0.
- **cin FMOD**—Indicates whether the ALU function modifier specifies that the carry-in should come from the C status bit in the status register. Note that when the cin function modifier is selected, the values of the other d0 bits related to carry-in do not matter.
- **A**—Shows the value of the arithmetic bit.
- **E**—Shows the value of the explicit multiple carry-in bit. If this bit is a 1, the carry-in is determined by the XOR of the LSB of the C port input for each segment of the ALU (assuming that the entire field is all 1s or 0s) with the F0 function signal. This allows a different carry-in for different segments of the split-ALU on the basis of a mask input to the C port.
- **S**—Shows the value of the sign-extend bit. If 1, this bit causes the four LSBs of the function code to be inverted if the sign bit of the input to the barrel rotator is a 1.
- **I**—Shows the value of the invert carry-in bit. Causes the carry-in (if derived from the C bit in d0) to be inverted if the sign bit of the input of the barrel rotator is a 1.
- **Rotator Input Sign**—Specifies the sign bit of the input to the barrel rotator. Certain settings allow the function code to be modified if the sign bit of the barrel rotator input is 1.
- **Carry-in**—Indicates the carry-in source for the given d0 bit settings and the rotator input sign. The carry-in column applies to each piece of the ALU when FMOD specifies multiple arithmetic.
  - **d0(C)**—Carry-in is set to the value of the C bit in d0.
  - **!d0(C)**—Carry-in is set to the inverse of the value of the C bit in d0.
  - **sr(C)**—Carry-in comes from the C status bit (status register bit 30).
  - **C(n)^F0**—Carry-in is derived from the XOR of the LSB of the input to each segment of the ALU with the F0 function code.
- **Invert Value**—Indicates whether bits 22–19 of d0 will be inverted before being supplied to the ALU control lines.

Chapter 14

Software Optimizations

There are many programming techniques and hardware features that you can use to optimize software for the PP. This chapter begins by discussing a look-up table (LUT) example of progressing from straight-line code to a tight loop in which several computations are pipelined to achieve higher throughput. A second group of examples shows how to combine several hardware features to achieve the best theoretical performance for a given calculation. The final section includes a list of programming hints that can be used to optimize critical loops.

Topics

14.1 Rolling Code Into a Pipelined Tight Loop ..... PP: 14-2

14.2 MSE Optimization Example ..... PP: 14-6

14.3 Programming Hints ..... PP: 14-17

14.1 Rolling Code Into a Pipelined Tight Loop

This section uses look-up table (LUT) examples to illustrate progression from straight-line code to a tight loop where several look-ups are pipelined to achieve higher throughput.

LUTs provide an efficient way of generating data that is repeatedly needed in a given algorithm. They are also frequently used in image coding and decoding. As shown in the following section, the PP can perform an LUT access in ~1.5 cycles when multiple accesses are pipelined. You should keep this fast LUT capability in mind when programming various algorithms. Using LUTs (instead of actually calculating required values) can often significantly increase efficiency. For example, tables that might be advantageous could contain transcendental function values (such as the cosine terms required by a given size DCT transform), Huffman decode (such as those defined by JPEG, MPEG, and H.261), and bit swizzling required by unorthodox data formats.

14.1.1 Individual LUT Access

The straight-line code for performing a single LUT access is shown in Example 14–1. The names in the example apply to the replacement of a pixel intensity with its histogram-equalized value retrieved from a previously generated table. However, this same code can be used for any LUT application.

The first instruction in Example 14–1 loads a pixel intensity into an index register. The contents of the index register can then be used to perform a load from a histogram equalization table. The table contains a precomputed histogram-equalized level corresponding to each possible pixel intensity in the input image.

Due to the FAE pipeline, one delay-slot instruction is required before the index register can be used to generate an address after it is written to by the execute stage of an operation. Thus, the second instruction in Example 14–1 is a nop. Although not shown in this particular example, it is possible to fill this delay-slot instruction with other useful operations.

After the delay-slot instruction, the index register is used to compute the address for the required table entry (the base of the histogram equalization table offset by the given pixel intensity). The index register is scaled by the data size (as indicated by “[ ]”) when the LUT access is performed. This converts the index from a unit of the data size to a byte offset from the base address of the table. In Example 14–1, the data size is byte, so scaling really isn’t necessary. However, when the data size is halfword or word, scaling allows the LUT access to be performed without requiring you to convert the index to a byte offset in the data unit.

Once the table entry address is computed, it is used to read the histogram-equalized intensity corresponding to the input pixel into a register. The contents of that register can then be stored back over the input pixel thus histogram-equalizing the pixel in the image.

Example 14–1. Straight-Line Coded LUT Access

```
Gx_PixelIndex:     .set   x8
PixelHistEqValue:  .set   d7
Ga_HistTblBase:    .set   a8
Ga_ImageData:      .set   a9

   Gx_PixelIndex =b *Ga_ImageData        ; Load pixel intensity into an index
                                         ;  register.

   nop                                   ; Delay Slot (Gx_PixelIndex cannot
                                         ;  be used in the address stage of
                                         ;  the pipeline until 1 Delay after
                                         ;  it is written to in the execute
                                         ;  stage of the pipeline).

   PixelHistEqValue =b *(Ga_HistTblBase+[Gx_PixelIndex])
                                         ; Load corresponding histogram
                                         ;  equalized intensity level from
                                         ;  table.

   *Ga_ImageData++ =b PixelHistEqValue   ; Store histogrammed pixel level and
                                         ;  increment image data pointer.
```

In Example 14–1, a nop is used for the required delay slot between writing to an index register and using it for address calculation. Also, in every instruction, portions of the opcode are not utilized (other operations can be performed in parallel). In order to make this code more efficient, multiple LUT accesses can be pipelined and rolled into a single loop, as discussed in the following section.

14.1.2 Pipelined LUT Accesses

Example 14–2 shows how the tight loop for performing an LUT access can be sped up from 4 cycles/access to 1.5 cycles/access with pipelined code. During each pass through the three-instruction loop, two LUT values are obtained. Notice that the first LUT access does not complete until the second instruction of the second pass through the loop. Thus, the latency for the LUT access is 5 (as opposed to 4 for the straight-line code shown in Example 14–1), but the effective throughput is 1.5 cycles/access.

For both Example 14–1 and Example 14–2, the data unit portion of the opcode is not used. Thus, if required by the application, other useful work can be performed on input data or returned table values in parallel with the LUT operation.

Example 14–2. Tight Loop for Pipelined LUT Accesses

```
Gx_Pixel0:         .set   x8
Lx_Pixel1:         .set   x0
Pixel0HistEqValue: .set   d7
Pixel1HistEqValue: .set   d6
Ga_ImageData:      .set   a9
Ga_HistTblBase:    .set   a8     ; NOTE: a8 and a1
La_HistTblBase:    .set   a1     ;  contain same value.
La_EqualizedData:  .set   a0

lut_loop:
   nop
      ||Gx_Pixel0 =b *Ga_ImageData++
      ||Pixel1HistEqValue =b *(La_HistTblBase + [Lx_Pixel1])

   nop
      ||Lx_Pixel1 =b *Ga_ImageData++
      ||*La_EqualizedData++ =b Pixel0HistEqValue

lut_loope:
   nop
      ||Pixel0HistEqValue = *(Ga_HistTblBase + [Gx_Pixel0])
      ||*La_EqualizedData++ =b Pixel1HistEqValue
```

14.2 MSE Optimization Example

This section uses mean squared error (MSE) to demonstrate progression from straight-line code to an optimized tight loop by using hardware features including multiple arithmetic, split multiply, and address unit arithmetic. Several optimization principles that can be applied to a broad class of algorithms are illustrated.

14.2.1 Straight-Line MSE Code

The mean squared error computes the sum of the squared difference between each piece of processed data (for example, a pixel) and the corresponding reference data. The MSE is often used as a measure of perceived quality because it weights large differences more than small differences.

$$MSE = \sum(p_i - o_i)^2$$

The following operations are repeated for each piece of data in order to compute the total squared error.

- Load a decoded or processed pixel.
- Load the corresponding original or reference pixel.
- Subtract the original data from the decoded data.
- Square the resulting error (or difference) term. This is done by multiplying the error by itself.
- Add the squared error to the accumulated sum of squared error terms. Once the total squared error has been computed, the mean can be obtained by dividing by the number of data elements.

Example 14–3 shows the straight-line code for accumulating the squared error term for one pixel. This will be referred to throughout this discussion as an MSE sum.

Example 14–3. MSE Sum: Straight-Line Code

```
d6 =b *a8++         ; Load original data.
  ||d5 =b *a0++     ; Load decoded data.

d7 = d6 – d5        ; Obtain error between
                    ;  original and decoded data.

d7 = d7 * d7        ; Square the error.

d4 = d4 + d7        ; Update total squared error.
```

14.2.2 Pipelined MSE Tight Loop (No Multiple Arithmetic)

Since the MSE sum is repeated for each pixel, the computations shown in Example 14–3 can be pipelined and rolled into a tight loop that uses zero-overhead looping. In Example 14–4, the following operations are performed:

- One multiply
- Two ALU operations
- Two parallel transfers

Since two ALU operations are performed, these operations cannot be rolled into a single-instruction loop.

In Example 14–4, the performance is two cycles per MSE sum. The bottleneck is that two ALU operations are being performed per MSE sum. One multiplication slot (an unutilized portion of the instruction word) is free, and two parallel transfer slots are free. The following sections show how to use these free slots to optimize the MSE tight loop.

Example 14–4. MSE: Pipelined Tight Loop

```
d3 = d6 – d5        ; Obtain error between
                    ;  original and decoded.

d7 = d3 * d3        ; Square the error.
 ||d4 = d4 + d7     ; Update total squared error.
 ||d6 =b *a8        ; Load original data.
 ||d5 =b *a0        ; Load decoded data.
```

14.2.3 Straight-Line MSE Code Using Address Unit Arithmetic

One of the key optimization opportunities on the PP is that some types of operations can be performed in different ways (that is, using different hardware) to obtain better performance. For example, if an add does not require status setting, it can be performed either in the data unit ALU or in one of the address unit's add/subtract units.

In Example 14–4, either of the ALU operations (a subtract and an add, respectively) could be performed by one of the address units instead. Example 14–5 shows the straight-line code for a second method of performing an MSE sum: using address unit arithmetic instead of an ALU operation to accumulate the running MSE sum from the squared error terms.

Using address unit arithmetic to compute the MSE requires an additional move of the multiply result from a D register (multiplies can only use D register operands) to an index register. Tradeoffs such as this are frequently involved in decisions between different ways of performing a computation.

Although not shown here, a tight loop using the method in Example 14–5 can obtain a performance of two cycles/MSE sum just like Example 14–4. However, in this case, the bottleneck is that four parallel transfers are required per MSE sum. This leaves one free multiplication slot and one free ALU slot. Because of the required delay slot between writing to an index register and using it for an address computation, a four-cycle loop is required with two MSE sums per loop iteration instead of a two-cycle loop with one sum per loop iteration.

Example 14–5. MSE: Straight-Line Code Using Address Unit Arithmetic

```
d2 =b *a9++          ; Load original data.
  ||d1 =b *a1++      ; Load decoded data.

d3 = d2 – d1         ; Obtain error between
                     ;  original and decoded data.

d3 = d3 * d3         ; Square the error.

x0 = d3              ; Move error to an index register.

nop                  ; Delay Slot before x0 can be used
                     ;  for address unit arithmetic.

d7 = &*(a3 += x0)    ; Increment MSE (stored in a3).
```

14.2.4 Pipelined MSE Tight Loop Combining Two Computation Methods

To achieve better performance, you can combine two different ways of performing a computation into the same tight loop. Example 14–6 shows one MSE sum performed using the ALU and one MSE sum performed using address unit arithmetic. The code is shown with the two MSE sum sequences in different columns. The comment field indicates the sequence in which the operations are performed for a particular data set.

The effective throughput is improved to 1.5 cycles/MSE sum.

Example 14–6. MSE: Pipelined Loop Using Two MSE Sum Methods

```
;;;;;;;;;;;;;;;;;;;INSTRUCTION 1;;;;;;;;;;;;;;;;;;;;;;;

         d7 = d7 * d7          ; COMPUTATION2d

  ||d3 = d2 – d1               ; COMPUTATION1c
            ||d6 =b *a8++      ; COMPUTATION2a
            ||d5 =b *a0++      ; COMPUTATION2b

;;;;;;;;;;;;;;;;;;;INSTRUCTION 2;;;;;;;;;;;;;;;;;;;;;;;

d3 = d3 * d3                   ; COMPUTATION1d
            ||d4 = d4 + d7     ; COMPUTATION2e
  ||d2 =b *a8++                ; COMPUTATION1a
  ||a3 = &*(a3+=x0)            ; COMPUTATION1f

;;;;;;;;;;;;;;;;;;;INSTRUCTION 3;;;;;;;;;;;;;;;;;;;;;;;

         d7 = d6 – d5          ; COMPUTATION2c
  ||x0 = d3                    ; COMPUTATION1e
  ||d1 =b *a0++                ; COMPUTATION1b
```

14.2.5 Straight-Line Code Using Multiple Arithmetic

The code in Example 14–6 obtains the best possible throughput for an MSE computation that uses 32-bit adds and 16-bit multiplies. However, better performance can be attained by using multiple arithmetic. Example 14–7 shows straight-line code for performing parallel MSE sums on byte input data by using multiple arithmetic. Note that once again, tradeoffs are involved. Since the difference between two 8-bit quantities yields a 9-bit result, you should observe the following principles to use multiple arithmetic correctly in this case.

- Since the difference, or error term, is going to be squared, only the absolute value of the error term is required. The carry-out from each byte ALU (which can be stored to mf) essentially contains the ninth bit of the result, which is the sign of the result. This sign bit information stored in mf can be used to invert the negative results. (Note that the carry bit is used for sign-detection in the unsigned case, but extended sign is used if the inputs are signed bytes.)

- Split multiply can be used to square two 8-bit error terms in a single cycle. The lower two bytes are already positioned correctly to be squared. The upper two bytes must be shifted down before the split multiply can be performed. This shifting down can be performed with the barrel rotator. Alternately, you can produce the same effect by using a halfword field extract move.

- Since each squared error term can require up to 16 bits, 32-bit sums must be used for the MSE increment. Each individual split multiply result must be extracted and added to the MSE sum. This can be done with either an ALU operation or the combination of a halfword field extract move and address unit arithmetic. The ALU operation requires a single operation; the field extract move and address unit arithmetic require two operations. For a particular tight loop, the choice between using an ALU operation or the combination of a field extract move and address unit arithmetic depends on where the bottleneck is.

Example 14–7. MSE: Multiple Arithmetic Straight-Line Code

```
Sq_ErrA:  .set   d2
Sq_ErrB:  .set   d1

   sr = 044
   d0 = SPLIT_MULT

   d6 =  *a8                       ; Load original data (4 bytes).
      ||d5 =  *a0                  ; Load decoded data (4 bytes).

   d7 =mc d6 – d5                  ; Perform a byte multiple subtract.

   d7 = 0 + ((d6&@mf)|(–d6&~@mf)   ; Byte multiple absolute value for
                                   ;  each error term.

   Sq_ErrA =m d7 * d7              ; Use split multiply to square lower two
      ||ealu(SPLIT_MULT)           ;  byte error terms.

   d4 = d4 + Sq_ErrA & ~%16        ; Increment MSE by lower byte
                                   ;  squared error

   d4 = d4 + Sq_ErrA>>16           ; Shift down squared error term for second
                                   ;  LSB and add to MSE.

   d7 =uh1 d7                      ; Field Extract of upper halfword
                                   ;  (containing the absolute error term for
                                   ;  the top two bytes).

   Sq_ErrB =m d7 * d7              ; Use split multiply to square upper two
      ||ealu(SPLIT_MULT)           ;  byte error terms.

   d4 = d4 + Sq_ErrA & ~%16        ; Increment MSE by 3rd byte squared error
                                   ;  term.

   d4 = d4 + Sq_ErrA>>16           ; Increment MSE by MSB squared error term.
```

14.2.6 Pipelined Tight Loop Using Multiple Arithmetic and Address Unit Arithmetic

In order to determine the best possible performance on the PP for a particular computation, it is useful to divide the required operations into two categories: fixed and selectable.

- The **fixed** category consists of operations that can be performed with the same efficiency in only one way. For example, the only way to perform a single-cycle multiply is by using the multiplier. The only way to perform multiple arithmetic operations is by using the ALU.

  Performing four MSE sums with byte-multiple arithmetic requires these operations:

  - Two loads
  - Two split multiplies
  - Two byte-multiple ALU operations

- The **selectable** category consists of operations that can be performed with same or comparable efficiency in more than one way. For example, an add that does not require status setting or multiple arithmetic can be performed using either the ALU or one of the address units.

  The selectable operations for performing four MSE sums using byte-multiple arithmetic are:

  - One field extract move or ALU (to right-justify the upper halfword of the byte-multiple absolute value result for performing the split multiply)
  - Four mask and add operations (each can be performed by either an ALU operation or a combination of field extract move and address unit arithmetic)

The fixed operations already use two ALU operations and two loads; therefore, combinations of the selectable operations that require more than two ALU operations or more than six parallel transfers cause the MSE sum to use five or more cycles.

The two remaining combinations of the selectable operations (two ALUs and five parallel transfers or two ALUs and six parallel transfers) allow the throughput to be four cycles/four MSE sums (effectively one cycle per MSE sum), assuming they can be packed into the four instructions. This is theoretically the best performance that can be achieved for an MSE sum on a single PP.

Example 14–8 shows a tight loop that uses two ALU operations and six parallel transfers in addition to the fixed operations to achieve the theoretical minimum of one cycle per MSE sum.

Example 14–8 also shows the setup and wrap-up code. Split multiply is required for two of the instructions in the tight loop. This means that two ALU operations must use the same EALU setup in d0. The tight loop uses two separate squared error sums. The two sums are consolidated during the wrap-up code. To find the mean squared error, you can divide the total squared error by the number of data elements.

Example 14–8. Mean Squared Error

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;MSE CODE
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  Tight Loop for computing Mean Squared Error between prediction block of
;  pixels and current block of pixels. Byte multiple arithmetic, split
;  multiply and address unit arithmetic are used to attain 1 cycle/pixel
;  performance.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Err:         .set   d7
ABS_ErrB:    .set   d7
CurrBlk:     .set   d6
PredBlk:     .set   d5
dummy:       .set   d5
Sq_ErrA:     .set   d4
ABS_Err:     .set   d4
d_val0:      .set   d3
Sq_ErrB:     .set   d2
MSE_SumB:    .set   d1

LA_SumA:     .set   a0
LA_Curr:     .set   a1
GA_Pred:     .set   a8

LX_SqErr0:   .set   x0
LX_SqErr2:   .set   x0
LX_SqErr1:   .set   x1

WIDTH:       .set   128

   .ptext

mse_setup:
   d0 = SH_ADD                              ; Set EALU for shift by 16 and
                                            ; add.
   sr = 0x24                                ; Byte Multiple arithmetic.
   MSE_SumB = 0                             ; Init. MSE_SumB to 0.
     ||d_val0 = &*(0)                       ; Init d_val0 to 0.
   Sq_ErrA = 0                              ; Init first pass Sq_ErrA.
     ||LX_SqErr2 = &*(0)                    ; Init SqErr1 to 0.
   le0= MSE_e                               ; Set Loop End.
   lrs0 = 31                                ; Loop 31 times (32 iterations)
   LA_SumA = 0                              ; Initialize Sq. Error SumA to 0.
     ||CurrBlk = *LA_Curr++                 ; Load first 4 bytes of
                                            ;  actual data.
   PredBlk = *GA_Pred++                     ; Load first 4 bytes of
                                            ;  decoded data.
     ||Sq_ErrB = &*(0)                      ; Initialize Sq. Error SumB to 0.

MSE:  Err =mc CurrBlk – PredBlk             ; Byte mult. diff between pixels.
     ||LX_SqErr0 =uh0 Sq_ErrA               ; Load byte0 Sq. Err to X Reg.
     ||dummy =&*(LA_SumA += LX_SqErr2)      ; MSE SumA update (byte 2).

   ABS_Err=m 0 + ((Err&@mf) | (–Err&~@mf))  ; Byte multiple absolute value of
                                            ;  the error.
```

Example 14–8. Mean Squared Error (Continued)

```
     ||LX_SqErr1 =uh1 Sq_ErrA              ; Load byte1 Sq. Err to X Reg.
     ||CurrBlk = *LA_Curr++                ; Load current pixels.

   Sq_ErrA =um ABS_Err * ABS_Err           ; Unsigned split multiply to
                                           ;  square the error.
     ||ABS_ErrB= ealu(SH_ADD: d_val0 + ABS_Err>>u 16)
                                           ; Shift top 2 pixels diffs. to
                                           ;  prepare for split mult.
     ||LX_SqErr2 =uh0 Sq_ErrB              ; Load byte2 Sq. Err to X Reg.
     ||dummy = &*(LA_SumA += LX_SqErr0)    ; MSE SumA update (byte 0 error).

MSE_e:Sq_ErrB =um ABS_ErrB * ABS_ErrB      ; Split multiply to square the
                                           ; error.
     ||MSE_SumB = ealu(SH_ADD: MSE_SumB + Sq_ErrB>>u 16)
                                           ; MSE SumB update.
     ||PredBlk = *GA_Pred++                ; Load predicted pixels.
     ||dummy = &*(LA_SumA += LX_SqErr1)    ; MSE SumA update.

   LX_SqErr0 =uh0 Sq_ErrA                  ; Load byte0 Sq. Err to X Reg.
     ||dummy = &*(LA_SumA += LX_SqErr2)    ; MSE SumA update (byte 2).

   LX_SqErr1 =uh1 Sq_ErrA                  ; Load byte1 Sq. Err to X Reg.

   LX_SqErr2 =uh0 Sq_ErrB                  ; Load byte2 Sq. Err to X Reg.
     ||dummy = &*(LA_SumA += LX_SqErr0)    ; MSE SumA update (byte 0 error).
br = iprs
     ||dummy =&*(LA_SumA += LX_SqErr1)     ; MSE SumA update.
   MSE_SumB = MSE_SumB + Sq_ErrB>>–d0      ; MSE SumB update.
     ||dummy = &*(LA_SumA += LX_SqErr2)    ; MSE SumA update.
   MSE_SumB = MSE_SumB + LA_SumA           ; final sum
```

14.3 Programming Hints

This section contains a brief description of several programming hints that can be used to optimize critical loops.

- Immediates can be loaded to a register via address unit arithmetic:

  ```
  d7 = &*(a15 + 7)       ; d7 = 7
     ||d6 = &*(a7 – 3)  ; d6 = –3
  ```

  which is equivalent to:

  ```
  d7 = &*7
     ||d6 = &*–3
  ```

- Write priority can be used to perform if/else-type operations in a single instruction (see Section 8.11, *Parallel Write Priority*).

- The zero overhead loop control logic can be used for hardware branches. Conditional hardware branches can be performed (as described in subsection 11.5.4, *Conditional Hardware Branching*) by loading the loop count register with a flag that is nonzero if the hardware branch is to be taken; zero if it is not.

- A byte or halfword field extract move in the parallel transfer portion of an instruction can be used to shift right by 24 or 16 (arithmetic or logical, respectively).

- Sometimes a 32-bit immediate source operand in an ALU operation can be avoided in one of these ways:

  - %imm allows immediates of the form (2^n –1) to be generated as inputs to the C port
  - 0x1\\imm allows immediates of the form (2^n) to be generated as inputs to the B port

- As shown in Example 14–9, you can create a condition code on two different conditions.

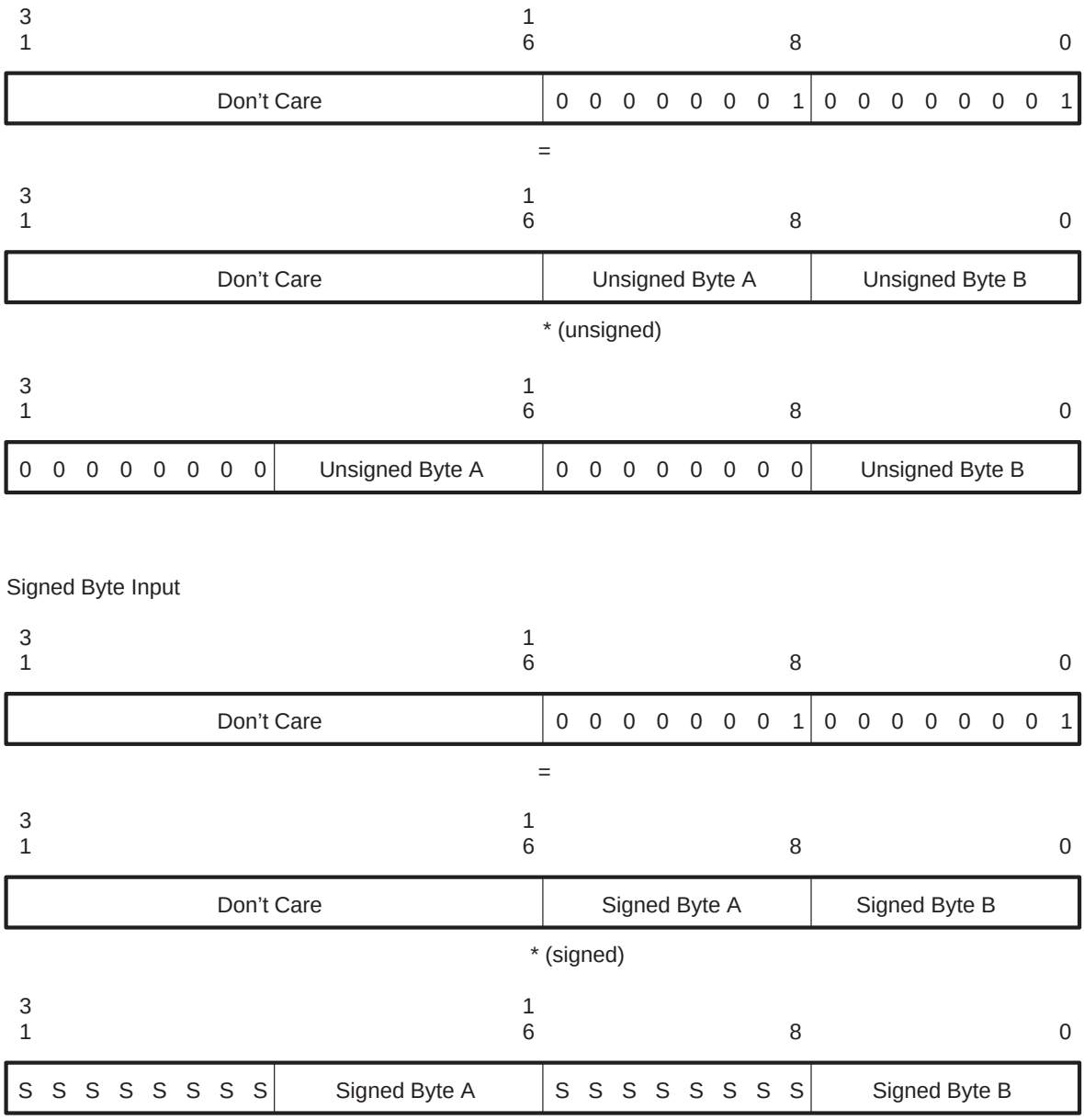
Example 14–9. Code With Two Conditions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example that returns only if both conditions are true.
;; Note this can be done for any pair with the same condition.
;; if (d1 == d2 && d3 == d4) return();
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   dDummy = d1 – d2            ; sets zero status bit if equal
   dDummy =[z] d3 – d4         ; keeps zero status bit set if equal
   br =[z] iprs                ; returns if both conditions are met

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example that will return if either of the conditions is true.
;; Note this can be done for any pair with the same condition
;; if the inverse condition is also supported.
;; if (d1 == d2 || d3 == d4) return();
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   dDummy = d1 – d2            ; sets zero status bit if equal
   dDummy =[nz] d3 – d4        ; only allows zero status bit to be
                               ;  set if not already cleared
   br =[z] iprs                ; returns if either condition is met

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Sometimes it is also possible to do two conditions that are different
;; using status protection and the combinatorial conditions:
;; if (d1 == d2 || d3 < d4) return();
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   dDummy = d1 – d2            ; sets zero status bit if equal
   dDummy =[u.z] d3 – d4       ; sets less than status if d3 is less than d4
   br =[le] iprs               ; returns if either condition is met
```

- ❑ When you need to use halfword multiple arithmetic with 8-bit data, you can convert byte pairs of data in memory to zero- or sign-extended halfword pairs in a register. You can do this quickly by using a split multiply, setting the multiplicand to 0x0101, as illustrated in Figure 14–1. In order to use split multiply, the data unit operation must be of the MPY||EALU form, and the M bit (bit 7 of d0) must be set to 1.

Figure 14–1. Right-Justified Byte Pair → Halfword Pair Using Split Multiply

Unsigned Byte Input



Signed Byte Input

- You can negate a halfword in a D register by multiplying it by –1, which can be done in parallel with an add or subtract.
- A rounded multiply by 0x8000 with a default multiply shift left of 1 can be used to swap the upper and lower halfwords in a register.

- A rounded multiply by 0 can be used to shift the upper half-word right by 16 (with zero extension).

- The same EALU configuration in d0 can often be used to perform several different operations. For example, if the EALU function code is set for (A&~C)|(B&C), B&C can be performed by setting the A input to the same value as the C input:

```
d0 = SAME
d7 = ealu(SAME: (d2&~d4) | (d3\\0&d4)); (A&~C) | (B&C)
d7 = ealu(SAME: (d4&~d4) | (d3\\0&d4)); effectively
                                      ; (B&C)
```

- If a link address is the first element in a group of structures being linked together (for example, linked packet transfers), the link addresses in the list can be set by using the following type of instruction:

```
*(a2++=SizeOfStructure) = a2
```

  A circularly linked list of four packet transfer requests could be set up as follows:

```
a15 = &*(a8 = pba + 0x200)
*(a8++=[16]) = a8
*(a8++=[16]) = a8
*(a8++=[16]) = a8
*(a8--=[48]) = a8
```

  This works because the increment of the address register takes place at the end of the address stage of the pipeline. Thus, the value being written is the incremented address; the address it is written to is the nonincremented address.

- In a PP subroutine or interrupt service routine, it is usually advantageous to distribute the PUSHs and POPs for saving state in parallel with other required operations instead of doing them all directly upon entry to or exit from the subroutine or ISR.

Chapter 15

Software Applications

This chapter provides application examples that use various PP instruction set features. The examples illustrate the flexibility, power, and features of the PP instruction set.

Topics

15.1	Graphics Applications	PP: 15-2
15.2	Image-Processing Applications	PP: 15-15
15.3	Bitstream Decoding Applications	PP: 15-32

15.1 Graphics Applications

Unlike previous programmable graphics processors, such as the TMS34010 and TMS34020, the MVP has no microcoded graphics operations. Instead, graphics operations such as line draw, PIXBLTs, and color expand are performed by a single-cycle PP instruction or a combination of several PP instructions.

The required hardware (such as the bit expander) for performing basic graphics operations is directly accessible to you and can be used in a wide range of applications. MVP flexibility lets you perform many types of graphics operations rapidly.

This section provides examples of how core graphics operations are performed on the PP.

15.1.1 Color Expand: Bit-to-Byte/Halfword/Word Translations

Bitmapped character fonts are usually stored at one bit per pixel and expanded into color as they are drawn. Color expansion can be time-consuming unless hardware supports it directly. The PP's expand operator (@mf) performs bit-to-byte, -halfword, or -word translations.

Example 15–1 illustrates code that uses the expand operator to perform an 8-bit color expand. The example assumes that the input bitmap and the output pixel map follow a big-endian ordering. Each bit with value of 1 is expanded to the 8-bit foreground color; each bit with 0 value is expanded to the 8-bit background color.

Example 15–1. Color Expand Code

```
FG_Color:             .set  d7
BG_Color:             .set  d6
num_bits:             .set  d5
ColorExpandedPixels:  .set  d1

La_ExpandedPixels:    .set  a0
Ga_BitPixels:         .set  a8

   d4 = num_bits>>5
      ||mf = *Ga_BitPixels++          ; Load first word of bit pixels
                                      ;  into mf.
   le0 = loop0_end                    ;
   lrs0 = –1 + d4                     ; Set lc1/lr1 and enable le0.
   lrse1 = 6                          ; Single instruction loop
                                      ; performed 7 times
                                      ; per pass through loop0.
   ls0 = loop0_start                  ; Correct ls0.
   sr = 0x60                          ; Set MSize to byte (4)
                                      ;  and R to 1.

loop0_start:
loop1_start_end:                      ; 1st Outer Loop Instruction
   ColorExpandedPixels = (FG_Color & @mf) | (BG_Color & ~@mf)
                                      ; Expand 4-bit pixels to 4-byte pixels.
   ||*La_ExpandedPixels++ = ColorExpandedPixels
                                      ; Store 4 color-expanded pixels.

loop0_end:                            ; 2nd Outer Loop Instruction
   ColorExpandedPixels = (FG_Color & @mf) | (BG_Color & ~@mf)
                                      ; Expand 4-bit pixels to 4-byte pixels.
   ||mf =w *Ga_BitPixels++            ; Load next word of bit pixels.
   ||*La_ExpandedPixels++ = ColorExpandedPixels
                                      ; Store 4 color-expanded pixels.

   *La_ExpandedPixels++ = ColorExpandedPixels
                                      ; Store final 4 color-expanded pixels.
```

For simplicity, the example code assumes that the number of bits being color expanded is an integer multiple of 32 and that the input data is word-aligned in memory. Loop0 is thus performed once per word of bit pixels. Without these assumptions, more setup and wrap-up code is required around loop0.

sr is set appropriately for bit-to-byte expansion of the four MSBs in **mf**. MSize (sr bits 5–3) is set to 4 (100b). This selects byte data size for mf expansion. **R** (sr bit 6) is set to 1. This bit causes the input to the mf expander to be taken from the MSBs of mf. It also indicates that the mf register should be rotated left by the number of bits specified in MSize (in this case, four) for ALU operations involving the expand operator (@mf).

Example 15–1 contains an outer loop (loop0) and an inner loop (loop1). The first instruction in the outer loop (loop0) is the single-instruction inner loop (loop1), which is performed seven times per pass through the outer loop.

The instruction expands four single-bit pixels to bytes while storing pixels expanded by the previous instruction. The second instruction in the outer loop is performed once each pass through the loop. In addition to byte-expanding four single-bit pixels and storing the four previously expanded pixels, it loads mf with the next 32-bit word of bit pixels.

A software write to mf dominates over mf rotation thus an entire 32-bit word is loaded into mf.

Example 15–2 shows one pass through loop0 for a sample set of data.

Example 15–2. Sample Pass Through Color Expand Loop0

Values upon entry to loop0:

- ❑ sr 0x60 (MSize is byte, and rotate mf is set)
- ❑ FG_Color 0x88888888
- ❑ BG_Color 0x33333333
- ❑ lc0 1
- ❑ lc1 6
- ❑ mf 0x12345678

After first instruction:

- ❑ ColorExpandedPixels = 0x33333388 = (0x88888888 & 0x000000FF) | (0x33333333 & 0xFFFFFF00)
- ❑ lc1 = 5
- ❑ mf = 0x23456781

After second instruction:

- ❑ ColorExpandedPixels = 0x33338833 = (0x88888888 & 0x0000FF00)| (0x33333333 & 0xFFFF00FF)
- ❑ lc1 = 4
- ❑ mf = 0x34567812

After third instruction:

- ❑ ColorExpandedPixels = 0x33338888 = (0x88888888 & 0x0000FFFF) | (0x33333333 & 0xFFFF0000)
- ❑ lc1 = 3
- ❑ mf = 0x45678123

After fourth instruction:

- ❑ ColorExpandedPixels = 0x33883333 = (0x88888888 & 0x00FF0000) | (0x33333333 & 0xFF00FFFF)
- ❑ lc1 = 2
- ❑ mf = 0x56781234

After fifth instruction:

- ❑ ColorExpandedPixels = 0x33883388 = (0x88888888 & 0x00FF00FF) | (0x33333333 & 0xFF00FF00)
- ❑ lc1 = 1
- ❑ mf = 0x67812345

After sixth instruction:

- ❑ ColorExpandedPixels = 0x33888833 = (0x88888888 & 0x00FFFF00) | (0x33333333 & 0xFF0000FF)
- ❑ lc1 = 0
- ❑ mf = 0x78123456

After seventh instruction:

- ❑ ColorExpandedPixels = 0x33888888 = (0x88888888 & 0x00FFFFFF) | (0x33333333 & 0xFF000000)
- ❑ lc1 = 6
- ❑ mf = 0x81234567

After eighth instruction:

- ❑ ColorExpandedPixels = 0x88333333 = (0x88888888 & 0xFF000000) | (0x33333333 & 0x00FFFFFF)
- ❑ lc1 = 6
- ❑ lc0 = 0
- ❑ mf = new word loaded from memory
- ❑ return to start of loop0

Note: When you use the PP debugger and single-stepping in this example, the loop counter may be different than it appears here; the debugger is working at the fetch stage, and the count value shown above is for the cycles left in the execute stage.

15.1.2 Bresenham Line Drawing

Like other graphics operations on the MVP, line drawing is performed by a combination of packet transfers and pixel processing that uses PP instructions instead of microcode.

The inner loop for generating the table for a delta-guided transfer according to the Bresenham line-drawing algorithm is shown in Example 15–3. The decision variable (D_Var) is updated by either the nondiagonal increment (NonD_Inc) if negative, or the diagonal increment (D_Inc) if nonnegative. In parallel with the ALU operation, an output delta address based on the previous ALU operation is stored; the stored address is nondiagonal delta (NonD_Delta) if negative, or the diagonal delta (D_Delta) if nonnegative.

The inner loop shown in Example 15–3 generates a guide table entry for every pixel in the desired line. In order to take advantage of accesses to bytes within the same doubleword in memory, you could use instead a variable-patch delta-guided transfer with a table entry for each nondiagonal run of pixels.

Note that a Bresenham line consists of runs of one of two lengths. The ratio of the number of short runs to long runs depends on the angle of the line. This ratio must be computed using a divide. Since a divide operation requires several cycles on the PP, this algorithm is efficient only for lines that are greater than a certain length and close to horizontal.

Example 15–3. Bresenham Line Draw Inner Loop

```
D_Var:          .set   d1 ; Decision Variable
NonD_Inc:       .set   d3 ; Nondiagonal Increment
D_Inc:          .set   d2 ; Diagonal Increment
Ga_delta_tbl:   .set   a8 ; Pointer to Delta Address Table
NonD_Delta:     .set   d5 ; Non-diagonal Delta Address
D_Delta:        .set   d4 ; Diagonal Delta Address

   D_Var = D_Var + NonD_Inc[n]D_Inc
                         ; If (negative)
                         ;   D_Var = D_Var + NonD_Inc;
                         ;  else
                         ;   D_Var = D_Var + D_Inc;
   ||*Ga_delta_tbl++ = NonD_Delta[n]D_Delta
                         ; If (negative)
                         ;   *GA_delta_tbl++ = NonD_Delta;
                         ;  else
                         ;    *GA_delta_tbl++ = D_Delta;
```

15.1.3 Pixel Block Transfers (PIXBLTs)

Pixel block transfer (commonly referred to as PIXBLT) operations combine the pixels in two source blocks according to one of many pixel-processing options. The result is written to a destination block that is typically one of the source blocks. Microsoft Windows defines three-operand PIXBLTs as shown in Figure 15–1.

While graphics processors such as the Texas Instruments TMS34020 use microcoded PIXBLT operations, the MVP performs PIXBLTs, as shown in Figure 15–2, by using user-programmed input and output packet transfers (PTs) serviced by the transfer controller. These packet transfers are “sandwiched” around pixel processing typically performed by one of the PPs. For fills at eight bits/pixel and above, the TC supports fill-with-value packet transfers, as described in subsection 15.1.3.1. **Move PIXBLTs**, in which a source block is copied to a destination block without any pixel processing, can be performed by a single packet transfer.

Figure 15–1. Three-Operand PIXBLTs

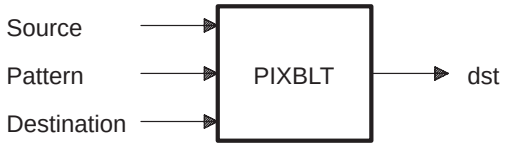
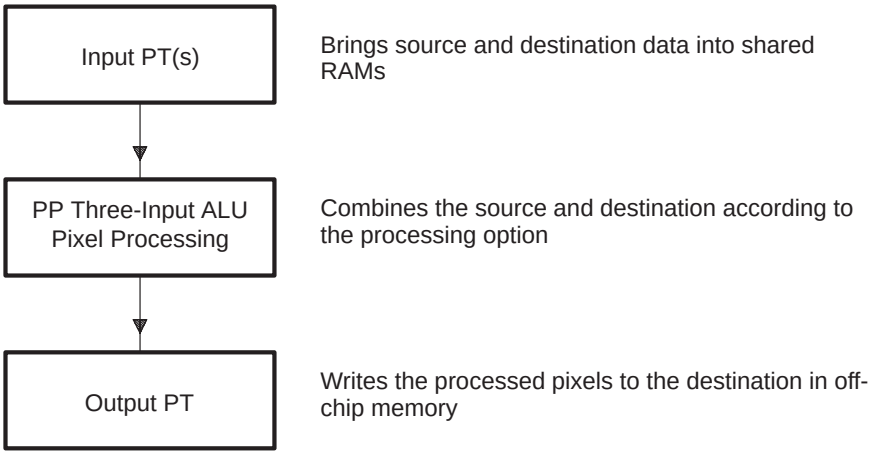


Figure 15–2. PIXBLT



15.1.3.1 Solid Fills

Solid fill is a simple form of PIXBLT that writes a specified color to the pixels in the destination. The transfer controller provides a fill-with-value source transfer mode for packet transfers that uses a 64-bit fill pattern as the source data. The destination transfer mode is independent of the source transfer mode and can thus use its multidimensional counts and associated pitches to define the destination block of data. The byte or bytes written within the destination doubleword are taken from the corresponding bytes within the fill value doubleword. All the PP (or MP) has to do to perform a fill is to setup the desired packet transfer parameters and submit the packet transfer request to the transfer controller.

One important use for fill-with-value packet transfers is to prezero a block or area of memory. Prezeroing is required in compression standards before decoding the nonzero IDCT input bins. The set-up code for a packet transfer request that toggles between two PP data RAMs prezeroing 768 bytes is shown in Example 15–4.

Example 15–4. Setup of Prezero Packet Transfer

```
STOP:               .set   0x1<<31            ; Constant used to set stop
                                              ;   bit in PT Options.
FILL:               .set   0x1<<12            ; Constant used to set fill
                                              ;  mode bit in PT Options.
DST_TOGGLE_UPDATE: .set   0x1                 ; Constant used to set toggle
                                              ;  update mode of destination
                                              ;  address.
Ga_PreZeroPT:       .set   a8                 ; Pointer to PT Parameters
PreZeroPT_Offset:   .set   0x200              ; Parameter RAM Offset for
                                              ; PT Parameters.
NUM_BYTES_TO_ZERO: .set   768                 ; Number of bytes to fill.

   .include ”packetpp.i”

Zero_PR:
   d7 = &*(pba + PreZeroPT_Offset)
   *(Ga_PreZeroPT =pba + PreZeroPT_Offset) = d7
                                              ; Set next entry address.
   d7 = STOP | FILL | DST_TOGGLE_UPDATE
                                              ; PT_Options.
   *Ga_PreZeroPT.sPT_Options = d7             ; Store PT options.
   d7 = &*(dba + 0x100)
   *Ga_PreZeroPT.sPT_DstStartAddress = d7     ; Initialize dst start address.
   d7 = NUM_BYTES_TO_ZERO
   *Ga_PreZeroPT.sPT_DstBACount = d7          ; Initialize dst A counts.
                                              ;  (B count = 0)
   *Ga_PreZeroPT.sPT_DstCCount  = a15         ; Dst C count = 0.
   *Ga_PreZeroPT.sPT_LSFillWord = a15         ;
   *Ga_PreZeroPT.sPT_DstBPitch  = a15
   *Ga_PreZeroPT.sPT_MSFillWord = a15         ;
   d7 = 0x800
   *Ga_PreZeroPT.sPT_DstCPitch  = d7          ;
```

15.1.3.2 Pixel Processing

Pixel processing (sometimes referred to as raster operations) controls how the source pixels are combined. For simple Boolean pixel processing on the PP, only one instruction is required.

Other raster operations such as MAX, MIN, add-with-saturate, and subtract-with-saturate require two instructions. For 8-bit pixels, multiple-byte arithmetic can often be used to process four pixels in parallel on a single PP.

15.1.3.3 MAX/MIN

By combining multiple arithmetic status saving to mf with expanded mf mask generation, you can merge byte, word, or halfword fields in two registers on the basis of status. This section shows how to use that combination to perform a multiple MAX and/or MIN operation. Either MAX or MIN can be used for combining source and destination data in PIXBLT operations. They can also be used together to sort data such as for performing a median filter.

Example 15–5 illustrates multiple-byte merging of the MAX and MIN data in two registers, d1 and d2. The first step is to perform a byte-multiple compare between the data in d1 and d2, setting flags in the mf register on the basis of the split ALU carries.

For subtract operations, carry corresponds to $\overline{\text{borrow}}$ (an alias for carry). Thus, split ALU results generating a carry indicate that the byte in d1 is greater than or equal to corresponding byte in d2.

The second step is to merge the bytes in d1 & d2 by using a mask created by expanding the carry flags that were saved to mf. For merge MAX, d1 is ANDed with @mf (zeros bytes that are less than the corresponding byte in d2), and the result is ORed with the result of d2 ANDed with ~@mf (which zeros bytes that are less than or equal to the corresponding byte in d1). Similarly, a MIN is performed by simply inverting the @mf sense.

Example 15–5. Merge MAX/MIN

```
sr = 0x24              ;  MSize & ASize set to byte.
d3 = mc d1 – d2        ;   mf gets byte, carries (borrow for subtract)

;; Example Data
;; Multiple Compare Setting Carries
;;
;;    d1   | D0 | 64 | 28 | 89 |
;;
;;
;;
;; –  d2   | A3 | 7A | 2D | 4E |
;;
;;
;;
;; =  d3   | 2D | EA | FB | 3B |
;;
;;
;;
;;                  3  2  1  0
;;
;;    mf   |      | 1  | 0  | 0  | 1  |          ;  Carries (borrow) from Split ALU
;;

d3 = (d1&@mf) | (d2&~@mf)                        ;  Merge MAX byte in d1 or d2
                                                 ;   according to contents of mf.
;; MAX Merge
;; Example Data
;;
;;    d1   | D0 | 64 | 28 | 89 |   & @mf    | FF | 00 | 00 | FF |
;;
;;
;;
;; |  d2   | A3 | 7A | 2D | 4E |   &~@mf    | 00 | FF | FF | 00 |
;;
;;
;;
;; =  d3   | D0 | 7A | 2D | 89 |
;;

d4 = (d1&~@mf) | (d2&@mf)                        ;  Merge MIN byte in d1 or d2
                                                 ;   according to contents of mf.
;; MIN Merge
;; Example Data
;;
;;    d1   | D0 | 64 | 28 | 89 |   &~@mf    | 00 | FF | FF | 00 |
;;
;;
;;
;; |  d2   | A3 | 7A | 2D | 4E |   & @mf    | FF | 00 | 00 | FF |
;;
;;
;;
;; =  d3   | A3 | 64 | 28 | 4E |
;;
```

15.1.3.4 Add-With-Saturate

An add-with-saturate operation stops the result of an add at the maximum unsigned value without allowing the result to overflow. The PP can perform an add-with-saturate by using two single-cycle instructions: an add, and a merge of the add result or the saturation level based on whether a carry-out (indicating overflow) occurred. If the pixel size is byte, these two instructions can perform four add-with-saturates in parallel by using multiple arithmetic, as shown in Example 15–6.

Example 15–6. Add-With-Saturate

```
    sr = 0x24              ;  MSize & ASize set to byte.
    d4 = 0xFFFFFFFF        ;  4 bytes set to 255
    d3 = mc d1 + d2        ;  mf gets byte carries

;; Example Data
;; Byte-Multiple Add Setting Carry-Out in mf
;;
;;    d1   | FF | 64 | C8 | 1E |
;;
;;
;;
;; +  d2   | 01 | 7A | 64 | 32 |
;;
;;
;;
;; =  d3   | 00 | DE | 2C | 50 |
;;
;;
;;
;;                  3  2  1  0
;;    mf   |        | 1 | 0 | 1 | 0 |          ;  Carries from Split ALU
;;

    d3 = (d4&@mf) | (d3&~@mf)         ;  Merge 255 or sum
                                       ;   according to contents of mf.
;; Merge Saturation Level or Sum
;; Example Data
;;
;;    d4   | FF | FF | FF | FF |   & @mf    | FF | 00 | FF | 00 |
;;
;;
;;
;; |  d3   | 00 | DE | 2C | 50 |   &~@mf    | 00 | FF | 00 | FF |
;;
;;
;;
;; =  d3   | FF | DE | FF | 50 |
;;
```

15.1.3.5 Subtract-With-Saturate

A subtract-with-saturate operation stops the result at 0 without allowing it to underflow. The PP can perform a subtract-with-saturate by using two single-cycle instructions: a subtract, and a merge of the subtract result or the saturation level based on whether a carry-out (indicating no underflow) occurred. If the pixel size is byte, these two instructions can perform four subtract-with-saturates in parallel by using multiple arithmetic, as shown in Example 15–7.

Example 15–7. Subtract-With-Saturate

```
    sr = 0x24              ;  MSize and ASize set to Byte.
    d4 = 0x0               ;  Saturate level = 0.
    d3 = mc d1 – d2        ;  Byte-Multiple Subtract
                           ;   setting carries in mf.

;; Example Data
;; Byte-Multiple Subtract Setting Carry-Out (borrow) in mf
;;
;;    d1   | FF | 64 | C8 | 1E |
;;
;;
;;
;; –  d2   | 01 | 7A | 64 | 32 |
;;
;;
;;
;; =  d3   | FE | F4 | 64 | EC |
;;
;;
;;
;;                  3  2  1  0
;;
;;    mf   |  §   | 1 | 0 | 1 | 0 |          ;  Carries from split ALU
;;              §
;;

    d3 = (d4&~@mf) | (d3&@mf)         ;  Merge saturation level
                                       ;   or difference according to
                                       ;   contents of mf.

;; Merge Saturation Level or Difference
;; Example Data
;;
;;    d4   | 00 | 00 | 00 | 00 |   &~@mf   | 00 | FF | 00 | FF |
;;
;;
;;
;; |  d3   | FE | F4 | 64 | EC |   & @mf   | FF | 00 | FF | 00 |
;;
;;
;;
;; =  d3   | FE | 00 | 64 | 00 |
;;
```

15.1.3.6 Plane Masking

Plane masking protects certain bits within pixels from being modified by pixel processing. For some of the pixel-processing operations, plane masking can be performed in the same instruction by feeding each of the two source pixels to an ALU input and the plane mask to the third ALU input.

In Example 15–8, even bits in the destination pixels are protected from modification; odd bits in the destination pixels are set to the AND of the source pixel and the original value of the destination pixel.

The PLANE_MASK constant in Example 15–8 could be modified to protect any combination of the bits in the destination pixels. Alternately, the plane mask could be stored in a register to allow it to be varied dynamically. Any Boolean operation can be performed between the destination pixel and the source pixel to determine the resulting pixel value for the bits that are not plane masked.

Example 15–8. Pixel Processing With Plane Masking

```
dest_pixel:     .set   d7
source_pixel:   .set   d6
PLANE_MASK:     .set   0x55555555

 dest_pixel= (dest_pixel&PLANE_MASK) | (dest_pixel&source_pixel&~PLANE_MASK)

;; Example Data (binary)
;;
;;    dest_pixel:     11100011100011100011100011100011
;;
;; & PLANE_MASK:      01010101010101010101010101010101
;;
;;        |
;;
;;    source_pixel:   00000001001000110100010101100111
;;
;; & ~PLANE_MASK:     10101010101010101010101010101010
;;
;;
;; dest_pixel =       01000001001001100001000001100011
```

15.1.3.7 Transparency

Transparency is a graphics operation that allows you to decide which pixels in an object will be visible. To draw an object on a background or on top of another object, the pixels in a source array that are not part of the object are made transparent so that the background or object beneath shows through.

Source transparency can be performed directly by the TC as part of a packet transfer. When the transparency packet transfer mode is selected, the source data will be written to the destination only if it is not equal to the transparency value.

Transparency can be specified with an 8-, 16-, 32-, or 64-bit data size. Thus, one 64-bit, two 32-bit, four 16-bit, or eight 8-bit comparisons are made. If any of the comparisons are true, the TC disables the corresponding write to off-chip memory.

Alternately, transparency can be performed by the PP during the pixel processing. The transparency bits corresponding to each pixel can be loaded into the mf register and then expanded to generate a transparency mask used for pixel processing.

15.2 Image-Processing Applications

Imaging applications require processing techniques that range from multiplication-intensive filtering and frequency-domain transforms to massive numbers of simple arithmetic operations on pixel data, bit-field extraction, and look-up table (LUT) operations. Many image-processing applications require adaptive algorithms that select among a variety of techniques according to characteristics detected in the image.

The PP provides acceleration hardware that is not dedicated to a narrow set of functions. For example, the same mf expand hardware that is used for color expansion enables a multiple sum of absolute differences operation that greatly speeds up motion estimation. The PP instruction set gives you direct access to flexible hardware functions without framing them within the constraints of specific algorithms.

15.2.1 Thresholding: Word/Halfword/Byte-to-Bit Translations

Multiple arithmetic can be used to perform word-, halfword-, or byte-to-bit translations. Example 15–9 shows how to threshold byte-sized pixels, converting them to one-bit pixels. It also demonstrates two optional multiple-arithmetic features: mf rotation and mf clearing, which are explained more thoroughly in Chapter 7, *Summary of PP Registers*.

In Example 15–9, one pass through loop1 generates a 32-bit word of one-bit pixels from eight words of packed bytes. The primary operation is a multiple-byte compare between the threshold level and the pixel data.

The split ALU carries are saved to the mf register. When the pixel is less than or equal to the threshold, no carry is generated and a one-bit pixel of 0 results. When the pixel is greater than the threshold, a carry is generated and a one-bit pixel of 1 results.

Loop1 contains two instructions. The first instruction writes the carries generated by the byte-multiple compare between the threshold and the pixel data into the four LSBs of mf. The 28 LSBs of mf are cleared. The global transfer stores the contents of the mf register (before it is modified by the data unit operation), which contains 32 one-bit pixels from the previous iteration of the loop.

The local transfer loads the next word of packed-byte pixels. The second instruction in loop1 is set up as single instruction loop2, which is performed seven times per pass through loop1. For this instruction, the **r** ALU function modifier causes the mf register to be rotated left by 32/ASize (in this case, four bits). Multiple status setting to the mf register overwrites the four LSBs of the left-rotated mf register. Thus, after seven rotations and eight sets of carry writes to mf, mf contains a full word of one-bit pixels.

Example 15–9. Byte-Bit Translation

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     Program:          Global Thresholding Tight Loop
;     Description:      Using byte multiple arithmetic, 4 pixels
;                       are thresholded in 2 cycles.
;
;     Register Allocation:
;
;     d1–   8-bit threshold value (replicated in each byte)
;     d2–   4 adjacent pixel values
;     a0–   Input pointer
;     a8–   Output pointer
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   sr = 0x4                           ; Set ASize for byte.

   d2 = *(a0 = dba)                   ; Load word of packed pixels.
   a8 = &*(dba + 0x8000)              ; Initialize a8.

   le2 = loop2
   le1 = loop2
   lrs1 = 39                          ; Outer loop performed 40 times.
   lrs2 = 6                           ; Inner loop is repeated 7 times
                                      ;  per outer loop.
   d1 = 0xCCCCCCCC                    ; Set threshold level to 0xCC (204).

loop1: d3 =mzc d1 – d2                ; Set mf based on carries from
                                      ;  comparison of 1 word containing
                                      ;  4 copies of the byte-sized
                                      ;  threshold value with 4 adjacent
                                      ;  pixels packed into a word.
            ||*a8++= mf               ; Store word of 1-bit pixels.
            ||d2 = *++a0              ; Load next word of packed pixels.

loop2: d3 =mrc d1 – d2                ; Set mf based on carries from
                                      ;  comparison of 1 word containing
                                      ;  4 copies of the byte-sized
                                      ;  threshold value with 4 adjacent
                                      ;  pixels packed into a word.
            ||d2 = *++a0              ; Load next word of packed pixels.
```

15.2.2 Binary Morphology

Binary morphology refers to topology or structure analysis that uses local transformations analogous to convolution based on logical rather than arithmetic operations. Two basic operators are used in binary morphology: erosion and dilation.

- **Erosion.** Translates a structuring element throughout an image, zeroing pixels where the structuring element is not a subset of the image data around the pixel. Erosion has the effect of shrinking objects.
- **Dilation.** Translates a structuring element throughout an image, setting pixels to 1 where the structuring element has a nonempty intersection with the image data around the pixel. Dilation has the effect of expanding objects.

When erosion is coupled with dilation, it is possible to define operations such as **open**, which smooths contours and suppresses small islands and sharp caps, or **close**, which blocks up narrow channels and thins lakes. Boundary extraction can be achieved by simply taking the set difference between the original image and the eroded image. The three-input ALU can be used to implement both erosion and dilation efficiently.

15.2.2.1 Erosion

Erosion of object X by structuring element B is defined as the set of all points x such that B_X—the translation of B so that its origin is located at x—is included in X.

Erosion: $X \Theta B, x{:}B_X \subset X)$

In other words, erosion is defined as the condition in which the object in an image totally contains the structuring elements. Erosion can be thought of as the intersection of all translations of the image with the structuring element.

Typically, erosion is implemented by translating the structuring element throughout the image, zeroing pixels where the structuring element is not a subset of the image data around the pixel. For example, consider a structuring element that is a 3×3 block. At each translation, the eroded output will be 1 only if each of the nine pixels in the image have the same binary value as each of the elements in the structuring element. Thus, the effect of erosion is to shrink objects in the image.

The PP tight loop for performing erosion on a 3×3 block is shown in Example 15–11. This tight loop assumes that the loop setup and variable assignments in Example 15–10 has been performed and that a shared RAM contains the binary pixel data for the image.

Each word that is operated on represents 32 pixels; therefore, multiple pixels are eroded simultaneously. Since the leftmost and rightmost bits in a word do not contain their corresponding neighbors, their output will not be valid.

For example, if the structuring element is 3×3 with the origin being the center pixel; the output will be valid only for 30 pixels because the leftmost and rightmost pixels cannot shift and obtain the necessary data. Thus, each time a new word is loaded, the next 30-bit pixels to be processed must be padded in a 32-bit word with a bit in the MSB and a bit in the LSB.

Since only 30 new pixels are loaded each time, the desired data does not usually fall within a 32-bit word in memory. Thus, it is necessary to merge data from two words in memory to form the 32-bit word to be processed in the tight loop.

Example 15–11 shows a 21-instruction tight loop that performs a 3×3 structuring-element erosion on 30 pixels. The achieved performance is 21/30 cycles/pixel.

Example 15–10. Setup and Variable Assignments for Erosion Tight Loop

```
d_32:               .set  d7
right_out_shift:    .set  d7
left_in_shift:      .set  d6
merged_output:      .set  d5
out_word1:          .set  d5
word2_row0:         .set  d5
mword_row1:         .set  d5
word2_row1:         .set  d5
mword_row2:         .set  d5
word2_row2:         .set  d5
out_word1:          .set  d5
mword_row0:         .set  d4
word1_row2:         .set  d4
left_out_shift:     .set  d3
SE_0:               .set  d3 ;  Structuring Element 0.
SE_1:               .set  d3 ;  Structuring Element 1.
SE_2:               .set  d3 ;  Structuring Element 2.
SE_3:               .set  d3 ;  Structuring Element 3.
SE_4:               .set  d3 ;  Structuring Element 4.
SE_5:               .set  d3 ;  Structuring Element 5.
SE_6:               .set  d3 ;  Structuring Element 6.
SE_7:               .set  d3 ;  Structuring Element 7.
SE_8:               .set  d3 ;  Structuring Element 8.
word1_row0:         .set  d2
word1_row1:         .set  d2
ORofXORs:           .set  d1
ErodedOut:          .set  d1
out_word2:          .set  d1
merged_out:         .set  d1
d0_one:             .set  d0
d0_31:              .set  d0
La_row0:            .set  a0
La_row1:            .set  a1
La_row2:            .set  a2
La_Scratch:         .set  a3

Ga_StructElem:      .set  a8
Ga_Out:             .set  a9
Ga_Scratch:         .set  a11

 ; Setup Loop Controllers for Erosion Tight Loops

   lctl = 0xBA0             ; Enable le2 and le1 and associate lc2 and lc1.
   le1 = end2
   ls1 = loop1
   lr1 = 1                  ; & lc1 = 1
   le2 = end2
   ls2 = loop2
   lr2 = 5                  ; Repeat 6 times 2nd time through.
     ||d5 = &*(La_row1–= [1])
   lc2 = 15                 ; Repeat 16 times 1st time through.
     ||d5 = &*(La_row2–=[1])

; Enter Inner Erosion Tight Loop First

   call = loop2
   left_in_shift = 32
   x9 = 0x0
     ||d4 = *La_row0
```

Example 15–11. Erosion Tight Loop

```
; Erosion Outer Loop:  Adjusts input pointers and handles situation where a full
; 32-bit output word is not available after pass through the inner loop. This
; occurs once every 16 passes through the inner loop.

loop1: left_in_shift = 30                ; Initialize left/-right input shift.
     ||word1_row0 = *(La_row0 – [2])     ; Input word1 for row n.

   x9 = a15
     ||word2_row0 = *(La_row0 –= [1])    ; Input word2 for row n.

   mword_row0 = word2_row0>>u –left_in_shift
                                         ; Shift word2 for merging.
     ||a15 = *La_row1––                  ; decrement row n+1 pointer.

   mword_row0 = (word1_row0<<left_in_shift) | mword_row0
                                         ; Merge input from row n.
     ||a15 = *La_row2––                  ; decrement row n+2 pointer.

; Erosion Inner Loop:  Erodes 30 1-bit pixels in parallel.  Must merge input from
; two words in memory for each of the three rows of data. Must also merge the
; ”left-over” output data from the previous pass with output data from the current
; pass to produce a 32-bit output of eroded pixels.

loop2: ORofXORs = mword_row0<< d0_one ^ SE_0          ; tmp = S0 XOR I0.
     ||SE_1 = *(Ga_StructElem + [1])                  ; Load S1.
     ||d0_31 = &*(–1)                                 ; Load d0 for rotate right
                                                      ;  of 1 (rotate left by 31).

   ORofXORs = ORofXORs | (mword_row0^SE_1)            ; tmp = tmp OR (S1 XOR I1).
     ||SE_2 = *(Ga_StructElem + [2])                  ; Load S2.
     ||word1_row1 = *La_row1                          ; Load word1 from row n+1.

   ORofXORs = ORofXORs | (mword_row0\\ d0 ^ SE_2)     ; tmp = tmp OR (S2 XOR I2).
     ||word2_row1 = *++La_row1                        ; Load word2 from row n+1.

   mword_row1 = word2_row1 >>u –left_in_shift

   mword_row1= word1_row1<<left_in_shift|mword_row1  ; Shift word1 for merging.
     ||SE_3 = *(Ga_StructElem + [3])                  ; Load S3.
     ||d0_one = &*(1)                                 ; default rotate = 1.

   ORofXORs = ORofXORs | (mword_row1\\ d0_one^SE_3)  ; tmp = tmp OR (S3 XOR I3).
     ||SE_4 = *(Ga_StructElem + [4])                  ; Load S4.
     ||d0_31 = &*(–1)                                 ; default rotate = 31.

   ORofXORs = ORofXORs | (mword_row1^SE_4)            ; tmp = tmp OR (S4 XOR I4).
     ||SE_5 = *(Ga_StructElem + [5])                  ; Load S5.
     ||word1_row2 = *La_row2                          ; Load word1 from row n+2.

   ORofXORs = ORofXORs | (d5\\ d0_31 ^ SE_5)          ; tmp= tmp OR (S5 XOR I5).
     ||SE_6 = *(Ga_StructElem + [6])                  ; Load S6.
     ||word2_row2 = *++La_row2                        ; Load word2 from row n+2.

   mword_row2 = word2_row2 >>u –left_in_shift

   mword_row2=word1_row2<<left_in_shift|mword_row2   ; Shift word1 for merging
     ||d0_one = &*(1)                                 ; default rotate = 1.

   ORofXORs = ORofXORs | (d5\\ d0 ^ SE_6)             ; tmp = tmp OR (S6 XOR I6).
     ||SE_7 = *(Ga_StructElem + [7])                  ; Load S7.
     ||word1_row0 = *La_row0                          ; Load word1 from row n.
```

Example 15–11. Erosion Tight Loop (Continued)

```
   ORofXORs = ORofXORs | (d5^SE_7)                    ; tmp = tmp OR (S7 XOR I7).
     ||SE_8 = *(Ga_StructElem + [x8])                 ; Load S8 (X8 = 32).
     ||d0_31 = &*(–1)                                 ; default rotate = 31.

   ErodedOut = ~ORofXORs & (d5\\d0_31 ^ ~SE_8)        ; out = ~tmp & (S8 XNOR I8).
     ||d_32 = &*(32)

   ErodedOut = ErodedOut & 0x7FFFFFFE                 ; Mask off msb and lsb.

   right_out_shift = d_32 – left_in_shift
     ||out_word1 = *Ga_Scratch                        ; Load out_word1.

   left_out_shift = 30 – left_in_shift

   out_word2 = ErodedOut>>u –right_out_shift
     ||*La_Scratch = d1                               ; Tmp store next loop word1.

   merged_out=out_word1<<left_out_shift | out_word2   ; Shift word1 for out
     ||SE_0 = *Ga_StructElem                          ; (los=ris) Load S0.
     ||word2_row0 = *++La_row0                        ; Load word2 from row n.

   left_in_shift =left_in_shift –2                    ; Update left/right input
                                                      ;  shift.

   mword_row0 = word2_row0>>u –left_in_shift          ; Shift word2 for merging.
     ||*(Ga_Out++=[x9]) = merged_out                  ; Store merged output.
     ||d0_one = &*(1)                                 ; default rotate = 1
end2:
   mword_row0 = (d2<<left_in_shift) | mword_row0      ; Merge new data from row n.
     ||x9 = d0_one
```

15.2.2.2 Dilation

Dilation can be performed using the same tight loop as shown in Example 15–11, with the exception that the three-input Boolean operation used to perform erosion is changed to the three-input Boolean function used to perform dilation: dst = A | B&C.

15.2.3 MAD—Mean Absolute Difference (Motion Estimation)

For MPEG or h.261, motion estimation is frequently performed by comparing the MAD (mean absolute difference) between a block of data in the frame being coded and selected blocks from the previous decoded (reference) frame. The block from the reference frame that yields the lowest MAD is then used as the predicted data for the block being coded.

The core operation for calculating the MAD between two blocks of data is the sum of absolute difference. This can be performed on the PP in two data unit operations:

- The first operation takes the difference between the pixels in the two blocks.
- The second operation adds the absolute value of the difference to a running sum of the total absolute difference.

To achieve the highest performance, you should use multiple-byte arithmetic. Example 15–12 shows that by tracking carries in the mf register (when the error sum overflows eight bits), it is possible to compute eight sums of absolute difference in four instructions.

Figure 15–3 illustrates the computations of the sum of absolute differences performed in one pass through the motion estimation inner loop. The first data unit operation in the loop performs a multiple-byte subtract between four pixels in the current frame and the corresponding (for the motion displacement being tested) pixels in the reference frame.

The carry ($\overline{\text{borrow}}$) from each of the byte subtracts is saved to the mf register. This bit indicates the sign of the difference (carry = 1 indicates a positive difference, carry = 0 indicates a negative difference). In the next data unit operation, the carries saved in the mf register are used to generate a mask.

This mask determines whether each individual 8-bit difference term is added to (in the case of a positive difference) or subtracted from (in the case of a negative difference term) the running sum. This effectively adds four byte-size absolute differences to the running four-byte error sums.

Thus in two data unit operations, four-byte sums of absolute differences are performed. The next data unit operation rotates the carry bits in the mf register left by four so that they will not be overwritten. Also, it performs a multiple-byte subtract between four additional current and previous frame pixels (which have been loaded by parallel transfers). Then, another sum of absolute differences computation is performed. At this point, there are eight carry flags in the mf register. Each carry flag indicates an absolute difference of 256.

In the next pass through the loop, while two more multiple-byte sums of absolute differences are being performed, some of the parallel transfers are used to look up the number of carries from the previous two multiple-byte sums of absolute differences. Address unit arithmetic is used to add the number carries from the previous pass through the loop (determined by the table look-up) to the total number of carries. After the inner loop is completed, the four 8-bit absolute difference sums are combined into one 10-bit sum. Also, the total number of carries is multiplied by 256 and added to the 10-bit combined difference sum. The result is the total absolute difference between the current and previous block of data.

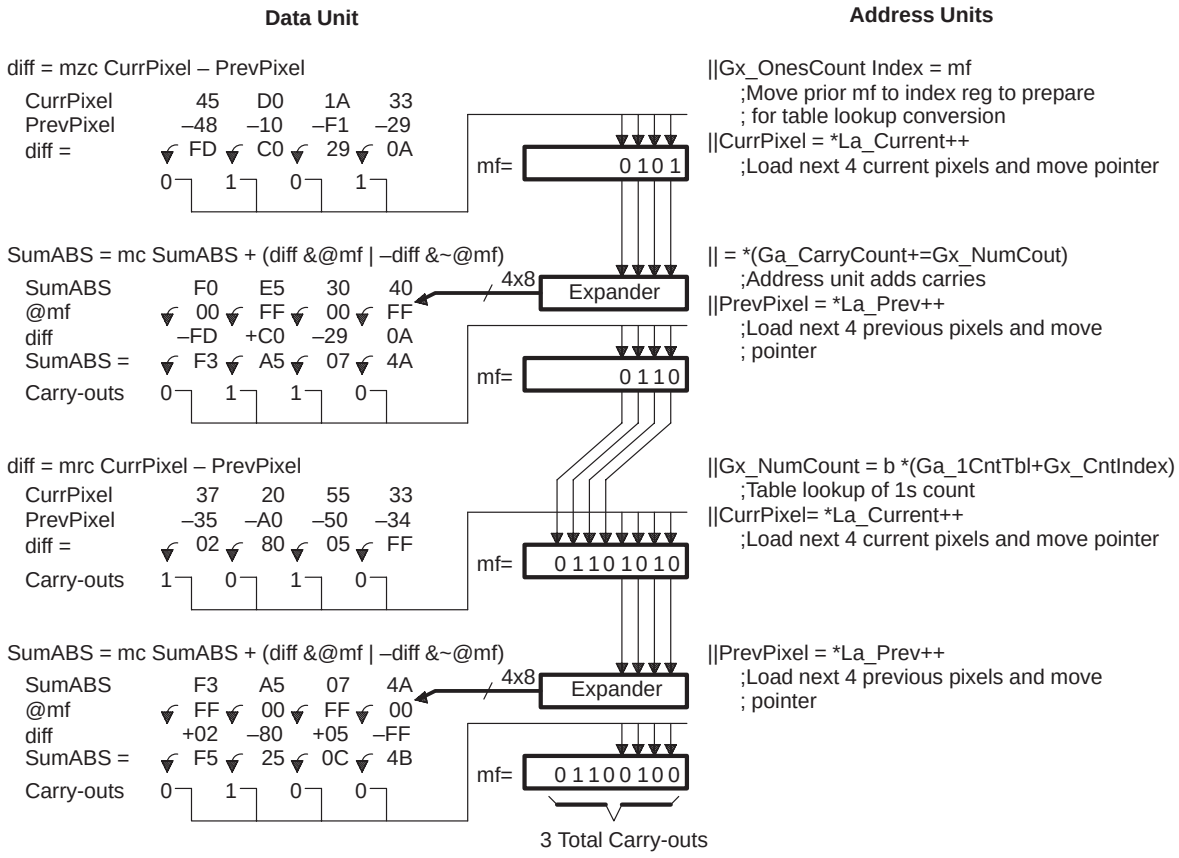
Note:

If either the current or previous frame block is not word-aligned in the shared RAMs, an extra two cycles per loop are required to merge the required bytes from two different words into a register for comparison.

Example 15–12. Motion Estimation Inner Loop Code for Absolute Differences

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
diff:              .set  d7 ; Contains the 8 LSBs of 9 bit result
                            ;  for 4 Pixel Differences.
CurrPixel:         .set  d6 ; Word containing 4 pixels from
                            ;  current frame.
PrevPixel:         .set  d5 ; Word containing 4 pixels from
                            ;  previous frame.
SumABS:            .set  d4 ; Running 4-byte sums of the
                            ;  total Absolute difference
                            ;  between pixels in the current
                            ;  and previous frame.
Gx_OnesCountIndex: .set  x8 ; 8 LSBs contain the carries recorded from
                            ;  the 8 previous pixel differences.
Gx_NumCout:        .set  x9 ; Number of carries determined by a
                            ;  table lookup of Gx_OnesCountIndex
La_Current:        .set  a0 ; Pointer to current block data.
La_Previous:       .set  a1 ; Pointer to previous block data.
Ga_1CntTbl:        .set  a8 ; Base pointer to 1s count table.
Ga_CarryCount:     .set  a9 : Running Sum of 1s count.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   diff =mzc CurrPixel – PrevPixel         ; 4 Pixel Differences
      ||Gx_OnesCountIndex = mf             ; Move prior mf to index reg
                                           ;  to prepare for Table
                                           ;  Lookup of Carries
      ||CurrPixel = *La_Current++          ; Load next 4 current pixels
                                           ;  and move pointer

   SumABS =mc SumABS + ((diff&@mf) | (–diff&~@mf))
                                           ; Absolute Value and Sum
      ||a15 = *(Ga_CarryCount+= Gx_NumCout) ; Address Unit Add of carries.
      ||PrevPixel = *La_Previous++         ; Load next 4 previous pixels
                                           ;  and move pointer

   diff =mrc CurrPixel – PrevPixel         ; 4 Pixel Differences
      ||Gx_NumCout =b *(Ga_1CntTbl+ Gx_OnesCountIndex)
                                           ; Table Lookup of 1’s Count
      ||CurrPixel = *La_Current++          ; Load next 4 current pixels
                                           ;  and move pointer

   SumABS =mc SumABS + ((diff&@mf) | (–diff&~@mf))
                                           ; Absolute Value and Sum
;;    ||Free Global Transfer Slot
      ||PrevPixel = *La_Previous++         ; Load next 4 previous pixels
                                           ;  and move pointer
```

Figure 15–3. Sample Data for a Motion Estimation Inner Loop



15.2.4 Discrete Cosine Transform (DCT)

The DCT is the transform of choice for many data compression standards such as JPEG, CCITT H.261, and MPEG. The following sections describe EALU features that are particularly useful for implementing fast DCT algorithms on the PP.

15.2.4.1 Rounding

For highest performance, you should use halfword multiple arithmetic to implement fast DCT and IDCT algorithms. IDCT implementations for video compression (for example, CCITT H.261 and MPEG) must satisfy a stringent precision specification to avoid excessive compounding of errors.

In order to meet this specification, rounding must be performed instead of truncation when the 16 MSBs of the 32-bit multiply result are obtained. The PP supports a rounding mode for MPY||EALU operations that adds 1 to bit 15 of the multiply result; the carry from this add is added to bit 16, therefore, if bit 15 is a 1, rounding up occurs; if bit 15 is 0, rounding down occurs.

If you use this rounding mode and an appropriate fast IDCT algorithm (such as the Chen IDCT), the IDCT precision specification for H.261 and MPEG can be met with 16×16 multiplies and halfword multiple arithmetic.

A halfword swapper included in the multiplier data path allows back-to-back 16×16 multiplies to produce two rounded 16-bit results packed into a single 32-bit word. This feature is discussed in Section 8.4, *The Extended Set of ALU Operations (EALU)*.

15.2.4.2 Default Multiply Shift

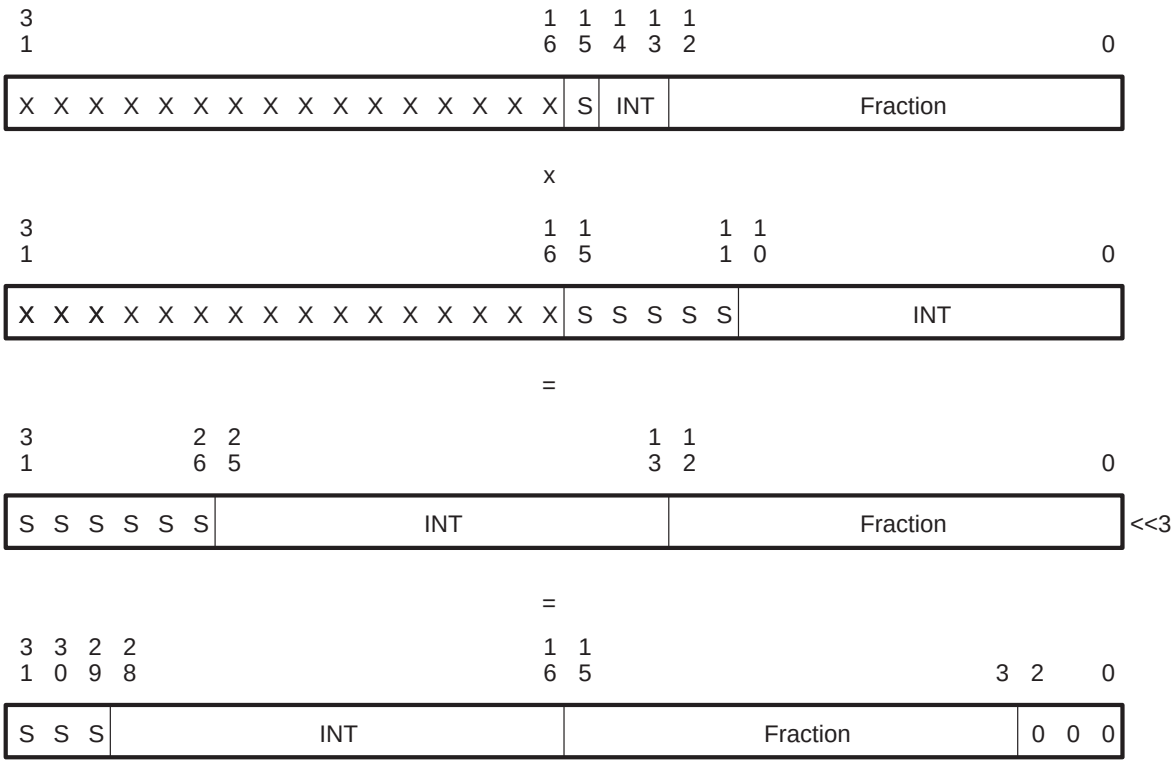
A typical IDCT implementation requires a multiply of a signed 12-bit integer DCT coefficient (sign-extended to 16 bits) by a positional weighting fraction. For example, the 8-point Lee Fast IDCT algorithm (1 × 8) uses weights ranging from 1/2 to 3.

This requires a sign bit and two bits of integer, leaving 13 bits of fraction to fill out a 16-bit multiply input. As shown in Figure 15–4, a 16 × 16 multiply between a sign-extended 12-bit integer and a signed weighting fraction (with 1 sign bit, 2 integer bits, and 13 bits of fraction) produces a 32-bit result with the integer portion of the result starting at bit 13.

To prepare this result for a subsequent halfword multiple arithmetic operation, a shift left by 3 is required to align the result to the upper halfword. For efficiency, MPY||EALU allows a post shift of the output of the multiplier (before the optional rounding).

The shift left amount is specified in d0 and can be 0, 1, 2, or 3 bit positions. In Figure 15–4, if the default multiply shift amount in d0 (bits 12–8) is set to 3, the integer portion of the multiplier output is aligned to bit 16 before the result is written to the destination register.

Figure 15–4. Default Multiply Shift



15.2.4.3 ealuf()

Transform operations such as the DCT and IDCT often involve both multiply-shift-and-add and multiply-shift-and-subtract operations in the same inner loop. Both of these operations require MPY||EALU. To support both of these operations with the same d0 setup, MPY||EALU lets you use either the ALU function code in d0 (ealu()) or the inverse of the ALU function code in d0 (ealuf()) to specify EALU operations performed in parallel with the multiply.

When the same label is associated with more than one operation, the assembler attempts to find a d0 set up capable of supporting all of the operations. The assembler is intelligent enough to identify inverse operations that can be supported with the same d0 setup by taking advantage of ealu() and ealuf(). In Example 15–13, the assembler determines the 32-bit constant required in d0 to perform the operations associated with the label MPY_SRA. The assembler finds that a multiply-shift-and-add and a multiply-shift-and-subtract can be performed with the same set-up in d0 because they are inverse operations. The assembler will therefore encode the multiply-shift-and-add with ealu() and the multiply-shift-and-subtract with ealuf().

Example 15–13. Add and Subtract With Same EALU Setup

```
d0 = MPY_SRA        ; Set d0 to constant determined
                    ;  by the assembler that allows
                    ;  operations associated with
                    ;  label MPY_SRA to be performed.

d7 = d6 * d5
  ||d4 = ealu(MPY_SRA: d3 + d2>>1)

d7 = d6 * d5
  ||d4 = ealu(MPY_SRA: d3 – d2>>1)
```

15.2.4.4 Mixing Halfword Adds and Halfword Subtracts

When writing tight loops for fast implementations of frequency domain transforms such as fast DCT algorithm, you can typically use halfword multiple arithmetic. For an efficient fast algorithm implementation, it is often useful to perform an add in the upper halfword segment of the ALU and a subtract in the lower halfword segment of the ALU (or vice versa).

Example 15–14 shows how to use an EALU to perform an add between upper halfwords and a subtract between the lower halfwords in the same operation (or vice versa). Note that the input to the C port controls the operation performed by each individual segment of the split ALU.

The example also shows how the assembler determines the 32-bit constant required in d0 to perform the operation associated with the label h1add_h0sub. The ALU function determined by the assembler is A + ((B&~C) | (–B&C)) with the E bit set, causing the carry-in to each segment of the split ALU to be a function of the input to the C port (see subsection 13.2.4, *Explicit-Multiple Carry-In (E Bit)*, for details on the E bit).

The %% operator in the EALU sets the bit in h1add_h0sub. This causes the mask generator to produce 0x0000 FFFF from the input of 16, even though the ALU is split. This provides the 16 1s mask that is needed to cause the add and subtract at the same time in the ALU.

Example 15–14. MSB Halfword Add, LSB Halfword Subtract

```
d0 = h1add_h0subd7 =m ealu(h1add_h0sub: d6 + ((d5 & ~%%16) | (–d5 & %%16)))
```

15.3 Bitstream Decoding Applications

JPEG, P × 64, MPEG, and other compression/decompression (codec) standards define a coded bitstream format that contains variable length codes (VLCs). In order to decode a VLC bitstream, it is necessary to track the current location of the bitstream pointer.

Lookup tables (LUTs) are used to identify the represented data value and number of bits for the next VLC in the bitstream. Since the VLCs can be long (up to 20 bits for h.261), several look-ups can be required to identify a code.

For certain VLCs (for example, the h.261 MTYPE), the bit-detection logic can be used to reduce the required table size and the number of LUTs required.

15.3.1 Tracking the Bitstream Pointer

At any given time, the next VLC in the bitstream may cross over word boundaries in memory. One way to handle this problem (as shown in Example 15–15) is to merge the remaining bits from the current word and the first bits of the next word into a single word containing the next 32 bits of undecoded bitstream.

This can be performed between every decoded VLC or, for sake of efficiency, at least often enough to ensure that the number of bits decoded in between merges does not exceed 32.

As shown in Example 15–15, two 32-bit aligned words from memory are required: word1 (contains the leftmost undecoded bit), and word2 (next 32-bit aligned word in memory after word1). Also, it is necessary to track the number of bits in word1 that have been decoded.

Three actions occur when the number of decoded bits in word1 exceeds or equals 32:

- word2 becomes word1,
- The bitstream pointer is incremented and a new word2 is loaded, and
- The number of decoded bits in word1 is decremented by 32 to reflect the number of decoded bits in the new word1.

Example 15–15. Merge 32-Bit Bitstream Word

```
num_decoded_bits:  .set  d7       ; Number of bits already decoded in word1
bstream:           .set  d6       ; Next 32-bits in the bitstream
num_bits:          .set  d5       ; Number of bits in previous VLC
modulo:            .set  d4       ; num_decoded_bits – 32
word1:             .set  d3       ; Aligned word in memory that contains the
                                  ;  leftmost undecoded bit in the
                                  ;  bitstream
word2:             .set  d2       ; Aligned word in memory after the word1
Ga_BitStream:      .set  a8       ; Bitstream pointer

   num_decoded_bits = num_decoded_bits + num_bits
                                  ; Add number of previous VLC bits to
                                  ;  number of used bits in word1.

   modulo = num_decoded_bits – 0x1\\5 ; Test for >= 32 used bits in word1.

   num_decoded_bits =[nn.nz] modulo   ; If all bits in word1 decoded, adjust
                                      ;  number of decoded bits for new word1
      ||word1 =[nn] word2             ; and make word2 new word1.

   word2 =[nn] *Ga_BitStream          ; If new word1, load a new word2.

   bstream =[u.z] word1<<num_decoded_bits         ; Left-justify undecoded
                                                  ; bits in word1
      ||Ga_BitStream =[nn] &*(Ga_BitStream + [1]) ; Increment bitstream
                                                  ;  pointer if word2 became
                                                  ;  new word1.

   bstream =[nz] bstream + word2>>u -num_decoded_bits
                                  ; Merge undecoded bits from word1 with
                                  ;  appropriately shifted left-most
                                  ;  bits from word2 to form a 32-bit word.
```

15.3.2 Iterative Look-Up Tables

When you decode a bitstream, you should keep all of the required tables in on-chip memory. In order to reduce the required table size, VLC decode routines often use iterative look-up tables. Since these table are set up so that the shorter codes occur most of the time, usually only one look-up is required. For the rare occurrences of longer codes, two or more look-ups may be required.

To demonstrate how a VLC table and bitstream decoding routine can be implemented, the code for decoding the coded block pattern in an h.261 bitstream is presented. Table 15–1 shows the code table for the CBP field of a macroblock (six 8×8 byte blocks: four Luma, one Cb, and one Cr). Note that the code size varies from three to nine bits. Thus, decoding the CBP by using a single table for all codes is prohibitive (it would required 2^9 table entries). Instead, it is better to break up the decode into iterative LUTs. For this example, it is efficient to perform the initial table look-up on five bits. The value returned from the first LUT indicates either that a full VLC is contained in the five bits (in which case, the table also indicates the number of bits in the VLC) or that a second look-up is required on the next four bits in the bitstream. The format for the two tables is shown in Example 15–16.

Table 15–1. Coded Block Pattern VLC Table

CBP Pattern	Codeword Length	Codeword	CBP Pattern	Codeword Length	Codeword
111100	3	111	100011	8	00011100
000100	4	1101	001101	8	00011011
001000	4	1100	110001	8	00011010
010000	4	1011	010101	8	00011001
100000	4	1010	101001	8	00011000
001100	5	10011	001110	8	00010111
110000	5	10010	110010	8	00010110
010100	5	10001	010110	8	00010101
101000	5	10000	101010	8	00010100
111000	5	01111	001111	8	00010011
101100	5	01110	110011	8	00010010
110100	5	01101	010111	8	00010001
111000	5	01100	101011	8	00010000
000001	5	01011	011001	8	00001111
111101	5	01010	100101	8	00001110
000010	5	01001	011010	8	00001101
111110	5	01000	100110	8	00001100
011000	6	001111	011101	8	00001011
100100	6	001110	101101	8	00001010
000011	6	001101	110101	8	00001001
111111	6	001100	111001	8	00001000
000101	7	0010111	011110	8	00000111
001001	7	0010110	101110	8	00000110
010001	7	0010101	110110	8	00000101
100001	7	0010100	111010	8	00000100
000110	7	0010011	011111	9	000000111
001010	7	0010010	101111	9	000000110
010010	7	0010001	110111	9	000000101
100010	7	0010000	111011	9	000000100
000111	8	00011111	011011	9	000000011
001011	8	00011110	100111	9	000000010
010011	8	00011101			

Example 15–16. CBP Decode Routine

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; CBP Decode                                                              ;;
;; Description:                                                            ;;
;;    An indexed look-up is performed based on the top five bits of        ;;
;;    the right-justified bitstream. This code contains either an          ;;
;;    index to a second table in which case the top 2 bits are 0, or       ;;
;;    a 6-bit code (1 bit corresponding to each block).                    ;;
;;                                                                         ;;
;;    Table1                                                               ;;
;;       Indexed by 5 leftmost bits when the CBP code is                   ;;
;;       left justified in the bitstream. The 8-bit table entries          ;;
;;       signify the following:                                            ;;
;;                                                                         ;;
;;       bit    7  6  5  4  3  2  1  0     Code Size                       ;;
;;              0  0  index to next table  (greater than 5 bit code)       ;;
;;              0  1  p1 p2 p3 p4 p5 p6    (3-bit code)                    ;;
;;              1  0  p1 p2 p3 p4 p5 p6    (4-bit code)                    ;;
;;              1  1  p1 p2 p3 p4 p5 p6    (5-bit code)                    ;;
;;                                                                         ;;
;;    Table2 (used only if CBP code greater than 5 bits)                   ;;
;;                                                                         ;;
;;       bit    7  6  5  4  3  2  1  0     Code Size                       ;;
;;              0  0  p1 p2 p3 p4 p5 p6    (6-bit code)                    ;;
;;              0  1  p1 p2 p3 p4 p5 p6    (7-bit code)                    ;;
;;              1  0  p1 p2 p3 p4 p5 p6    (8-bit code)                    ;;
;;              1  1  p1 p2 p3 p4 p5 p6    (9-bit code)                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   .global   CBP_DEC
   .global   CBP_DEC_e

num_decoded_bits:  .set   d7
bstream:           .set   d6
num_bits:          .set   d5
num_bits_code:     .set   d3
Val1:              .set   d1

La_Table1:         .set   a0
Lx_CBP_Index1:     .set   x0

                                     ; input is left_justified bitstream
CBP_DEC:
   Lx_CBP_Index1 = bstream>>u 27     ; Right-Justify 5 bits to form index
                                     ;  to CBP table 1.

   num_bits_code = num_bits_code + 2 ; Add difference between num_bits code
                                     ;  from table and actual number of
                                     ;  bits. (for example, if num_bits
                                     ;  code is 01, number of bits is
                                     ;  actually 3 so need to pre-add 2).

   Val1 =ub *(La_Table1 + [Lx_CBP_Index1])  ; Table Look-up on 5 bits.

   num_bits_code = Val1>>u 6         ; Extract the code-length code from the
                                     ;  table entry.

   br =[z] Second_CBP_LUT            ; If code is 00,then branch to routine
                                     ;  that performs a second table lookup.
```

Example 15–16. CBP Decode Routine (Continued)

```
   num_decoded_bits =[nz.z] num_decoded_bits + num_bits
                                    ; Increment the number of decoded bits
                                    ;  in word1 by the used bits code
                                    ;  (either 1, 2, or 3) if a second
                                    ;  look-up is not required.
       .
       .
       .

Second_CBP_LUT:
       .
       .
       .
```

15.3.3 Using Bit-Detection Logic

For certain VLC tables, the bit-detection logic can be used to simplify the decoding process. A good example is the MTYPE field defined by CCITT recommendation h.261, which identifies the type of coding used for a macroblock.

There are ten VLCs ranging from one to ten bits. These codes are basically a 1 preceded by zero to nine 0s. Example 15–17 shows both a table for decoding the information specified by the MTYPE code and also the code for decoding a single MTYPE VLC.

This MTYPE VLC code follows the bitstream decoding procedure specified in subsection 15.3.1. The example also shows the code for maintaining the bitstream pointer to illustrate how it can be interleaved in the delay slots required by the VLC LUT access.

The first step in decoding the VLC is to obtain the leftmost one in the left-justified bits corresponding to the coded MTYPE. This value is then used as an offset into the MTYPE VLC table.

Example 15–17. h.261 MTYPE Decode Routine

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; MTYPE Decode                                                             ;;
;; Description:    Decode the VLC for MTYPE.   The index into the MTYPE     ;;
;;        table shown below is determined by the leftmost-one               ;;
;;        in bstream (the next 32 bits in the bitstream that                ;;
;;        need to be decoded).  Each byte-sized table entry                 ;;
;;        has 1 bit for each of the following parameters:                  ;;
;;                                                                          ;;
;;        bit 0: INTRA    (1 if INTRA block coding is used)                 ;;
;;        bit 1: MQUANT   (1 if MQUANT field is used)                       ;;
;;        bit 2: MC       (1 if MVD field is used)                          ;;
;;        bit 3: CODED    (1 if CBP field is used)                          ;;
;;        bit 4: LF       (1 if loop filter is used)                        ;;
;;                                                                          ;;
;; VLC    Table Offset    Table Entry     Block Parameters                  ;;
;; ---    ------------    -----------     -----------------                 ;;
;; 1           0          00001000        Inter coded block with no         ;;
;;                                           motion compensations           ;;
;; 01          1          00011100        Inter coded block with loop       ;;
;;                                           filtering and motion comp.     ;;
;; 001         2          00010100        Inter coded block with motion     ;;
;;                                           compensation and loop          ;;
;;                                           filtering. No coded error      ;;
;;                                           terms.                         ;;
;; 0001        3          00000001        Intra coded.                      ;;
;; 00001       4          00001010        Inter coded with no motion        ;;
;;                                           compensation. New quantization ;;
;;                                           stepsize.                      ;;
;; 000001      5          00011110        Inter code with motion comp       ;;
;;                                           loop filter.  New quantization ;;
;;                                           stepsize.                      ;;
;; 0000001     6          00000011        Intra coded. New quantization     ;;
;;                                           stepsize.                      ;;
;; 00000001    7          00001100        Inter coded with motion comp.     ;;
;; 000000001   8          00000100        Inter coded with motion comp.     ;;
;;                                           No coded error terms.          ;;
;; 0000000001  9          00001110        Inter coded with motion comp.     ;;
;;                                           New quantization stepsize.     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

num_decoded_bits:  .set  d7 ; Number of bits already decoded in word1
bstream:           .set  d6 ; Next 32 bits in the bitstream
num_bits:          .set  d5 ; Number of bits for the VLC
modulo:            .set  d2 ; num_decoded_bits - 32
word1:             .set  d0 ; Aligned word in memory that contains the
                            ;  leftmost undecoded bit in the bitstream
word2:             .set  d4 ; Aligned word in memory after the word1
MTYPE:             .set  d1 ; Table entry corresponding to lmo value
                            ;  returned by bstream
La_Table1:         .set  a0 ; Pointer to MTYPE table
Ga_BitStream:      .set  a8 ; Bitstream pointer
Ga_Out:            .set  a9 ; Decoded MTYPE output pointer
Lx_MTYPE_Index:    .set  x0 ; lmo value for bstream used as an index
                            ;  into the MTYPE table

   .global  MTYPE_DEC
   .global  MTYPE_DECe
   .ptext
```

Example 15–17. h.261 MTYPE Decode Routine (Continued)

```
MTYPE_DEC:
   num_bits = lmo(bstream)      ; Determine index into MTYPE table.
                                ; NOTE: Destination of lmo must be a D
                                ;  register.

   num_bits = num_bits + 1      ; Adjust lmo output for number
                                ;  of VLC bits.
   ||Lx_MTYPE_Index = num_bits  ; Move index to Index Register.

   num_decoded_bits = num_decoded_bits + num_bits
                                ; Add number of VLC bits to
                                ;  number of used bits in word1.

   modulo = num_decoded_bits – 0x1\\5
                                ; Test for >= 32 used bits in word1.
   ||MTYPE =ub *(La_Table1 + [Lx_MTYPE_Index])
                                ; Look-up MTYPE.

   num_decoded_bits =[nn.nz] modulo
                                ; If all bits in word1 used up,
                                ;  store used bits in word1 – 32.
   ||word1 =[nn] word2          ; and make word2 new word1.

   word2 =[nn] *Ga_BitStream    ; Load a new word2.

   bstream =[u.z] word1<<num_decoded_bits
                                ; Left-justify unused bits in word1.
   ||Ga_BitStream =[nn] &*(Ga_BitStream + [1])
                                ; Increment bitstream pointer, if
                                ;  word2 became new word1.
MTYPE_DECe:
   bstream =[nz] bstream + word2>>u –num_decoded_bits
                                ; Merge required bits from word1 to
                                ;  left-justified unused bits in word1
                                ;  to form a 32-bit word.
   ||*Ga_Out++ =b MTYPE         ; Store MTYPE information.
```

Appendix A

The PP Opcodes

This appendix describes the PP opcode in detail.

Topics

A.1	PP Opcode Summary	PP: A-2
A.2	Alphabetical Summary of Opcode Mnemonics	PP: A-4
A.3	Data Unit Opcode Formats	PP: A-8
A.4	Parallel Transfer Opcode Formats	PP: A-21

A.1 PP Opcode Summary

A PP instruction (referred to as a 64-bit operation code or opcode) is essentially divided into two parts:

- A data unit portion
- A parallel transfer portion

The full 64-bit opcode executed by the PP is illustrated in Figure A–1.

The data unit portion of the opcode is specified in the 25 MSBs of the instruction. These bits are categorized into one of five data unit opcode formats:

- Six-operand operations, including MPY||ADD, MPY||EALU, EALU||ROTATE, and divi (format A),
- Base set ALU with 5-bit immediate src2 (format B),
- Base set ALU with register src2 (format C),
- Base set ALU with 32-bit immediate src2, which requires the entire opcode (format D), or
- Miscellaneous operations that do not involve the data unit, including nop and eint (format E).

The parallel transfer portion of the opcode is specified in the 39 LSBs of the instruction. These bits are categorized into one of ten parallel transfer formats:

- Three parallel transfer formats allow both a global and local transfer.
- Two formats allow a single parallel transfer using a long offset (either a global or local transfer).
- One format allows a local transfer in parallel with a data unit operation using non-D register operands.
- Four formats allow conditional operations to be specified.

Figure A–1. Opcode Formats

(a) Data Unit Formats

63–32	31–29	28–26	25–23	22–20	19–17	16–14	13–11	10–9	8–0	Description
0 1 1	oper	src3	dst2	dst1	src1	src4	src2		Parallel Transfers	A. Six-Operand (MPY||ADD, etc.)
1	class	A	ALU Operation	dst	src1	0	imm. src2		Parallel Transfers	B. Base Set ALU (5-Bit Immediate)
1	class	A	ALU Operation	dst	src1	1 0	–	src2	Parallel Transfers	C. Base Set ALU (Register src2)
1	class	A	ALU Operation	dst	src1	1 1	dstbank	s1bnk	cond / 32-Bit Immediate	D. Base Set ALU (32-Bit Immediate)
1	0 0 0	1 – 0 – 0 – 0 – 0	– – – – – –			0	Operation		Parallel Transfers	E. Miscellaneous
0 0	Reserved									
0 1 0	Reserved									

(b) Transfers Formats for Parallel Transfers

38–32	31–29	28–26	25–23	22–20	19–17	16–14	13–11	10–9	8–0	Description
Lmode	d	e	size	s	La	Gim/X	L	0bank	L Gmode reg e size s Ga Lim/X	1. Double Parallel
Lmode	d	e	size	s	La	0 Lrm	dstbank	L	0 0 0 0 src srcbank dst Lim/X	2. Move||Local
Lmode	d	e	size	s	La	0 itm	dstbank	L	0 0 0 1 src e size D dst Lim/X	3. Field Move||Local
Lmode	d	e	size	s	La	1 Lrm	bank	L	0 0 Local Long Offset /X	4. Local (Long Offset)
0 0	Global Long Offset/X						bank	L	Gmode reg e size s Ga 0 Grm	5. Global (Long Offset)
Lmode	d	e	size	s	La	0 Lrm	Adstbnk	L	0 0 1 – – – – As1bank – – – Lim/X	6. Non-D DU||Local
0 0 –	cond	c r g	N C V Z			0 – –	dstbank	–	0 0 0 0 src srcbank dst – – –	7. Conditional DU||Conditional Move
0 0 –	cond	c r g	N C V Z			0 itm	dstbank	–	0 0 0 1 src e size D dst – – –	8. Conditional DU||Conditional Field Move
0 0 –	cond	c r g	N C V Z			Gim/X	bank	L	Gmode reg e size s Ga 1 Grm	9. Conditional DU||Conditional Global
0 0 –	cond	c r –	N C V Z			0 – –	Adstbank	–	0 0 1 – – – – As1bank – – – – – –	10. Conditional Non-D Data Unit

A.2 Alphabetical Summary of Opcode Mnemonics

Table A–1 and Table A–2 provide an alphabetical list of the mnemonics used in Figure A–1 for the data unit and parallel transfer portions of the opcode, respectively. The individual data unit and parallel transfer formats and the definitions for the corresponding opcode fields are described in more detail in see Section A.3, *Data Unit Opcode Formats*, and Section A.4, *Parallel Transfer Opcode Formats*, respectively.

Table 7–1, *The Register Codes*, shows the encodings used within the opcode fields to specify particular registers.

When a 3-bit field is used for a register code, only the three LSBs of the coding are present. The four MSBs of a register code are the register bank. In some opcode formats, where there is no register bank specified, the D-register bank is assumed. When a register bank is the specified in an opcode, the bank code is concatenated to the corresponding 3-bit register code to form a full 7-bit register code.

In a few opcode formats, the MSB of the 4-bit bank code cannot be specified. In these cases, it is assumed to be 0, limiting the register to be within the left half of the table. (These are referred to as the lower registers because they occupy the least significant half of the register space.)

Table A–1. Data Unit Mnemonics

Mnemonic	Function
A	Selects arithmetic operations in the ALU. A = 1 indicates arithmetic; A = 0 indicates Boolean.
ALU Operation	❑ When A = 0, the 8 bits in the ALU operation field correspond directly to the eight ALU function signals that determine the Boolean ALU operation. ❑ When A = 1, the odd bits specify an arithmetic ALU function, while the even bits define ALU function modifiers.
class	Specifies the operation class, which determines the routing of operands to the ALU.
cond	Specifies the condition code for conditional operations.
dst	Specifies the D register destination or the lower three bits of the register coding for a non-D destination.
dst1	Specifies the ALU destination for MPY||ADD, MPY||EALU, or EALU||ROTATE operations. It indicates a D register or the lower three bits of a non-D register.
dst2	Specifies the multiply destination D register for MPY||ADD or MPY||EALU operations or the rotate destination for EALU||ROTATE operations.
dstbank	Defines the ALU destination register bank. This allows the ALU destination register to be any PP register.
imm. src2	Specifies a 5-bit immediate value for src2 used in the ALU operation.
32-Bit Immediate	Specifies a 32-bit immediate value for src2 used in the ALU operation.
oper	Selects one of the following six-operand data unit operations: ❑ A multiply with a parallel add or subtract (MPY||ADD) ❑ A multiply with a parallel shift and add or subtract (MPY||SADD) ❑ A multiply with a parallel extended ALU operation (MPY||EALU) ❑ An extended ALU operation with optional assignment of the barrel rotator output to a destination register (EALU||ROTATE) ❑ A divide iteration operation (divi)
Operation	Selects one of the miscellaneous operations such as nop, dint, or eint.
src1	Specifies the three-bit code for the register used as ALU source 1. This defines a D register by default, unless srcbank or s1bank is used.
src2	Specifies the D register used as ALU source 2.
src3	Specifies the D register used as the rotate amount for EALU||ROTATE operations or the multiplier source 1 for MPY||ADD or MPY||EALU operations.
src4	Specifies the D register used as the ALU C port operand or mask generator input for EALU||ROTATE operations or the multiplier source 2 for MPY||ADD or MPY||EALU operations.
s1bnk	Specifies the src1 register bank, but is limited to the lower non-D registers. It specifies bits 5–3 of the register code of the bank containing the src1 register for the ALU operation. Bit 6 of the register code is assumed to be 0. This allows the ALU source 1 register to be any PP register other than the tag registers or the looping registers (lctl can also be specified). See Table 7–1, *The Register Codes.*

Table A–2. Parallel Transfer Mnemonics

Mnemonic	Function
0bank	Specifies bits 5–3 of the register code for global transfer source (L = 0) or destination (L = 1). Bit 6 of the register code is assumed to be 0. This allows the global transfer source for stores or destination for loads to be any PP register other than the tag registers or the looping registers (lctl can also be specified). See Table 7–1, *The Register Codes*.
Adstbnk	Specifies bits 6–3 of the register code for the destination of an ALU operation. This allows the ALU destination to be any PP register.
As1bank	Specifies bits 6–3 of the register code for source1 of an ALU operation. Used with the src1 field of data unit part of opcode to allow src1 to be any PP register.
bank	Specifies bits 6–3 of the register code for global (or local) store source (L=0) or global (or local) load destination (L = 1). This allows any PP register to be specified. See Table 7–1, *The Register Codes*.
c	Indicates a conditional choice of a D register from a D register pair for source operand 1 of the data unit operation.
C	Specifies the protection of the status register's carry bit.
cond	Indicates a condition code. See Table A–8.
d	Specifies a data register, d0–d7, for a local transfer load destination or store source or if combined with bank in format 4, it specifies the three LSBs of the source or destination register code.
D	Specifies the duplication of the least significant data during moves. D is set by the r (replicate) assembler modifier option.
dst	Specifies the 3 LSBs of the move or field move destination register.
dstbank	Specifies bits 6–3 of the register code for the destination of a move. This allows the destination of a move or field move to be any PP register.
e	Indicates sign-extend select: bit 31 for local, bit 9 for global. e = 1 and L = 0 selects address unit arithmetic. (See subsection A.4.1.)
g	Indicates conditional global transfer. The g bit specifies conditional source selection on store operations that use the global address unit and conditional write to destination for move and load operations.
Ga	Specifies the global address register for a global unit load, store, or address unit arithmetic.
Gim/X	Specifies the global address unit index (immediate offset or global index register).
Gmode	Specifies the global address unit addressing mode. See Table A–7.
Grm	Specifies the global PP-relative addressing mode.
itm	Specifies the number of the item selected during a field extract move.
L	Specifies whether a transfer is a load (L = 1) or a store/address unit arithmetic (L = 0). (See subsection A.4.1.)
La	Specifies the local address register for a local unit load, store, or address unit arithmetic.
Lim/X	Specifies the local address unit index (immediate offset or local index register).
Lmode	Specifies the local address unit addressing mode. See Table A–7.

Table A–2. Parallel Transfer Mnemonics (Continued)

Mnemonic	Function
Lrm	Specifies the local PP-relative addressing mode.
N	Specifies protection of the status register's negative bit.
r	Specifies conditional write of the ALU result.
reg	Indicates the number of the register used with bank or 0bank to specify the register accessed in the global load, store, or address unit arithmetic.
s	Enables index scaling or defines offset, according to the data size. Bit 28 for local, bit 6 for global. For byte accesses or byte address unit arithmetic operations that use an immediate index, the s bit is used as an additional index bit, providing a 4-bit immediate instead of 3-bit, or a 16-bit immediate instead of a 15-bit immediate.
size	Specifies size of data being transferred. Bits 30–29 for local, bits 8–7 for global.
src	Specifies the three LSBs of the register code for the source of a register-to-register move. For non-field moves (formats 2 and 7), used with srcbank to specify any PP register. For field moves (formats 3 and 8), it specifies a D register source.
srcbank	Specifies the four MSBs of the register code for the source of a register-to-register move.
V	Specifies the protection of the status register's overflow bit.
Z	Specifies the protection of the status register's zero bit.
–	Represents an unused opcode bit. The assembler will fill the bit with 0.

A.3 Data Unit Opcode Formats

The data unit opcode formats and their associated fields are described in the following sections. Section A.4, *Parallel Transfer Opcode Formats*, describes certain parallel transfer formats that dedicate bits to specify additional information regarding the data unit operation:

- Bank codes for the ALU destination and/or ALU data path source 1 (src1). This allows these operands to be non-D registers.
- A condition code and bits that control conditional assignment of the result of the data unit operation and conditional source selection of ALU data path source 1 (src1).

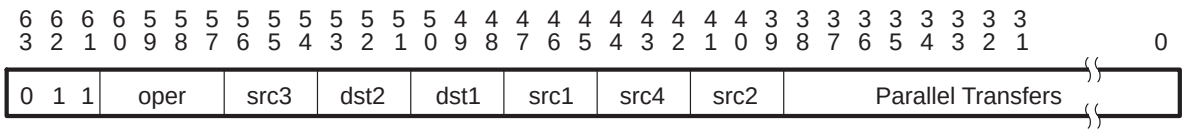
A.3.1 Data Unit Opcode Format A: Six-Operand

This opcode format allows six data unit operands to be specified in the instruction, as shown in Figure A–2. A 4-bit operation code specifies the type of operation to be performed:

- MPY||ADD,
- MPY||SADD,
- MPY||EALU,
- EALU||ROTATE, or
- divi.

Each of these operations support parallel writes to two destinations: dst1 for the result of the ALU operation, and dst2 for either the result of the multiplier data path or the output of the barrel rotator. Several of these operations specify an extended ALU (EALU) operation. For EALUs, the operation performed by the ALU data path is configured by fields in the d0 register.

Figure A–2. Data Unit Opcode Format A: Six-Operand



The following text describes the fields in the data unit opcode format A.

❑ **oper** (bits 60–57)—Data unit operation code

This 4-bit code designates 1 of 16 six-operand operations and an associated operand data path. The 4-bit operation codes, 0u0s, 0u1f, 10fk, 1010, and 11us are shown in the following table.

oper Field Bit Number					
60	**59**	**58**	**57**	**Operation Type**	**Number of Variations**
0	u	0	s	MPY||ADD	4
0	u	1	f	MPY||EALU	4
1	0	f	k	EALU||ROTATE	3
1	0	1	0	divi	1
1	1	u	s	SPY||SADD	4

■ **0u0s**—MPY||ADD

A signed (u = 0) or unsigned (u = 1) multiply of the 16 LSBs in src3 by the 16 LSBs in src4 is performed, producing either a signed or unsigned 32-bit result written to dst2.

In parallel, src1 is added to (s = 0) or subtracted from (s = 1) src2, and the result is written to dst1. Refer to subsection 8.5.1, *Using the MPY||ADD and MPY||SADD Operations*, for details on the assembly syntax and functionality of these operations.

■ **0u1f**—MPY||EALU

A signed (u = 0) or unsigned (u = 1) multiply of src3 by src4 as configured by bits 9–6 of d0 is performed, and the result is written to dst2. In parallel, an ALU data path operation as configured by bits 31–14 of d0 with src1 and src2 as inputs is performed with the result written to dst1.

The input to the A port of the ALU is src2. The input to the B port of the ALU is src1, rotated left by the barrel rotate amount specified in bits 4–0 of d0. The input to the C port is a mask generated from bits 4–0 of d0 if bit 13 of d0 is 0 or from @mf if bit 13 of d0 is 1.

The ALU function code is specified either by bits 27–19 of d0 (f = 0) or the 1s complement of bits 27–19 of d0 (f = 1). Refer to subsection 8.5.2, *Multiplying in Parallel With an EALU Operation*, for details on the assembly syntax and functionality of these operations.

- **10fk**—EALU||ROTATE

  An ALU data path operation configured by bits 31–14 of d0 is performed, and the result is written to dst1. The input to the A port of the ALU is src2. The input to the B port of the ALU is src1 rotated by the barrel rotate amount specified in bits 4–0 of src3.

  The output of the barrel rotator can be written to dst2 and is also used as the input to the B port of the ALU. If dst2 is the same register as dst1, the output of the barrel rotator is not saved in dst2. The input to the C port can be either src4 (k = 0), a mask generated from bits 4–0 of src4 (k = 1, d0⟦F⟧ = 0), or the output of the mf expander (k = 1, d0⟦F⟧ = 1).

  The ALU function code is specified either by bits 27–19 of d0 (f = 0) or the 1s complement of bits 27–19 of d0 (f = 1). The case in which f = 0 and k = 0 is the code for a divi operation, Therefore, ealuf with src4 as the direct input of the C port of the ALU is not supported.

  Refer to subsection 8.4.6, *Using EALU Operations Without a Parallel Multiply*, for details on the assembly syntax and functionality of the EALU||ROTATE operations.

- **1010**—divi

  The ALU performs an unsigned divide iterate operation. Refer to Section 8.6, *The Divide Operation*, for details on the assembly syntax and functionality of this operation.

- **11us**—MPY||SADD

  A signed (u = 0) or unsigned (u = 1) multiply of the 16 LSBs in src3 by the 16 LSBs in src4 is performed, producing either a signed or unsigned 32-bit result written to dst2.

  In parallel, src1, shifted right by the 2s complement of the five LSBs of d0, is added to (s = 0) or subtracted from (s = 1) src2, and the result is written to dst1. The right shift is arithmetic/signed (u=0) or unsigned (u=1), jointly with the multiply. Refer to subsection 8.5.1, *Using the MPY||ADD and MPY||SADD Operations*, for details on the assembly syntax and functionality of these operations.

❑ **src3** (bits 56–54)

This field specifies three LSBs of the D-register code for source operand 3. For operations involving a multiply, src3 is an inputs to the multiplier. For all other six-operand operations, src3 specifies the barrel rotate amount.

- **dst2** (bits 53–51)

  This field specifies the three LSBs of the D-register code for the destination of either the multiplier result or the output of the barrel rotator. If the oper field specifies a multiply, dst2 is the destination of the multiplier result. Otherwise, dst2 is the destination of the barrel rotator output (which is also fed into the B port of the ALU).

- **dst1** (bits 50–48)

  This field specifies the three LSBs of the register code for the destination of the ALU result. Parallel transfer formats six and ten let you specify a 4-bit bank code to allow the ALU destination to be any register. For all other parallel transfer formats, the D-register bank is implied.

- **src1** (bits 47–45)

  This field specifies the three LSBs of the register code for ALU data path source operand 1. Parallel transfer formats six and ten let you specify a 4-bit bank code, which allows source operand 1 to be any register. For all other parallel transfer formats, the D-register bank is implied.

  Parallel transfer formats 7, 8, 9, and 10 allow either the odd or even register in a D register pair to be conditionally selected as ALU data path source 1. When condition source selection is used, src1 must be an odd D register (d7, d5, d3, or d1).

  Depending on the six-operand operation being performed, src1 is used either as a direct input to the ALU B port or rotated by some amount and then input to the ALU B port.

- **src4** (bits 44–42)

  This field specifies the three LSBs of the D-register code for source operand 4. For MPY||ADD or MPY||EALU, src4 is an input to the multiplier. For EALU||ROTATE, src4 is a direct input to the C port of the ALU or can be used to generate a mask that is fed to the C port of the ALU. Src4 is not used for divi operations.

- **src2** (bits 41–39)

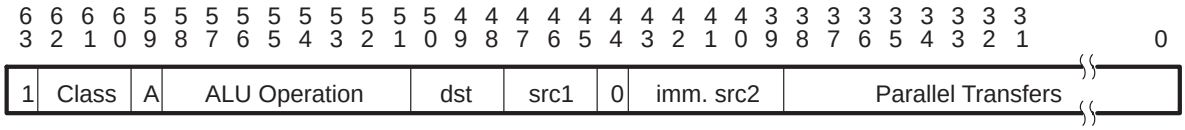
  This field specifies the three LSBs of the D-register code for ALU data path source operand 2. For all six-operand operations, src2 is a direct input to the A port of the ALU. src2 cannot be an immediate with the six-operand operations.

A.3.2 Data Unit Opcode Format B: Base Set ALU With 5-Bit Immediate src2

This format specifies the ALU operation directly in the instruction. It is one of the base set of ALU operations (or base set ALU). As described in Section 8.3, *The Base Set of ALU Operations*, the base set of ALU operations includes all of the 256 possible Boolean combinations of the three inputs to the ALU and a subset containing 19 of the most useful mixed arithmetic and Boolean operations.

Three operands are specified in the instruction, along with a 3-bit code that specifies an operation class that selects one of eight possible data routings into the ALU. See Figure A–3.

Figure A–3. Data Unit Opcode Format B: Base Set ALU With 5-Bit Immediate src2



The following text describes the fields in the data unit opcode format B.

- **Class** (bits 62–60)—Operation class

  This field specifies one of eight different routings of data to the A, B, and C ports of the ALU; the classes are shown in Table A–3. The operation classes are described in detail in subsection 8.3.3, *Operation Classes*.

Table A–3. Base Set ALU Operation Class Summary

Class	Destination	A Port	B Port			C Port
000	dst	src2	src1			@mf
001	dst	dstc	src1	\\	d0	src2
010	dst	dstc	src1			%src2
011	dst	dstc	src1	\\	src2	%src2
100	dst	src2	src1	\\	d0	%d0
101	dst	src2	src1	\\	d0	@mf
110	dst	dstc	src1			src2
111	dst	src1	1	\\	src2	src2

Notes:
1) \\ Rotate left
2) % Mask generation (2^n–1)
3) @mf Expand function
4) dst Destination D register or any PP register if dstbnk or Adstbnk is used with dst
5) dstc Companion D register for ALU destination (3-bit dst code with D register file bank)
6) src2 A D register or an immediate.
7) src1 It can be any PP register if src1 is used with As1bank or any lower register if src1 is used with s1bnk.

- **A** (bit 59)

  This bit is the arithmetic operation enable. If **A** is 1, then the 8-bit ALU code specifies an arithmetic operation; otherwise, if **A** is 0, a Boolean operation is specified.

❑ **ALU Operation** (bits 58–51)

This field specifies the operation performed by the ALU. The **A**bit determines whether this field specifies a Boolean or arithmetic ALU function. The functionality and assembly syntax for these operations is described in detail in subsection 8.3.2, *Boolean and Arithmetic Functions*.

■ **Boolean (A = 0)**

The ALU function is formed by a sum of Boolean products controlled by the function signals specified directly in the instruction, as shown in Table A–4. This allows any of the 256 possible Boolean combinations of three inputs to be specified.

Table A–4. Base Set ALU Boolean Function Codes

Opcode Bit	ALU Function Signal	Products
58	F7	A & B & C
57	F6	~A & B & C
56	F5	A & ~B & C
55	F4	~A & ~B & C
54	F3	A & B & ~C
53	F2	~A & B & ~C
52	F1	A & ~B & ~C
51	F0	~A & ~B & ~C

Note: A, B, and C are the inputs to the ALU ports.

■ **Arithmetic (A = 1)**

For arithmetic operations, the four odd bits (bits 57, 55, 53, and 51) specify one of the arithmetic operations described in Table A–5. Some of these codes are used for different ALU functions, depending on the operation class they are specified with. Also, some codes allow the function code to be changed according to the sign of the input to the barrel rotator.

The four even bits (bits 58, 56, 54, and 52) specify one of the ALU function modifiers shown in Table A–6. These function modifiers are described in detail in subsection 8.3.4, *ALU Function Modifiers*.

Table A–5. Base Set Arithmetics

Opcode Bits				Derived ALU Function Bits		Carry In	Algebraic Description	Natural Function	Modified Function (If Different From Natural Function)
57	55	53	51	FFFF FFFF 76543210	Hex				
0	0	0	0	10101010	a8	x	Refer to Format E		
0	0	0	1	10101001	a9	1	A–(B|C)	A–B<1<	
0	0	1	0	10100110	a6	0	A+(B&~C)	A+B<0<	
0	0	1	1	10100101	a5	1	A–C	A–C	
0	1	0	0	10011010	9a	1	A–(B|~C)	A–B>1>	Flips to 95 (A–(B&C)) if sign = 0.
0	1	0	1	10011001	99	1	A–B	A–B	
0	1	1	0	10010110	96	C(n)	A–(B&@mf|–B&~@mf)	A+B/A–B	If operation class 0 or 5.
						1/0	A+|B|	A+B/A–B	If class 1–4 or 6–7, flips to 99 if sign = 1.
0	1	1	1	10010101	95	1	A–(B&C)	A–B>0>	
1	0	0	0	01101010	6a	0	A+(B&C)	A+B>0>	
1	0	0	1	01101001	69	~C(n)	A+(B&@mf|–B&~@mf)	A–B/A+B	If class 0 or 5.
						0/1	A–|B|	A–B/A+B	If class 1–4 or 6–7, flips to 66 if sign = 1.
1	0	1	0	01100110	66	0	A+B	A+B	
1	0	1	1	01100101	65	0	A+(B|~C)	A+B>1>	Flips to 6a (A+(B&C)) if sign = 0.
1	1	0	0	01011010	5a	0	A+C	A+C	
1	1	0	1	01011001	59	1	A–(B&~C)	A–B<0<	
1	1	1	0	01010110	56	0	A+(B|C)	A+B<1<	
1	1	1	1	01100000	60	0	(A&C)+(B&C)	field A+B	

Notes: 1) C(n) = the LSB of each part of the C port register if @mf class (operation class 0 or 5)
2) <0< = zero-extended shift left.
3) <1< = one-extended shift left.
4) >0> = zero-extended shift right.
5) >1> = one-extended shift right.
6) >s> = sign-extended shift right.

Table A–6. Class 0 to 7 Function Modifier Opcodes

Function Modifier Code				
58	**56**	**54**	**52**	**Modification Performed**
0	0	0	0	Normal operation.
0	0	0	1	cin
0	0	1	0	%! if maskgen instruction. lmo if not maskgen instruction.
0	0	1	1	%! and cin if maskgen instruction. rmo if not maskgen instruction.
0	1	0	0	A port = 0.
0	1	0	1	A port = 0 and cin.
0	1	1	0	A port = 0 and %! if maskgen instruction. lmbc if not maskgen instruction.
0	1	1	1	A port = 0, %!, and cin if maskgen instruction. rmbc if not maskgen instruction.
1	0	0	0	mc. Multiple; mf bit(s) set by carry-out(s).
1	0	0	1	me. Multiple; mf bit(s) set on the basis of the MSS field of the status register.
1	0	1	0	mrc. Multiple; rotate mf by Asize, mf bit(s) set by carry-out(s).
1	0	1	1	mre. Multiple; rotate mf by Asize, mf bit(s) set on the basis of the MSS field of the status register.
1	1	0	0	mzc. Multiple; clear mf, mf bit(s) set by carry-out(s).
1	1	0	1	mze. Multiple; clear mf, mf bit(s) set on the basis of the MSS field of the status register.
1	1	1	0	mx. Multiple; no setting of bits in the mf register.
1	1	1	1	Reserved.

■ Status register MSS field

The mf status selection (MSS) field in the status register selects which type of ALU result information is used to set MF for the me, mre, and mze function modifiers. The function modifiers enable the setting of bits in the mf register as shown in Table 7–4, *MSS Field Values*.

❑ **dst** (bits 50–48)

This field specifies the three LSBs of the register code for the destination of the ALU result. Parallel transfer formats 6 and 10 let you specify a 4-bit bank code to associate with the three LSBs of the register code specified in dst. This allows the ALU destination to be any register. For all other parallel transfer formats, the D-register bank is implied.

- **src1** (bits 47–45)

  This field specifies the three LSBs of the register code for ALU source operand 1. Parallel transfer formats 6 and 10 let you specify a 4-bit bank code to associate with the three LSBs of the register code in src1. This allows source operand 1 to be any register. For all other parallel transfer formats, the D-register bank code is implied.

  Parallel transfer formats 7, 8, 9, and 10 allow the ALU source 1 to be conditionally selected between odd and even registers in a D register pair. When condition source selection is used, src1 must be an odd D register (d7, d5, d3, or d1).

  Depending on the operation class, src1 is used either as a direct input to the ALU B port, rotated by some amount and then input to the ALU B port, or used as a direct input to the ALU A port.

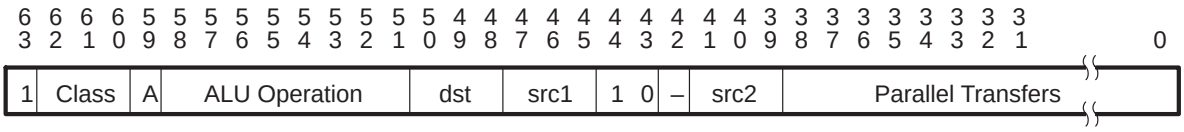
- **imm. src2** (bits 43–39)

  This field specifies an unsigned 5-bit immediate value used as ALU data path source 2. Depending on the operation class, source 2 is used to specify a direct input to the A port of the ALU, a direct input to the C port of the ALU, or the input to the mask generator and/or a barrel rotate amount.

A.3.3 Data Unit Opcode Format C: Base Set ALU With Register src2

Format C allows the exact same base set of ALU operations described for Format B, but ALU data path source 2 is a D register rather than a 5-bit immediate. See Figure A–4.

Figure A–4. Data Unit Opcode Format C: Base Set ALU With Register src2



Fields **Class**, **A**, **ALU Operation**, **dst**, and **src1** are identical to those defined previously for data unit format B (see subsection A.3.2). The assembler always specifies bit 42 as 0 but the bit is not decoded by the machine.

src2 (bits 41–39)

Designates the three LSBs for the D-register code of ALU source operand 2.

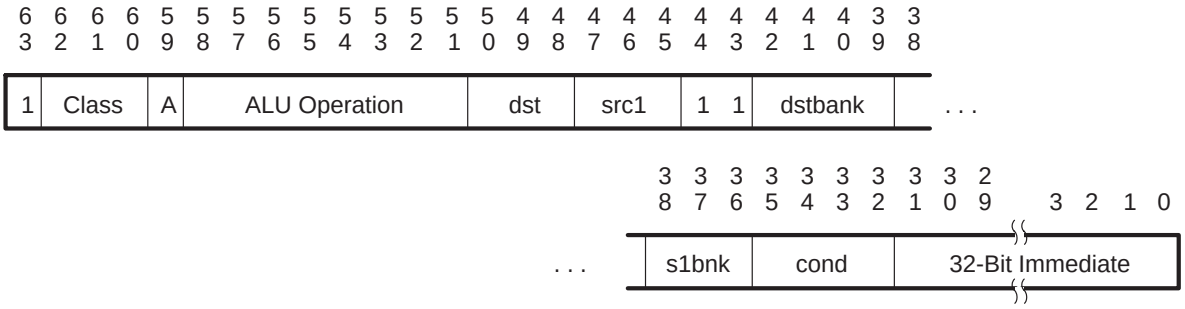
A.3.4 Data Unit Opcode Format D: Base Set ALU With 32-Bit Immediate

This format supports a base set ALU operation with a 32-bit immediate for the ALU data path source 2.

Note that this format requires the entire instruction; therefore, no parallel transfers can be specified. Any register can be specified as the ALU destination, and any lower register can be specified as ALU data path source 1. See Figure A–5.

Figure A–5. Data Unit Format D: Base Set ALU With 32-Bit Immediate

The d0 Register



Fields **Class**, **A**, **ALU Operation**, **dst**, and **src1** are identical to those defined previously for data unit format B (see subsection A.3.2). The following text describes the remaining fields in the d0 register data unit format D.

- **dstbank** (bits 42–39)—ALU destination register bank

  This 4-bit field is used in conjunction with the 3-bit dst field (bits 50–48) to designate any register as the destination for the ALU result.

- **s1bnk** (bits 38–36)

  This 3-bit field comprises the three LSBs of the register bank code for ALU source 1 used in conjunction with the 3-bit src1 field (bits 45-47) to designate the six LSBs of the register code for ALU source 1. The MSB of the register code is an implied 0. This allows any of the lower registers to be specified as the source 1 input to the ALU.

- **cond** (bits 35–32)

  These bits select a condition code. The specified ALU destination register is modified only in the event that the condition specified by this 4-bit code is true. The 4-bit condition codes are defined in Table 1. Note that conditional source selection and status bit protection is not supported by this format.

- **32-bit immediate** (bits 31–0)

  These bits contain the 32-bit immediate value used as ALU data path source 2.

A.3.5 Data Unit Opcode Format E: Miscellaneous Operations

This format is actually a subset of format B, which the ALU function code is 0 and the arithmetic bit is 1. It is used for miscellaneous operations that do not involve the data unit. The miscellaneous operations are described in detail in Section 8.7, *Miscellaneous Operations*. See Figure A–6.

Figure A–6. Data Unit Opcode Format E: Miscellaneous Operations

The d0 Register

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	0
1	0	0	0	1	–	0	–	0	–	0	–	0	–	–	–	–	–	–	0	Operation					Parallel Transfers								

Operation (bits 43–39)

This 5-bit field specifies one the following miscellaneous operations:

Value	Operation
00000	**nop**. No data unit operation is performed. Status is not modified.
00001	Reserved.
00010	**eint**. Globally enable interrupts (individual interrupts must still be enabled in the inten register).
00011	**dint**. Globally disable all interrupts.
00100–11111	Reserved.

A.4 Parallel Transfer Opcode Formats

The following sections describe each field of the 10 different parallel transfer formats. The assembly syntax and functionality of the operations supported by these formats are described in detail in Section 8.8, *Parallel Transfers*.

A.4.1 Parallel Transfer Opcode Format 1: Double Parallel

The double parallel format is shown in Figure A–7. It allows both address units to perform a memory access or address unit arithmetic in parallel with a data unit operation that does not use any non-D register operands.

The index for each address unit operation must be either an index register or a short offset; it cannot be a long offset. PP-relative addressing and conditional operations are not supported by this format.

Figure A–7. Parallel Transfer Opcode Format 1: Double Parallel

38–35	34–31	30	29–28	27	26–24	23–21	20	19–18	17	16–14	13–11	10	9	8–5	4	3–0
Lmode	d	e	size	s	La	Gim/X	L	0bank	L	Gmode	reg	e	size	s	Ga	Lim/X

The following text describes the fields in parallel transfer format 1.

- **Lmode** (bits 38–35)—Local transfer addressing mode

  This field specifies the type of address computation performed by the local address unit. The four bits are encoded as described in Table A–7.

Table A–7. Addressing Mode Codes

Coding				Expression Syntax	Description
0	0	X	X		Nop (nonaddressing mode operation specified).
0	1	0	0	*(an++=xm)	Postaddition of index register, with modify
0	1	0	1	*(an– –=xm)	Postsubtraction of index register, with modify
0	1	1	0	*(an++=imm)	Postaddition of immediate, with modify
0	1	1	1	*(an– –=imm)	Postsubtraction of immediate, with modify
1	0	0	0	*(an+xm)	Preaddition of index register
1	0	0	1	*(an–xm)	Presubtraction of index register
1	0	1	0	*(an+imm)	Preaddition of immediate
1	0	1	1	*(an–imm)	Presubtraction of immediate
1	0	0	0	*(an+=xm)	Preaddition of index register, with modify
1	1	0	1	*(an–=xm)	Presubtraction of index register, with modify
1	1	1	0	*(an+=imm)	Preaddition of immediate, with modify
1	1	1	1	*(an–=imm)	Presubtraction of immediate, with modify

Notes: 1) an = an address register in the appropriate address unit.
2) xm = index register in same address unit as the address register.
3) imm = immediate offset.
4) Index scaling syntax is not shown. See instruction descriptions.

- **d** (bits 34–32)

  This field contains the three LSBs of the register code for the D register (d7–d0) source (for store) or destination (for load or address unit arithmetic) of the local transfer.

- **L,e** (bits 21 and 31, respectively, for local transfer; bits 17 and 9, respectively, for global transfer)

  The L and e bits combine to specify the type of parallel transfer performed.

  - L = 1, e = 0—Zero-extended load
  - L = 1, e = 1—Sign-extended load
  - L = 0, e = 0—Store
  - L = 0, e = 1—Address unit arithmetic

- **size** (bits 30–29 for local transfer; bits 8–7 for global transfer)—Transfer data size

  This field specifies the size of the data being transferred.

  - 00—Byte (8 bits)
  - 01—Halfword (16 bits)
  - 10—Word (32 bits)
  - 11—Reserved

- **s** (bit 28 for local transfer; bit 6 for global transfer)

  The s bit is used as an index scale enable bit if the data size is halfword or word; it is used as an additional offset bit if the data size is byte.

  - **Index Scale**

    For halfword or word data size, this field enables index scaling to be performed before the addressing computation. If s = 1, scale the index (specified in an index register or offset) by shifting left by one bit if data size is halfword, or by two bits if data size is word (specified in the size field of the opcode). If s = 0, do not scale index.

  - **Fourth Offset Bit**

    If the data size is byte and the addressing mode specifies an immediate offset, the s bit becomes an additional offset bit that allows a larger access range for a short offset. The s bit becomes the MSB of a 4-bit offset. The three LSBs of the short offset are specified in the Lim/X field for the local transfer and Gim/X field for the global transfer. If the addressing mode specifies an index register and the data size is byte, the s bit is ignored.

- **La** (bits 27–25)

  This field specifies the three LSBs of the register code for the local address register containing the base address for the local transfer.

- **Gim/X** (bits 24–22)

  This field specifies a short offset or the three LSBs of the register code for a global index register. If the transfer data size is byte, then the scale bit (s) becomes a fourth bit (MSB) for the short offset. The opcode's Gmode field specifies whether this field contains a short offset or an index register code.

- **0bank** (bits 20–18)—Global transfer register bank.

  This 3-bit field is combined with the 3-bit reg field of the opcode to designate any one of the lower registers as the source or destination register for the global transfer. The MSB of the register code is an implied 0.

- **Gmode** (bits 16–13)—Global transfer addressing mode.

  This field specifies how the address is calculated in the global address unit. The four bits are encoded as described in Table A–7.

- **reg** (bits 12–10)

  This field specifies the three LSBs for the global transfer destination (for a load or address unit arithmetic) or source (for a store) register. This 3-bit field is combined with the 0bank field to form the six LSBs of the register code for the global transfer destination or source register. The MSB of the register code is an implied 0; therefore, the register must be one of the lower registers.

- **Ga** (bits 5–3)

  This field specifies the three LSBs of the register code for the global address register containing the global transfer base address.

- **Lim/X** (bits 2–0)

  The Lmode field, which specifies the addressing mode, determines whether Lim/X is a short offset or the three LSBs of the register code for a local index register. If the data size for the transfer is byte and the addressing mode specifies an immediate offset, then the scale bit (s) becomes a fourth bit (MSB) for the short offset.

A.4.2 Parallel Transfer Opcode Format 2: Move||Local

Parallel transfer format 2, shown in Figure A–8, allows a register-to-register move between any two registers to be specified in parallel with a local transfer. The parallel data unit operation cannot use a non-D register operand, and none of the operations can be conditional. The index for the local transfer must be an index register or a short offset; it cannot be a long offset. PP-relative addressing is supported for the local transfer.

Figure A–8. Parallel Transfer Opcode Format 2: Move||Local

38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Lmode			d			e	size			s	La			0	Lrm		dstbank				L	0	0	0	0	src			srcbank				dst				Lim/X		

Fields **Lmode, d, L, e, size, s, La,** and **Lim/X** are identical to those defined previously for parallel transfer format 1 (see subsection A.4.1), except that the L bit (bit 17) applies to the local transfer. The following text describes the remaining fields in parallel transfer format 2.

- **Lrm** (bits 23–22)

  This field specifies whether the local address unit performs PP-relative addressing.

  - 00—Normal absolute addressing
  - 01—Reserved
  - 10—PP-relative addressing using the base address for LRAM0 (**dba**) as the base address
  - 11—PP-relative addressing using the base address for the PP's associated parameter RAM (**pba**) as the base address

- **dstbank** (bits 21–18)

  This field specifies the four MSBs of the register code for the destination of the register-to-register move performed by the global address unit. This allows any register to be specified as the destination for the move.

Note:

If the dstbank and dst fields combine to specify a15 (the read-only zero register) as the destination for the move, this indicates that the 32-bit data in the source register will be sent as an inter-processor command.

- **src** (bits 12–10)

  This field specifies the three LSBs of the register code for the source of the register-to-register move performed by the global address unit. When combined with the srcbank field, this allows any register to be specified as the source for the move.

- **srcbank** (bits 9–6)

  This field specifies the four MSBs of the register code for the source of the register-to-register move performed by the global address unit. This allows any register to be specified as the source for the move.

- **dst** (bits 5–3)

  This field specifies the three LSBs of the register code for the destination of the register-to-register move performed by the global address unit. When combined with the dstbank field, this allows any register to be specified as the destination for the move.

A.4.3 Parallel Transfer Opcode Format 3: Field Move||Local

Parallel transfer format 3, shown in Figure A–9, allows a field move to be specified in parallel with a local transfer operation. The source register must be a D register, while the destination register can be any PP register. Note that PP-relative addressing is not supported using this format.

Figure A–9. Parallel Transfer Opcode Format 3: Field Move||Local

38–35	34–31	30	29–28	27	26–24	23–22	21–18	17	16–15	14–13	12–10	9	8–6	5	4–0
Lmode	d	e	size	s	La	0	itm	dstbank	L	0 0	0 1	src	e	size	D	dst	Lim/X

Fields **Lmode, d, L, e (bit 31), size (bits 30–28), s, La,** and **Lim/X** are identical to those defined previously for parallel transfer format 2 (see subsection A.4.2). The following text describes the remaining fields in parallel transfer format 3.

- **itm** (bits 23–22)

  For a field extract move, this field indicates the location within the source data of the field to be extracted. If the data size is byte, itm indicates the little-endian item number for the desired byte. (For example, 11 indicates the MSbyte, and 00 indicates the LSbyte.) If the data size is halfword, it indicates the little-endian number for the desired halfword. The itm field is not used for field replicate moves.

- **dstbank** (bits 21–18)

  This field specifies the four MSBs of the register code for the destination of the register-to-register move performed by the global address unit. This allows any register to be specified as the destination for the move.

- **src** (bits 12–10)

  This field specifies the three LSBs of the register code for the D register source of the register-to-register move performed by the global address unit.

- **e** (bit 9)

  For field extract moves, this field indicates whether the field of data is sign-extended or zero-extended to 32 bits. This bit is ignored for field replicate moves.

- **size** (bits 8–7)

  Indicates the size of the field from the source register that is either replicated or extracted.

  - 00—Byte (8 bits)
  - 01—Halfword (16 bits)
  - 1x—Reserved

- **D** (bit 6)

  This field indicates whether the field move is a field replicate or field extract move.

  - If D = 1, **field replicate move**

    The least significant byte or halfword in the source data is replicated to fill a 32-bit word.

  - If D = 0, **field extract move**

    The byte or halfword indicated by the itm field is extracted from the source data and either zero- or sign-extended to 32 bits.

- **dst** (bits 5–3)

  This field specifies the three LSBs of the register code for the destination of the field move performed by the global address unit. When combined with the dstbank field, this allows any register to be specified as the destination for the field move.

A.4.4 Parallel Transfer Opcode Format 4: Local (Long Offset)

Parallel transfer format 4, shown in Figure A–10, allows the local transfer to use a long offset. For this format, no global transfer can be specified. Thus, the local transfer is performed over the global port. Note that since no global transfer operation is specified in parallel, the local transfer automatically takes place over the global port with no speed penalty.

Figure A–10. Parallel Transfer Opcode Format 4: Local (Long Offset)

38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Lmode			d			e	size			s	La			1	Lrm	bank					L	0	0	Local Long Offset /X														

Fields **Lmode, d, L, e, size, s, La,** and **Lrm** are identical to those defined previously for parallel transfer format 2 (see subsection A.4.2). The following text describes the remaining fields in parallel transfer format 4.

- **bank** (bits 21–18)—Local transfer register bank.

  This 4-bit field is combined with the 3-bit d field of the opcode to designate any of the PP registers as the source or destination for the local transfer.

- **Local Long Offset/X** (bits 14–0)

  This field specifies either a 15-bit long offset (which is extended to 16 bits for byte accesses by using the s bit) or the three LSBs of register code for a local index register.

A.4.5 Parallel Transfer Opcode Format 5: Global (Long Offset)

Parallel transfer format 5, shown in Figure A–11, allows a global address unit memory access or address unit arithmetic operation to be specified by a long offset.

Figure A–11. Parallel Transfer Opcode Format 5: Global (Long Offset)

38	37	36–22	21–18	17	16–14	13–10	9	8–7	6	5–3	2	1–0
0	0	Global Long Offset/X	bank	L	Gmode	reg	e	size	s	Ga	0	Grm

Fields **Gmode, L, e, size, s,** and **Ga** are identical to those defined previously for parallel transfer format 1 (see subsection A.4.1). The following text describes the remaining fields in parallel transfer format 5.

- **Global Long Offset/X** (bits 36–22)

  This field specifies either a 15-bit long offset (which is extended to 16 bits for byte accesses using the s bit) or the three LSBs of register code for a global index register.

- **bank** (bits 21–18)—Global transfer register bank.

  This 4-bit field is combined with the 3-bit reg field of the opcode to designate any of the PP registers as the source (for store) or destination (for load or address unit arithmetic) for the global transfer.

- **reg** (bits 12–10)

  This field specifies the three LSBs for the global transfer destination or source register. This 3-bit field is combined with the bank field to form the full 7-bit register code. This allows any register to be specified as the source or destination for the global transfer.

- **Grm** (bits 1–0)

  This field specifies whether the global address unit performs PP-relative addressing.

  - 00—Normal absolute addressing
  - 01—Reserved
  - 10—PP-relative addressing using the base address for LRAM0 (**dba**) as the base address
  - 11—PP-relative addressing using the base address for the PP's associated parameter RAM (**pba**) as the base address

A.4.6 Parallel Transfer Opcode Format 6: Non-D DU||Local

Parallel transfer format 6, shown in Figure A–12, uses the global transfer portion of the opcode to specify a non-D register destination and/or source1 for the parallel data unit operation. A local transfer can be specified in parallel with the data unit operation, provided that the index is either an index register or short offset. Note that the local transfer occurs over the global data port.

Figure A–12. Parallel Transfer Opcode Format 6: Non-D DU||Local

38–37	36–35	34–33	32–31	30–29	28–27	26–25	24–23	22–21	20–19	18–17	16–15	14–13	12–11	10–9	8–7	6–5	4–3	2–1	0
Lmode	d	e	size	s	La	0	Lrm	Adstbnk		L	0 0	1	– – – –		As1bank		– – –		Lim/X

Fields **Lmode, d, L, e, size, s, La, Lrm,** and **Lim/X** are identical to those defined previously for parallel transfer format 2 (see subsection A.4.2). The following text describes the remaining fields in parallel transfer format 6.

- **Adstbnk** (bits 21–18)—Data unit destination register bank

  This 4-bit field is combined with the 3-bit dst or dst1 field in the data unit portion of the opcode to form the full 7-bit register code for the ALU destination. This allows any register to be specified as the destination for the ALU result.

- **As1bank** (bits 9–6)—Data unit source1 register bank

  This 4-bit field is combined with the 3-bit src1 field in the data unit portion of the opcode to form the full 7-bit register code for the ALU data path source1. This allows any register to be specified as source1 of the ALU data path.

A.4.7 Parallel Transfer Opcode Format 7: Conditional Move

Parallel transfer format 7 is shown in Figure A–13. This format supports an optionally conditional register-to-register move, in parallel with an optionally conditional data unit operation.

Figure A–13. Parallel Transfer Opcode Format 7: Conditional Move

38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	–	cond				c	r	g	N	C	V	Z	0	–	–	dstbank				–	0	0	0	0	src				srcbank			dst				–	–	–

Fields **dstbank, src, srcbank,** and **dst** are identical to those defined previously for parallel transfer format 2 (see subsection A.4.2). The following text describes the remaining fields in parallel transfer format 7.

- **cond** (bits 35–32)

  This field specifies a condition code that can be associated with the data unit operation and/or the transfer performed by the global address unit. The 16 condition codes are defined in Table A–8. The condition code applies to the data unit operation or global unit transfer according to the **c, r,** and **g** bits.

Table A–8. Condition Codes

Coding	Mnemonic	Description	Status Bit Combination
0 0 0 0	u	Unconditional (default)	None
0 0 0 1	p	Positive	~N&~Z
0 0 1 0	ls	Lower than or same	~C|Z
0 0 1 1	hi	Higher than	C&~Z
0 1 0 0	lt	Less than	(N&~V)|(~N&V)
0 1 0 1	le	Less than or equal	(N&~V)|(~N&V)|Z
0 1 1 0	ge	Greater than or equal	(N&V)|(~N&~V)
0 1 1 1	gt	Greater than	(N&V&~Z)|(~N&~V&~Z)
1 0 0 0	hs,c	Higher than or same, carry	C
1 0 0 1	lo,nc	Lower than, no carry	~C
1 0 1 0	eq,z	Equal, zero	Z
1 0 1 1	ne,nz	Not equal, not zero	~Z
1 1 0 0	v	Overflow	V
1 1 0 1	nv	No overflow	~V
1 1 1 0	n	Negative	N
1 1 1 1	nn	Nonnegative	~N

- **c** (bit 31)—Conditional source enable
  - If C = 0, source 1 of the ALU data path is specified unconditionally by the src1 field (bits 47–45) in the data unit portion of the opcode. Source 1 can be any D register.
  - If C = 1, either an odd or even register in a D register pair is selected as source 1 of the ALU data path, according to the negative condition. The D register pair is identified by the odd register number in the src1 field (bits 47–45). If the condition is true, the odd register in the D register pair is used. If the condition is false, the even register in the D register pair is used.
- **r** (bit 30)—Conditional result enable
  - If r = 0, the data unit results are written unconditionally to the destination register.
  - If r = 1, the data unit results are written to the destination register only if the designated condition is true. If the condition is false, data unit operations are discarded.
- **g** (bit 29)—Conditional global transfer move enable

  This bit specifies whether the register-to-register move occurs conditionally.
  - If g = 0, the move occurs unconditionally.
  - If g = 1, the move to the destination register occurs only if the specified condition is true.
- **N** (bit 28)

  If set to 1, the N bit indicates that the negative status bit should be protected from modification by this instruction.
- **C** (bit 27)

  If set to 1, the C bit indicates that the carry status bit should be protected from modification by this instruction.
- **V** (bit 26)

  If set to 1, the V bit indicates that the overflow status bit should be protected from modification by this instruction.
- **Z** (bit 25)

  If set to 1, the Z bit indicates that the zero status bit should be protected from modification by this instruction.

A.4.8 Parallel Transfer Opcode Format 8: Conditional Field Move

Parallel transfer format 8, shown in Figure A–14, allows an optionally conditional field move to be specified in parallel with an optionally conditional data unit operation. The source of a field move must be a D register.

Figure A–14. Parallel Transfer Opcode Format 8: Conditional Field Move

38	37	36	35–34	33–32	31	30	29	28	27	26	25	24	23–22	21–20	19	18	17	16	15	14–13	12–11	10	9	8–7	6	5–4	3–2	1–0
0	0	–	cond		c	r	g	N	C	V	Z	0	itm	dstbank		–	0	0	0	1	src		e	size	D	dst	–	– –

Fields **cond, c, r, g, N, C, V, Z, dstbank, src,** and **dst** are identical to those defined previously for parallel transfer format 7 (see subsection A.4.7). The following text describes the remaining fields in parallel transfer format 8.

- **itm** (bits 23–22)

  For a field extract move, this field indicates the location within the source data of the field to be extracted. For byte data size, itm indicates the little-endian item number for the desired byte. For a halfword data size, it indicates the little-endian number for the desired halfword. The itm field is not used for field replicate moves.

- **e** (bit 9)

  For field extract moves, this field indicates whether the field of data is sign extended or zero extended to 32 bits. This bit is ignored for field replicate moves.

- **size** (bits 8–7)

  This field indicates the size of the field from the source register that is either replicated or extracted and extended.

- **D** (bit 6)

  This field indicates whether the field move is a field replicate or field extract move.

  - If D = 1, **field replicate move**

    The least significant byte or halfword in the source data is replicated to fill a 32-bit word.

  - If D = 0, **field extract move**

    The byte or halfword indicated by the itm field is extracted from the source data and either zero- or sign-extended to 32 bits.

A.4.9 Parallel Transfer Opcode Format 9: Conditional DU||Conditional Global

Parallel transfer format 9, shown in Figure A–15, allows an optionally conditional global transfer to be specified in parallel with an optionally conditional data unit operation.

Figure A–15. Parallel Transfer Opcode Format 9: Conditional DU||Conditional Global

38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	–	cond				c	r	g	N	C	V	Z	Gim/X			bank			L	Gmode			reg				e	size		s	Ga				1	Grm	

Fields **cond, c, r, N, C, V,** and **Z** are identical to those defined previously for parallel transfer format 7 (see subsection A.4.7). Fields **Gmode, bank, reg, L, e, size, s, Ga, Grm,** and **Gim/X** are identical to those defined previously for parallel transfer format 5 (see subsection A.4.5). The following text describes the remaining fields in parallel transfer format 9.

- **g** (bit 29)—Conditional global transfer enable.

  This bit specifies whether the global transfer is conditional.

  - If g = 1, the global transfer is conditional.

    For **loads** or **address unit arithmetic**, the write to the destination register occurs only if the specified condition is true. If the condition is false, no write to the destination register occurs. Note that any update of the address register and for loads, the memory access occurs, regardless of the whether the condition is true or false.

    For **stores**, the odd or even register in a D register pair is conditionally selected as the source register. The condition code must be the negative condition and the src field must specify an odd D register. If the negative condition is true, the odd register in the D register pair is used as the source. If the negative condition is false, the even register is selected. bank must specify a D register for conditional stores.

  - If g = 0, the global transfer is not conditional.

- **Gim/X** (bits 24–22)

  This field specifies the three LSBs of the register code for a local index register or a 3-bit (short) offset according to the addressing mode (Gmode). Note that if the data size for the transfer is byte and the addressing mode specifies an offset, then the s bit (bit 28) becomes a fourth bit (MSB) for the short offset.

A.4.10 Parallel Transfer Opcode Format 10: Conditional Non-D DU

Parallel Transfer format 10, shown in Figure A–16, lets you specify a conditional data unit operation with a non-D register destination and/or a non-D register source1. Individual status bits can be protected from modification. No parallel transfers can be specified with this format.

The write to the destinations of the data unit operations can be performed conditionally. If conditional write to destination is specified, the write to the data unit destinations occur only if the specified condition is true. If the condition is false, the data unit results are discarded.

The odd or even register in a D register pair can optionally be selected as ALU data path source 1 according to the negative condition. If conditional source selection is specified, src1 specified in the data unit portion of the opcode must specify an odd D register (d7, d5, d3, or d1).

Figure A–16. Parallel Transfer Format 10: Conditional Non-D DU

38	37	36	35 34 33 32	31	30	29	28	27	26	25	24	23	22	21 20 19 18	17	16	15	14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
0	0	–	cond	c	r	–	N	C	V	Z	0	–	–	Adstbank	–	0	0	1	– – – –	As1bank	– – – – – –

Fields **cond, c, r, N, C, V,** and **Z** are identical to those defined previously for parallel transfer format 7 (see subsection A.4.7). The following text describes the remaining fields in parallel transfer format 10.

- **Adstbnk** (bits 21–18)—Data unit destination register bank.

  This 4-bit field is combined with the 3-bit dst or dst1 field in the data unit portion of the opcode to form the full 7-bit register code for the ALU destination. This allows any register to be specified as the destination for the ALU result.

- **As1bank** (bits 9–6)—Data unit source1 register bank.

  This 4-bit field is combined with the 3-bit src1 field in the data unit portion of the opcode to form the full 7-bit register code for the ALU data path source1. This allows any register to be specified as source1 for the ALU data path.

Appendix B

packetpp.i Include File

This appendix contains the packetpp.i include file. packetpp.i has two big-endian structure definitions for accessing packet transfer parameters.

- The first structure definition assumes that the structure base is at the first element in the structure. In this form of the structure, several of the elements in the packet transfer parameters cannot be accessed with short offsets.
- The second structure definition uses the .access directive to locate the structure base in the middle of the packet transfer parameters. This allows all of the structure elements to be accessible via short offsets.

```
*-----------------------------------------------------------------------------
*          Copyright (C) 1992–1994  Texas Instruments Incorporated
*                           All Rights Reserved
*-----------------------------------------------------------------------------
*
*  packetpp.i   --  PP assembly language include file
*
*               --  defines data structures, memory addresses,
*                   and shift amounts for using packet transfers
*
*               --  only BIG ENDIAN supported in release 1.0
*
*-----------------------------------------------------------------------------
*  History:
*    04/13/94 ... Initial 1.0 Release ......................... C. Read
*-----------------------------------------------------------------------------



*   Linked List Start Address
ePT_LinkedListStart:     .set     0xfc

*   Bit shift amounts for bit testing/setting in the comm register
PT_SubmitShift:          .set     28
PT_QueuedShift:          .set     29
PT_SuspendShift:         .set     30
PT_ForegroundShift:      .set     31


*--- General PT Definitions – Access point is at lowest address in the PT ----
                         .struct
                         .access                 ; *************Access Point
sPT_Next:                    .word               ; 0     All PTs
sPT_Options:                 .word               ; 1     All PTs
                             .union
sPT_SrcStartAddress:             .word           ; 2     Dimensioned Src PTs
sPT_SrcBaseAddress:              .word           ; 2     Guided Src PTs
                             .endunion
                             .union
sPT_DstStartAddress:             .word           ; 3     Dimensioned Dst PTs
sPT_DstBaseAddress:              .word           ; 3     Guided Dst PTs
                             .endunion
                             .union
sPT_SrcBACount:                  .word           ; 4     Dimensioned Src PTs
                                 .struct
ePT_SrcBCount:                       .uhalf      ; 8     Dim/Fixed Patch Src PTs
ePT_SrcACount:                       .uhalf      ; 9     Dim/Fixed Patch Src PTs
                                 .endstruct
                             .endunion
                             .union
sPT_DstBACount:                  .word           ; 5     Dimensioned Dst PTs
                                 .struct
ePT_DstBCount:                       .uhalf      ; 10    Dim/Fixed Patch Dst PTs
ePT_DstACount:                       .uhalf      ; 11    Dim/Fixed Patch Dst PTs
                                 .endstruct
                             .endunion
```

```
                                .union
sPT_SrcCCount:                      .word           ; 6     Dimensioned Src PTs
sPT_SrcNumEntries:                  .word           ; 6     Guided Src PTs
                                .endunion
                                .union
sPT_DstCCount:                      .word           ; 7     Dimensioned Dst PTs
sPT_DstNumEntries:                  .word           ; 7     Guided Dst PTs
                                .endunion
                                .union
ePT_SrcBPitch:                      .word           ; 8     Dim/Fixed Patch Src PTs
ePT_LSFillWord:                     .word           ; 8     Fill with Value PTs
                                .endunion
ePT_DstBPitch:                  .word               ; 9     Dimd/Fixed Patch Dst PTs
                                .union
ePT_SrcCPitch:                      .word           ; 10    Dimensioned Src PTs
ePT_MSFillWord:                     .word           ; 10    Fill with Value PTs
ePT_SrcGuideTablePtr:               .word           ; 10    Guided Src PTs
                                .endunion
                                .union
ePT_DstCPitch:                      .word           ; 11    Dimensioned Dst PTs
ePT_DstGuideTablePtr:               .word           ; 11    Guided Dst PTs
                                .endunion
                                .union
ePT_TransWord0:                     .word           ; 12    Transparency PTs
ePT_ColorRegWord0:                  .word           ; 12    Block Write PTs
                                .endunion
                                .union
ePT_TransWord1:                     .word           ; 13    Transparency PTs
ePT_ColorRegWord1:                  .word           ; 13    Block Write PTs
                                .endunion
ePT_Reserved1:                      .word           ; 14    Makes PT_Size accurate
ePT_Reserved2:                      .word           ; 15    Makes PT_Size accurate

ePT_Size:                   .endstruct              ; 16    words/64 bytes = PT size


*----------------- Alternate Definition for short (3 bit) offsets -------------

*   Address Register points at Dst C Count (pPT + 7<<2)
*   This means all values can be reached with a 3 bit immediate offset
sPTC_Offset:                .set     28          ;   Byte Offset
ePTC_Offset:                .set     28          ;   Byte Offset


*  Central Access PT Definitions
                            .struct
sPTC_Next:                      .word               ; -7    All PTs
sPTC_Options:                   .word               ; -6    All PTs
                                .union
sPTC_SrcStartAddress:               .word           ; -5    Dimensioned Src PTs
sPTC_SrcBaseAddress:                .word           ; -5    Guided Src PTs
                                .endunion
                                .union
sPTC_DstStartAddress:               .word           ; -4    Dimensioned Dst PTs
sPTC_DstBaseAddress:                .word           ; -4    Guided Dst PTs
                                .endunion
```

```
                                .union
sPTC_SrcBACount:                    .word           ; -3    Dimensioned Src PTs
                                    .struct
sPTC_SrcBCount:                         .uhalf      ; -6    Dim/Fixed Patch Src PTs
sPTC_SrcACount:                         .uhalf      ; -5    Dim/Fixed Patch Src PTs
                                    .endstruct
                                .endunion
                                .union
sPTC_DstBACount:                    .word           ; -2    Dimensioned Dst PTs
                                    .struct
sPTC_DstBCount:                         .uhalf      ; -4    Dim/Fixed Patch Dst PTs
sPTC_DstACount:                         .uhalf      ; -3    Dim/Fixed Patch Dst PTs
                                    .endstruct
                                .endunion
                                .union
sPTC_SrcCCount:                     .word           ; -1    Dimensioned Src PTs
sPTC_SrcNumEntries:                 .word           ; -1    Guided Src PTs
                                .endunion
                            .access                 ; *************Access Point
                                .union
sPTC_DstCCount:                     .word           ; 0     Dimensioned Dst PTs
sPTC_DstNumEntries:                 .word           ; 0     Guided Dst PTs
                                .endunion
                                .union
sPTC_SrcBPitch:                     .word           ; 1     Dim/Fixed Patch Src PTs
sPTC_LSFillWord:                    .word           ; 1     Fill with Value PTs
                                .endunion
sPTC_DstBPitch:                 .word               ; 2     Dim/Fixed Patch Dst PTs
                                .union
sPTC_SrcCPitch:                     .word           ; 3     Dimensioned Src PTs
sPTC_MSFillWord:                    .word           ; 3     Fill with Value PTs
sPTC_SrcGuideTablePtr:              .word           ; 3     Guided Src PTs
                                .endunion
                                .union
sPTC_DstCPitch:                     .word           ; 4     Dimensioned Dst PTs
sPTC_DstGuideTablePtr:              .word           ; 4     Guided Dst PTs
                                .endunion
                                .union
sPTC_TransWord0:                    .word           ; 5     Transparency PTs
sPTC_ColorRegWord0:                 .word           ; 5     Block Write PTs
                                .endunion
                                .union
sPTC_TransWord1:                    .word           ; 6     Transparency PTs
sPTC_ColorRegWord1:                 .word           ; 6     Block Write PTs
                                .endunion
sPTC_Reserved1:                 .word               ; 7
ePTC_Reserved2:                 .word               ; 8
                            .endstruct
```

```
*
*  ----------------- Definitions for guide table entries -----------------
*

*
*--- Variable Patch Guides ---
*

                .struct
                    .union
sVPG_BACount:           .word           ; 0     Variable Patch Guide Dimensions
                        .struct
sVPG_BCount:                .uhalf      ; 0     Variable Patch Guide Dimensions
sVPG_ACount:                .uhalf      ; 2     Variable Patch Guide Dimensions
                        .endstruct
                    .endunion
                    .union
sVPG_Offset:            .word           ; 4     Offset from base for patch
sVPG_Delta:             .word           ; 4     Delta from last patch
                    .endunion
sVPG_Size:      .endstruct              ; 8

*
*--- Fixed Patch Guides ---
*

                .struct
                    .union
sFPG_Offset:            .word           ; 0     Offset from base for patch
sFPG_Delta:             .word           ; 0     Delta from last patch
                    .endunion
sFPG_Size:      .endstruct              ; 4
```

Appendix C

Understanding the Binary Number System

To take full advantage of the PP's architecture, you must have a good working knowledge of the binary number system, particularly when performing multiple (split-ALU) arithmetic. Knowing how and why numbers carry out or overflow can help you maintain precision. This appendix serves as a refresher about how binary numbers behave under certain conditions.

Topics

C.1	Carry-Outs From Addition and Subtraction	PP:C-2
C.2	Signed and Unsigned Numbers	PP:C-4
C.3	Unsigned Number Overflow	PP:C-5
C.4	Signed Number Overflow	PP:C-7
C.5	Extended-Precision Support for Signed Operations With Multiple Arithmetic	PP:C-8

C.1 Carry-Outs From Addition and Subtraction

Arithmetic and logic units (ALUs) perform 2s-complement arithmetic during additions and subtractions. The operations can be treated as either signed or unsigned.

The carry-out for an addition is straightforward. If the result of the addition does not fit in the number of bits allocated for the result, then a carry-out occurs.

For extended-precision additions (add with carry), the carry-out from the addition of the lower order quantities can be injected as the carry-in to the addition of the higher-order quantities.

ALUs perform subtraction by using 2s-complement arithmetic. A subtract (minuend – subtrahend) is performed by adding the 2s complement of the subtrahend to the minuend. Typically, this is implemented by adding the 1s complement (logical inversion) of the subtrahend to the minuend and injecting a carry into the LSB of the addition, since $-x = \overline{x} + 1$.

If the minuend is greater than or equal to the subtrahend, then a carry-out occurs from the MSB of the ALU. Thus, a carry-out of 1 for a subtraction indicates that a borrow did not occur (borrow = 0; carry = $\overline{\text{borrow}}$ = 1).

If the minuend is less than the subtrahend, then a carry-out does not occur, indicating that there was a borrow ($\overline{\text{borrow}}$ = 0). For extended-precision subtractions (subtract with borrow), the carry-out ($\overline{\text{borrow}}$) is injected as the carry-in to the next higher-order subtraction.

If there is no carry-out from the least significant subtraction, then the carry-in to the higher-order subtraction is 0. This has the effect of subtracting 1 (borrowing) from the result of the higher-order subtraction.

Note:

When setting the carry bit in the status register or writing the multiple arithmetic carry-outs to mf, the PP always stores the natural carry or carries generated by the ALU.

Most other microprocessors set carry status to the carry bit for addition but set it to not-carry (or borrow) when performing subtraction. When performing subtract with borrow, these other processors re-invert the not-carry in order to get the proper carry-in to the ALU.

The PP's handling of carries/not borrows makes them easy to use but results in a slightly different set of conditional branch equations than other processors to get the same branch conditions.

C.2 Signed and Unsigned Numbers

Depending on the application, you may want to treat binary numbers as being either signed or unsigned. The ALU performs the same binary operation for signed or unsigned arithmetic; the difference is in how the result is treated and what constitutes an overflow.

C.3 Unsigned Number Overflow

Often, in image and graphics processing, pixel values are expressed as unsigned binary numbers. Unsigned numbers use all the available bits to express the magnitude of the number.

For example, the 8-bit unsigned binary number 11111111b is greater than 00000001b.

C.3.1 Unsigned Addition Overflow

When two unsigned numbers are added, the result overflows if it is too big to fit into the number of bits allocated for the result. Performing the 8-bit addition of the 8-bit binary numbers 11111111b and 00000001b results in an answer of 00000000b with a carry-out of 1.

The carry-out contains the ninth bit of precision that overflowed the 8-bit range. Thus, the carry-out can be used to detect overflows and, if saved, to maintain the precision of unsigned numbers.

C.3.2 Unsigned Subtraction Overflow

A negative number cannot be expressed by an unsigned number. Thus, an underflow occurs if the result of subtracting two unsigned numbers is negative (a borrow).

As for addition overflows, you can detect unsigned subtraction underflows by looking at the carry-out of the operation. On the PP, carry-out for subtraction means not-borrow ($\overline{\text{borrow}}$); therefore, an underflow from an unsigned subtraction is indicated by not having a carry.

Example C–1 to Example C–3 demonstrate the carry-out behavior for positive, negative, and zero results when 8-bit numbers are subtracted. In each case, the subtrahend is 1s complemented and then added to the minuend plus 1. These examples demonstrate that a carry-out of 1 indicates a nonnegative (and thus legal) unsigned result.

Note that the result from Example C–2 can still be useful, even though an underflow has occurred. The magnitude of the difference is still correct, but the sign of the difference is in the carry bit.

Example C–1. Unsigned Subtraction With a Positive Result (Carry-Out = 1)

```
  00000111
+ 11111010     (1s complement of 00000101)
+        1     (injected carry-in)
  00000010
```

Example C–2. Unsigned Subtraction With a Negative Result (Carry-Out = 0)

```
  00000101
+ 11111000     (1s complement of 00000111)
+        1     (injected carry-in)
  11111110
```

Example C–3. Unsigned Subtraction With a Result of Zero (Carry-Out = 1)

```
  00000000
+ 11111111     (1s complement of 00000000)
+        1     (injected carry-in)
  00000000     with a carry-out = 1
```

C.4 Signed Number Overflow

Signed (2s complement) binary numbers embed the sign in the number. This allows numbers in the range from $(2^{n-1})-1$ to $-(2^{n-1})$ to be expressed (where n is the number of bits in the number). Overflows on signed arithmetic are a little more difficult to detect than for unsigned arithmetic.

An overflow on a signed add occurs if both operands are positive and the sign bit of the result is a 1 (negative), indicating that the result has rolled over from positive to negative. Overflow on an add also occurs if both operands are negative and the result's sign bit is 0 (positive).

Thus, overflow on addition occurs if both of the sign bits of the operands are the same and the result has a different sign bit. Similarly, for subtraction (minuend – subtrahend), an underflow occurs if the minuend and the 2s complement of the subtrahend have the same sign and the result has a different sign bit.

C.5 Extended-Precision Support for Signed Operations With Multiple Arithmetic

The extended-precision option for multiple arithmetic specified by the function modifier code and the MSS field of the status register supports extending the precision on signed arithmetic operations to assist in handling overflow. In effect, an extra MSB of precision is generated for each split of the ALU and saved in the mf register.

Adding/subtracting two *n*-bit (for the example below, $n = 16$) signed numbers may overflow to $n + 1$ bits. The overflow may be "temporary" in the sense that the add or subtract operation could be followed by a 1-bit shift-right, adjusting the fixed point. The extra bit(s) of precision that is saved in the mf register can applied as part of a three-input ALU operation with rotation/shifting. The bits saved in the mf can also be accumulated to produce an extended precision result.

C.5.1 Examples of the Math Involved

As an example, a 16-bit FFT butterfly is an algorithm that at each stage must generate a series of 16-bit results. Thus, it always scales the result of the add back to 16 bits of precision.

To understand better how the extended-precision option works, the examples below show what happens when 16-bit numbers are sign-extended to 32 bits before adding/subtracting, then compare with using the extended-precision option. A 16-bit number sign-extended to 32 bit ends up with 17 sign bits (the MSB of the 16-bit number plus 16 replicated sign bits). Thus, the 17 MSBs in a negative number are all 1.

Example C–4 through Example C–9 show some of the most interesting cases for signed addition. They show the propagate and carry-out bits from bit 15 (the 16th bit of precision). These bits can generate the 17th bit of precision, as discussed in subsection C.5.2.

Example C–4. Adding a Positive to a Positive With No Signed Overflow in the 16 LSBs

propagate (15) = 0, carry-out(15) = 0

```
  0000 0000 0000 0000  0011 1111 1111 1111
+ 0000 0000 0000 0000  0011 0000 0000 0000
-----------------------------------------
  0000 0000 0000 0000  0110 1111 1111 1111
shift right one and truncate
                       0011 1111 1111 1111
```

Example C–5. Adding Two Large Positive Numbers Returning Signed Overflow in the 16 LSBs

propagate (15) = 0, carry-out(15) = 0

```
  0000 0000 0000 0000  0111 1111 1111 1111
+ 0000 0000 0000 0000  0011 0000 0000 0000
-----------------------------------------
  0000 0000 0000 0000  1010 1111 1111 1111
shift right one and truncate
                       0101 1111 1111 1111
```

Example C–6. Adding a Negative to a Negative With No Signed Overflow in the 16 LSBs

propagate (15) = 0, carry-out(15) = 1

```
  1111 1111 1111 1111  1111 1111 1111 1111
+ 1111 1111 1111 1111  1011 0000 0000 0000
-----------------------------------------
  1111 1111 1111 1111  1010 1111 1111 1111
shift right one and truncate
                       1101 0111 1111 1111
```

Example C–7. Adding Two Very Negative Numbers Returning Signed Overflow in the 16 LSBs

propagate (15) = 0, carry-out(15) = 1

```
  1111 1111 1111 1111  1110 1111 1111 1111
+ 1111 1111 1111 1111  1001 0000 0000 0000
-----------------------------------------
  1111 1111 1111 1111  0111 1111 1111 1111
shift right one and truncate
                       1011 0111 1111 1111
```

The next two examples show a positive number and a negative number being added together. In Example C–8, the result is negative; in Example C–9, the result is positive. Recall that in the case of signed arithmetic with different signs, there can never be an overflow (as such).

Example C–8. Adding a (More) Negative to a Positive With No Signed Overflow in the 16 LSBs

propagate (15) = 1, carry-out(15) = 0

```
  0000 0000 0000 0000  0011 0000 0000 0000
+ 1111 1111 1111 1111  1100 0000 0000 0000
-----------------------------------------
  1111 1111 1111 1111  1111 0000 0000 0000
shift right one and truncate
                       1111 1000 0000 0000
```

Example C–9. Adding a (Less) Negative to a Positive With No Signed Overflow in the 16 LSBs

propagate (15) = 1, carry-out(15) = 1

```
  0000 0000 0000 0000  0011 0000 0000 0000
+ 1111 1111 1111 1111  1110 0000 0000 0000
-----------------------------------------
  0000 0000 0000 0000  0001 0000 0000 0000
shift right one and truncate
                       0000 1000 0000 0000
```

C.5.2 Generating the 17th Bit of Precision

Because the numbers above are sign-extended 16-bit numbers and the 17 MSBs for each input are sign bits, the propagate term for bit 15 and bits 16 to 31 are **always** the same. The only difference between bit 15 and bit 16 of the result is their respective carry-in bits. The carry-in to bit 16 is the carry-out of bit 15. Since the ALU simply XORs propagate with carry-in to get the final result, bit 16 of a sign-extended 16-bit result can be generated by taking the propagate for bit 15 and XORing it with the carry-out of bit 15.

This way of predicting bit 16 (the 17th bit of precision) is analogous for signed arithmetic to unsigned arithmetic's carry out (its 17th bit of precision).

By using the propagate term in the equation above the method of predicting the 17th bit of signed precision works for subtraction as well.

The bit(s) of extra precision are saved in the mf register. Special capabilities of the three-input ALU injects these bits into the MSB of a shift-right of a 16-bit number. The Boolean operation below (or some equivalent) can be used to force the corresponding saved 17th bit in the mf register over the rotating-in "garbage" bit(s) (the LSB(s) that are to be discarded in the shift-right):

(~A&B)|(A&C)

Where:

- C is the expanded mf register
- B is the number(s) to be modified rotated right by 1 bit
- A is a mask bit with a 1 in the MSB position(s) and zero everywhere else (for example, 0x8000 8000 for a dual 16-bit case).

Appendix D

Quick Reference

Once you have familiarized yourself with the MVP parallel processor and its instruction set, you can use this appendix as a reference guide.

Topics

D.1	The Opcode Formats and Parallel Transfer Formats	PP: D-2
D.2	The Data Unit Operations	PP: D-5
D.3	The EALU Operations	PP: D-14
D.4	The Parallel Transfers	PP: D-18
D.5	The Interprocessor Commands	PP: D-23
D.6	The Condition Codes	PP: D-24
D.7	The Registers	PP: D-25
D.8	The Memory Maps	PP: D-31

D.1 The Opcode Formats and Parallel Transfer Formats

Figure D–1 shows you the data unit formats and the parallel transfer formats for the opcode. For a complete description of the individual opcode fields, refer to Appendix A, *The PP Opcodes.*

Table D–1 shows you the supported parallel operations, their format, and whether they are global or local transfers. For more information on parallel operations, refer to Section 8.10, *Parallel-Operation Combinations*.

Figure D–1. Opcode Formats

(a) Data Unit Formats

63–32	31–29	28–26	25–23	22–20	19–17	16–14	13–11	10–8	7–5	4–3	2–1	0	
0 1 1	oper	src3	dst2	dst1	src1	src4	src2	Parallel Transfers					A. Six-Operand (MPY||ADD, etc.)
1	class	A	ALU Operation	dst	src1	0	imm. src2	Parallel Transfers					B. Base Set ALU (5-Bit Immediate)
1	class	A	ALU Operation	dst	src1	1 0	– src2	Parallel Transfers					C. Base Set ALU (Register src2)
1	class	A	ALU Operation	dst	src1	1 1	dstbank	s1bnk	cond	32-Bit Immediate			D. Base Set ALU (32-Bit Immediate)
1	0 0 0	1	– 0 – 0 – 0 – 0	– – – – – –		0	Operation	Parallel Transfers					E. Miscellaneous
0 0	Reserved												
0 1 0	Reserved												

(b) Transfers Formats (for Parallel Transfers)

38–37	36–35	34–33	32–31	30–29	28–27	26–25	24–23	22–21	20–19	18–17	16–15	14–13	12–11	10–9	8–7	6–5	4–3	2–1	0	
Lmode	d	e	size	s	La	Gim/X		L	0bank	L	Gmode		reg	e	size	s	Ga	Lim/X		1. Double Parallel
Lmode	d	e	size	s	La	0	Lrm	dstbank		L	0 0	0 0	src	srcbank			dst	Lim/X		2. Move||Local
Lmode	d	e	size	s	La	0	itm	dstbank		L	0 0	0 1	src	e	size	D	dst	Lim/X		3. Field Move||Local
Lmode	reg	e	size	s	La	1	Lrm	bank		L	0 0	Local Long Offset /X								4. Local (Long Offset)
0 0	Global Long Offset/X							bank		L	Gmode		reg	e	size	s	Ga	0	Grm	5. Global (Long Offset)
Lmode	d	e	size	s	La	0	Lrm	Adstbnk		L	0 0	1	– – – –	As1bank			– – –	Lim/X		6. Non-D DU||Local
0 0	–	cond		c	r	g	N C V Z	0	– –	dstbank	–	0 0	0 0	src	srcbank		dst	– – –		7. Conditional DU||Conditional Mode
0 0	–	cond		c	r	g	N C V Z	0	itm	dstbank	–	0 0	0 1	src	e	size	D	dst	– – –	8. Conditional DU||Conditional Field Move
0 0	–	cond		c	r	g	N C V Z	Gim/X		bank	L	Gmode		reg	e	size	s	Ga	1 Grm	9. Conditional DU||Conditional Global
0 0	–	cond		c	r	–	N C V Z	0	– –	Adstbank	–	0 0	1	– – – –	As1bank			– – – – – –		10. Conditional Non-D DU

Table D–1. Summary of Parallel Transfer Formats

Format	ALU Operands		Cond	Status Protection	Global Transfer				Local Transfer			
					move	Load/Store/AUA			Load/Store/AUA			
	dst1	src1			src → dst	S/D	Index	Rel	S/D	Index	Rel	Port
Double parallel	D	D	No	No	—	Lower	X/short	No	D	X/short	No	Local
Move||local	D	D	No	No	Any → Any	—	—	—	D	X/short	Yes	Local
Field move||local	D	D	No	No	D → Any	—	—	—	D	X/short	No	Local
Global (long offset)	D	D	No	No	—	Any	X/long	Yes	—	—	—	—
Local (long offset)	D	D	No	No	—	—	—	—	Any	X/long	Yes	Global
Non-D DU||local	Any	Any	No	No	—	—	—	—	D	X/short	Yes	Global
Conditional move	D	D	Yes	Yes	Any → Any	—	—	—	—	—	—	—
Conditional field move	D	D	Yes	Yes	D → Any		—	—	—	—	—	—
Conditional global	D	D	Yes	Yes	—	Any	X/short	Yes	—	—	—	—
Conditional non-D DU	Any	Any	Yes	Yes	—		—	—	—	—	—	—
32-bit immediate base ALU	Any	Lower	Yes	No	—							

Note: DU = Data unit
AUA = Address unit arithmetic
— = Not applicable

D.2 The Data Unit Operations

This section summarizes the data unit operators, syntaxes, classes, function combinations, and modifiers.

D.2.1 Data Unit Operators and Operation Syntax

The data unit operators are shown in Table D–2. The operators are listed in order of precedence. The operators not separated by a horizontal line are of equal precedence and are evaluated left to right.

Table D–2. PP Operator Precedence

Operator	Function	See Page
src1[n]src1–1	Selection of either the odd (n is true) or even (n is false) register in a D register pair based on the negative condition code	PP: 8-17
( )	Subexpression delimiters	PP: 8-21
@mf	Expander operator (for expanding each bit to a byte, halfword or word)	PP: 8-22
%	Mask generator	PP: 8-24
%%	Nonmultiple mask generator (EALUs only)	PP: 8-24
%!	Modified mask generator, 0xFFFF FFFF output for 0 input. Same as for nonzero input.	PP: 8-24
%%!	Nonmultiple shift right mask generator (EALUs only)	PP: 8-24
\\	Rotate left	PP: 8-26
<<	Shift left (assembler pseudo-operator for rotate and mask)	PP: 8-27
>>u	Unsigned shift right	PP: 8-27
>> or >>s	Signed shift right	PP: 8-29
&	Bitwise AND	PP: 8-20
^	Bitwise XOR	PP: 8-20
|	Bitwise OR	PP: 8-20
+	Addition	PP: 8-18
–	Subtraction	PP: 8-18
=[cond]	Conditional assignment	PP: 8-15
=[cond.pro]	Conditional assignment with status protection	PP: 8-16
=	Equate	PP: 8-15

Table D–3 shows a summary of the supported data unit operations and their syntax. The square brackets indicate optional fields. The ⟦ symbol indicates required square brackets.

Table D–3. Summary of Data Unit Operations

Operation	**Base set ALUs**
Description	Perform an ALU operation in which the opcode specifies an ALU function, one destination and two source operands, and the routing of the operands to the inputs of the ALU. The ALU function can be any one of the 256 three-input Boolean operations or, alternatively, one of 17 arithmetic ALU operations (shown in Table 8–11, *Class-Independent Base Set Arithmetic Functions*, and Table 8–12, *Class-Specific Base Set Arithmetic Functions*) combined with one of 15 ALU function modifiers.
Syntax	dst = [fmod] [⟦cond [.pro] ⟧] ALU_EXPRESSION
Examples	`d6 = (d6 ^ d4) & d2` `d3 =[nn.nv] d1 – 1`
See Page	PP:8-30
Operation	**EALU||ROTATE**
Description	Perform an extended ALU (EALU) operation (as specified in d0) with one of two routings of data to the ALU (as specified in the opcode); the output from the barrel rotator can optionally be written to a second destination register. The ALU function can be any of the 256 Boolean or 256 arithmetic operations. The opcode specifies four source operands and two destination operands. You must enter this syntax on one line.
Syntax	dst1 = [⟦cond [.pro] ⟧] ealu (src2, [dst2 =] [⟦cond⟧] src1[⟦n⟧src1–1]\\src3, [%] src4) dst1 = [fmod] [⟦cond [.pro] ⟧] ealu(label: EALU_EXPRESSION [|| dst2 = ⟦cond⟧src1[⟦n⟧src1–1]\\src3])
Examples	`d7 =[nn] ealu(d2, d6 =[nn] d3\\d1, %d4)` `d3 =mzc ealu(foobar: d4 + (d5\\d6 & %d7) || d1 = d5\\d6)`
See Page	PP:8-66

Table D–3. Summary of Data Unit Operations (Continued)

Operation	**MPY||ADD**
Description	Perform a 16-bit-by-16-bit multiply, yielding a 32-bit result with an add or subtract optionally performed in parallel. The multiply operands can be signed or unsigned. If either operation is conditional, both multiply and add must be conditional, and both must use the same condition code.
Syntax	dst2 = [sign] [⟦cond⟧] src3 * src4 [‖ dst = [⟦cond [.pro] ⟧] src2 + src1[⟦n ⟧src1–1]]
	dst2 = [sign] [⟦cond⟧] src3 * src4 [‖ dst = [⟦cond [.pro] ⟧] src2 – src1[⟦n ⟧src1–1]]
Example	`d7 =u d6 * d5` `|| d5 = d4 – d1`
See Page	PP:8-88
Operation	**MPY||SADD**
Description	Perform a 16-bit-by-16-bit multiply, yielding a 32-bit result with a right-shift and add or subtract performed in parallel. The multiply operands can be signed or unsigned. If either operation is conditional, both the multiply and the shift and add must be conditional, and both must use the same condition code.
Syntax	dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ dst = [⟦cond [.pro] ⟧] src2 + src1[⟦n ⟧src1–1]>>–d0
	dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ dst = [⟦cond [.pro] ⟧] src2 – src1[⟦n ⟧src1–1]>>–d0
Example	`d7 =u d6 * d5` `|| d5 = d4 – d1 >> –d0`
See Page	PP:8-88

Table D–3. Summary of Data Unit Operations (Continued)

Operation	**MPY||EALU**
Description	Perform a multiply and an optional EALU operation in parallel. The multiply can use special features such as rounding, scaling, or splitting (two 8-bit-by-8-bit multiplies instead of one 16-bit-by-16-bit multiply). The EALU provides additional ALU arithmetic functions beyond an add or subtract and enables the special multiply modes. The actual ALU function performed by the EALU is specified in bits 27–19 of the d0 register (ealu) or the 1s complement of those bits (ealuf).
Syntax	**Generic form:** dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ dst1 = [⟦cond [.pro] ⟧] ealu[f] (src2, src1[⟦n⟧src1–1]\\d0, %d0) dst2 = [sign] [⟦cond⟧] src3 * src4 ‖ ealu() **Explicit form:** dst2 = [sign] [opt] [⟦cond⟧] src3 * src4 [<< dms] ‖ dst1 = [fmod] [⟦cond [.pro] ⟧] ealu(label: EALU_EXPRESSION) dst2 = [sign] [opt] [⟦cond⟧] src3 * src4 [<< dms] ‖ ealu(label)

Example	`d7 =[p] d5 * d3 ; generic form`
	`  || d2 =[p] ealu(d1, d6\\d0, %d0)`
	`d2 =m d4 * d7 ; explicit form`
	`  ||d3 = ealu(foobar: d3 + d2 >> 9)`

See Page	PP:8-66
Operation	**divi**
Description	Perform one iteration of an unsigned divide algorithm. Generates one quotient bit per execution by using an iterative subtraction. Additional instructions can be used to implement a signed divide function.
Syntax	dst1=[⟦cond[.pro]⟧] divi (src2, dst2= ⟦cond⟧ src1 [⟦n⟧src1–1])
Examples	`d3 = divi(d1, d2 = d2)` `d3 = divi(d1, d2 = d3[n]d2)`
See Page	PP:8-98

Table D–3. Summary of Data Unit Operations (Continued)

Miscellaneous Operations	
Operation	**dint**
Description	Globally disable interrupts.
Syntax	dint
See Page	PP:8-113
Operation	**eint**
Description	Globally enable interrupts.
Syntax	eint
See Page	PP:8-114
Operation	**nop**
Description	Do nothing in the data unit.
Syntax	nop
See Page	PP:8-115

D.2.2 The Base Set of ALU Operations

Table D–4 summarizes the base set of ALU operation classes. Note that the assembler determines the operation class required for an operation and associates the operands with the corresponding opcode field.

Table D–4. Base Set ALU Operation Class Summary

Class	Dst	A Port Input	B Port Input	C Port Input
0	dst	src2	src1	@mf
1	dst	dstc	src1\\d0	src2
2	dst	dstc	src1	%src2
3	dst	dstc	src1\\src2	%src2
4	dst	src2	src1\\d0	%d0
5	dst	src2	src1\\d0	@mf
6	dst	dstc	src1	src2
7	dst	src1	1\\src2	src2

Notes: 1) **dst** can be any register.

2) **dstc** is the companion D register to the specified destination register (dst) in classes 1, 2, 3, and 6 and is used as the input to the A port.

3) **src2** can be either a D register or a 5-bit or 32-bit immediate. If it is a 32-bit immediate, no parallel operations can be specified.

4) **\\d0** specifies rotation left by the default rotation amount specified in d0's five LSBs.

5) **%d0** specifies mask generation according to the default rotation amount specified in d0's 5 LSBs.

6) **@mf** specifies bit expansion of the (32 / datasize) LSBs (if sr⟦R⟧ = 0) or MSBs (if sr⟦R⟧ = 1) of the mf register.

The functions performed by the ALU can be any one of 256 possible Boolean combinations of the three inputs or 17 arithmetic operations. Table D–5 shows some of these combinations.

Table D–5. Combinations of Base Set Boolean and Arithmetic Operations

Base Set Boolean Operations	Base Set Arithmetic Operations
dst = 0	dst = A + B
dst = ~A & ~B & ~C	dst = A +C
dst = A & ~B & ~C	dst = A – B
dst = ~B & ~C	dst = A – C
dst = ~A & B & ~C	dst = A + (B | C)
dst = (A ^ B) & ~C	dst = A + (B & C)
dst = A & ~C	dst = A + (B & ~C)
dst = (A & ~C) | (~B & ~C)	dst = A – (B | C)
.	dst = A – (B & C)
.	dst = A – (B & ~C)
.	dst =(A & C) + (B & C)
dst = (A ^ ~B) | C	**Class 3 or 4**
dst = A | C	dst = A + b >> [u] n
dst = A | ~B | C	dst = A – b >> [u] n
dst = B | C	**Class 1, 2, 3, 6, or 7**
dst = ~A | B | C	dst = A + |B|
dst = A | B | C	dst = A – |B|
dst = 1	**Class 0 or 5**
	dst = A + ((B & @mf) | (–B & ~@mf))
	dst = A – ((B & @mf) | (–B & ~@mf))

Notes: 1) Signed shift is the default. You can indicate an unsigned shift by appending a **u** to the shift operator (for example, >>u).

2) n represents either an immediate or the 2s complement of the five LSBs in the source register(for example, –d0).

3) b represents the input to the barrel rotator.

D.2.2.1 The ALU Function Modifiers

Table D–6 shows the function modifiers and the modifier code. You can use the function modifiers with the base set of arithmetics or you can configure them in the d0 register for EALU operations; however, you cannot use them with MPY||ADD operations or base set Boolean operations.

Table D–6. ALU Function Modifier Codes

Function Modifier Code	Modification Performed
0 0 0 0	Normal operation
0 0 0 1	cin
0 0 1 0	%! if ALU operation uses mask generator; otherwise, lmo
0 0 1 1	%! and cin if ALU operation uses mask generator; otherwise, rmo
0 1 0 0	A port = 0
0 1 0 1	A port = 0 and cin
0 1 1 0	A port = 0 and %! if ALU operation uses mask generator; otherwise, lmbc
0 1 1 1	A port = 0, %!, and cin if ALU operation uses mask generator; otherwise, rmbc
1 0 0 0	mc: multiple arithmetic, mf bit(s) set by carry-out(s)
1 0 0 1	me: multiple arithmetic, mf bit(s) set on the basis of the MSS field in the status register.
1 0 1 0	mrc: multiple arithmetic, rotate mf, mf bit(s) set by carry-out(s).
1 0 1 1	mre: multiple arithmetic, rotate mf, mf bit(s) set on the basis of the MSS field in the status register.
1 1 0 0	mzc: multiple arithmetic, zero mf, mf bit(s) set by carry-out(s).
1 1 0 1	mze: multiple arithmetic, zero mf, mf bit(s) set on the basis of the MSS field in the status register.
1 1 1 0	mx: multiple arithmetic with no mf bits set.
1 1 1 1	Reserved.

Opcode bit	EALU d0 bit
52	28
54	29
56	30
58	31

Note: **cin** is a carry-in to ALU from the carry bit in the status register, **sr**(**C**).
%! is the right-shift-mask modifier (%!0 = 0xFFFF FFFF).
A port = 0 forces a zero input to the A port.

D.2.2.2 The Bit-Detection Modifiers

The following bit-detection operations can be performed on a D register operand. The bit-detection operations return the values given in Table D–7.

- **lmo (Leftmost One)** returns the position of the leftmost (most significant) bit that is a 1. The overflow status bit is set if no bit in the source has a value of 1.
- **rmo (Rightmost One)** returns the position of the rightmost (least significant) bit that is a 1. The overflow status bit is set if no bit in the source has a value of 1.
- **lmbc (Leftmost Bit Change)** returns the position of the leftmost bit that is not the same as the sign bit. The overflow status bit is set if all bits in the source are equal.
- **rmbc (Rightmost Bit Change)** returns the position of the rightmost bit that is not the same as bit 0. The overflow status bit is set if all bits in the source are equal.

Table D–7. The lmo, rmo, lmbc, and rmbc Operation Results

Bit position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Result	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

You can use the bit-detection operations with either a class-independent base set arithmetic or an EALU operation; not a base set Boolean operation.

When a bit-detection operation is performed, the ALU is split at 8 bits. Therefore, if you use the ALU to adjust the output of the bit-detection logic, the ALU result may be correct only through the first 8 bits (8 LSBs).

Only two unique data routings are supported for bit-detection operations. These are summarized in Table D–8.

Table D–8. Modified Base Set ALU Operation Classes for Bit-Detection Operations

0	dst	src2	src1	lmo(src1)
5	dst	src2	src1\\d0	lmo(src1)

Notes: 1) dst must be a D register.
2) src1 must be a D register and is used as the input to both the barrel rotator and the bit detection logic.
3) src2 can be any register or an immediate.
4) lmo can be replaced by rmo, lmbc, or rmbc.

Note that no status setting of the mf register occurs during bit-detection operations.

D.3 The EALU Operations

The supported EALU data paths are summarized in Table D–9. The multiplier data path features supported in parallel with an EALU are described in subsection 8.5.2.

Table D–9. Data Path for EALU Operations

MPY||EALU			EALU||Rotate		
A	B	C	A	B	C
src2	src1\\d0	%d0 or @mf	src2 src2	src1\\src3 src1\\src3	src4 or @mf %src4

Note that you can use conditional source selection for src1.

D.3.1 The Supported EALU Functions

The EALU supports any of the 256 Boolean or 256 arithmetic operations provided by the three-input ALU. Carry-in is controlled by bits in the d0 register; therefore, you can add a 1 to the result of the ALU function (for both Boolean and arithmetic operations) by forcing a carry-in. Alternately, you can take the carry-in from the carry status bit in the sr register (specified by +cin).

Also, you can perform special operations such as absolute value and signed shifts. To do this, use the bits in the d0 register that control the ALU function on the basis of the sign of one of your inputs.

D.3.2 The Boolean Operations

The Boolean operations performed by the ALU are a sum of products controlled by the ALU function code. You can add 1 to the result of the Boolean sum with a carry-ripple, or you can take the carry-in from the carry status bit in the status register.

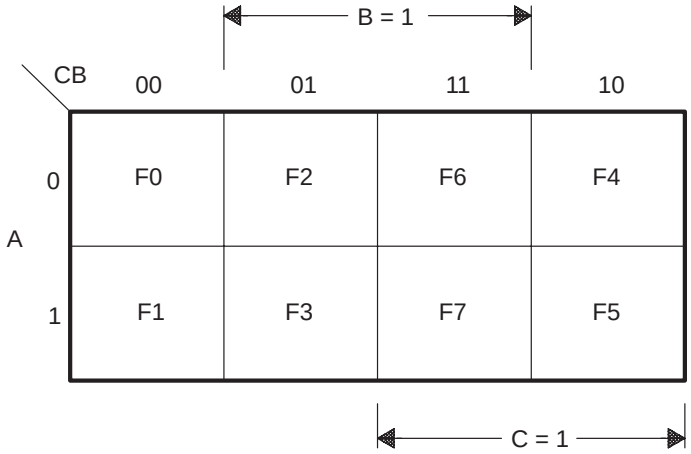
The following Boolean function is the ALU function, derived from the signals in Table D–10:

(F0&(~A&~B~&~C) | F1&(A&~B&~C) | F2&(~A&B&~C) | F3&(A&B&~C) | F4&(~A&~B~&C) | F5&(A&~B&C) | F6&(~A&B&C) | F7&(A&B&C)) [+1|+cin]

Table D–10. ALU Function Code Signals

d0 Bit	Opcode Bit	ALU Function Signal	Products
26	58	F7	A & B & C
25	57	F6	~A & B & C
24	56	F5	A & ~B & C
23	55	F4	~A & ~B & C
22	54	F3	A & B & ~C
21	53	F2	~A & B & ~C
20	52	F1	A & ~B & ~C
19	51	F0	~A & ~B & ~C

Table D–11. ALU Function Code Signal and Karnaugh Map



D.3.3 The Arithmetic Operations

The EALU supports any of the 256 arithmetic operations provided by the three-input ALU. Carry-in is controlled in the d0 register; therefore, you can add a 1 to the result of the ALU function (for both Boolean and arithmetic operations) by forcing a carry-in. Alternately, you can take the carry-in from the carry status bit in the sr register.

arithmetic function = A&f1(B,C) + f2(B,C) [+1 | +cin]

f1(B,C) and f2(B,C) are independent Boolean combinations of B and C, as shown in Table D–12. The ALU function code can be determined by: f1 XOR f2.

Table D–12. Possible f1(B, C) or f2(B, C) Functions

f1 Code	f2 Code	Subfunction	Common Use
00	00	0	Zeros one of the terms
AA	FF	all 1s = –1	All 1s or –1
88	CC	B	B
22	33	–B–1	Negate B
A0	F0	C	C
0A	0F	–C–1	Negate C
80	C0	B&C	Force bits in B to 0 where C is 0
2A	3F	–(B&C)–1	Force bits in B to 0 where C is 0 and negate
A8	FC	B|C	Force bits in B to 1 where C is 1
02	03	–(B|C)–1	Force bits in B to 1 where C is 1 and negate
08	0C	B&~C	Force bits in B to 0 where C is 1
A2	F3	–(B&~C)–1	Force bits in B to 0 where C is 1 and negate
8A	CF	B|~C	Force bits in B to 1 where C is 0
20	30	–(B|~C)–1	Force bits in B to 1 where C is 0 and negate
28	3C	(B&~C)|((–B–1)&C)	Choose B if C = all 0s and –B if C = all 1s
82	C3	(B&C)|((–B–1)&~C)	Choose B if C = all 1s and –B if C = all 0s

D.3.4 Special Operations Supported by the EALU

The EALU supports other operations by using either bits that modify the ALU function and/or carry-in, or the C port. For absolute value or sign extension, the bits that modify the ALU function or carry-in are based on the sign of the input to the barrel rotator. For operations using the C port, you can add or subtract B in each split-ALU segment according to the input to the C port.

EALU operations support taking the **absolute value** of B and optionally adding it to or subtracting it from A. For example:

- [A +] |B|
- [A] – |B|

EALU operations support **shift operations**. The shifted result can be added to or subtracted from A. For example:

- [A +] b << r
- [A] – b << r
- [A +] b >> n
- [A +] b >>u n
- [A] – b >> n
- [A] – b >>u n

The EALU supports **conditional add or subtract,** depending on the input to the C port. These EALU functions assume that there is a uniform mask input to the C port for each split-ALU segment. The uniform mask determines, for each ALU segment, whether B or –B is added to A. For example:

- A + (B&C | –B&~C)
- A + (–B&C | B&~C)

D.4 The Parallel Transfers

Both the local and global address units support the 12 addressing modes shown in Table D–13.

Table D–13. PP Addressing Modes

Description	Syntax	
	Unscaled Index	**Scaled Index**
Postaddition of index register with modify	*(am++=xn)	*(am++=[xn])
Postsubtraction of index register w/ modify	*(am– –=xn)	*(am– –=[xn])
Preaddition of index register	*(am+xn)	*(am+[xn])
Presubtraction of index register	*(am–xn)	*(am–[xn])
Preaddition of index register w/ modify	*(am+=xn)	*(am+=[xn])
Presubtraction of index register w/ modify	*(am–=xn)	*(am–=[xn])
Postaddition of offset with modify	*(am++=offset)	*(am++=[offset])
Postsubtraction of offset w/ modify	*(am– –=offset)	*(am– –=[offset])
Preaddition of offset	*(am+offset)	*(am+[offset])
Presubtraction of offset	*(am–offset)	*(am–[offset])
Preaddition of offset w/ modify	*(am+=offset)	*(am+=[offset])
Preaddition of offset w/ modify	*(am–=offset)	*(am –=[offset])

Notes: 1) The ***** symbol indicates address indirection.

2) The **am** in the syntax represents any address register.

3) The **xn** in the syntax indicates an index register in the same address unit as the base address register.

4) Offset indicates an immediate offset.

5) The **[ ]** symbols indicates that the index is scaled according to the data size.

Table D–14 provides a summary, the syntax, and a short example of the supported parallel transfers. The square brackets indicate optional fields. The ⟦ symbol indicates required square brackets.

Table D–14. Summary of Parallel Transfers

Operation	**Load**
Description	Memory-to-register transfer
Syntax	dst = [sign] [size] [⟦ cond ⟧] ∗ addrexp
	dst = [sign] [size] [⟦ cond ⟧] ∗ an.element
Examples	`d3 =uh[n] *(a9++=[2])` `d1 = * a2.sPTC_BCount`
See Page	PP:8-126
Operation	**Store**
Description	Register-to-memory transfer
Syntax	∗ addrexp = [size] src [⟦ n ⟧ src − 1]
	∗ an.element = [size] src [⟦ n ⟧ src − 1]
Examples	`*--a2 = d3` `*a9.sPT_Next = a3`
See Page	PP:8-130
Operation	**Address Unit Arithmetic**
Description	Computed address-to-register transfer
Syntax	dst = [size] [⟦ cond ⟧] & ∗ addrexp
	dst = [size] [⟦ cond ⟧] & ∗ an.element
Examples	`d2 = &*(a3+x0)` `a1 = &*a9.sPT_DstCCount`
See Page	PP:8-139

Table D–14. Summary of Parallel Transfers (Continued)

Moves	
Operation	**Move**
Description	Register-to-register transfer
Syntax	dst = [g] [⟦ cond ⟧] src
Examples	`x2 = mf` `d1 =g d3`
See Page	PP:8-134
Operation	**Field extract move**
Description	Register-to-register transfer extracting and right-aligning one byte or halfword
Syntax	dst = [sign] [size item] ⟦ cond ⟧ src
Example	`d3 =ub2 d1`
See Page	PP:8-136
Operation	**Field replicate move**
Description	Register-to-register transfer replicating the LSbyte or LShalfword to 32 bits
Syntax	dst = r [size] ⟦ cond ⟧ src
Example	`d7 =rh d3`
See Page	PP:8-138

Table D–15 shows some addressing shortforms supported by the assembler.

Table D–15. Supported Addressing Shortforms

Description	Syntax	Equivalent Addressing Mode
Indirect	* am	* (am+0)
Indirect Index	* xn	* (zero + xn)
Absolute	* offset	* (zero + offset)
Postincrement	* am++	* (am++= [1])
Postdecrement	* am– –	* (am– –= [1])
Preincrement	* ++am	* (am+= [1])
Predecrement	* – –am	* (am–= [1])

The immediate index ranges for the different data sizes and scaling options are shown in Table D–16.

Table D–16. Immediate Index Range

	Scaled Offset		Unscaled Offset	
Category	**Short**§	**Long**§	**Short**†	**Long**‡
byte	–	–	–15 to +15	–0x8FFF to +0x8FFF
halfword	–14 to +14	–0xFFFE to +0xFFFE	–7 to +7	–0x7FFF to +0x7FFF
word	–28 to +28	–0x1FFFC to +0x1FFFC	–7 to +7	–0x7FFF to +0x7FFF

† 4-bit short offset range possible for byte data because the scale bit is used as an additional offset bit.
‡ 16-bit long offset range possible for byte data because the scale bit is used as an additional offset bit.
§ Scaled offset must be a multiple of the data size. For scaled halfword, only even offsets within the specified range can be obtained; for scaled word, only offsets within the specified ranges that are a multiple of 4 can be obtained.

Table D–17 shows the PP-relative addressing modes.

Table D–17. PP-Relative Addressing Modes

Description	Syntax
Postaddition of index register with modify	*xba,am=xba + xn
Postsubtraction of index register w/ modify	*xba,am=xba– xn
Preaddition of index register	*(xba+xn)
Presubtraction of index register	*(xba–xn)
Preaddition of index register w/ modify	*(am=xba + xn)
Presubtraction of index register w/ modify	*(am=xba – xn)
Postaddition of offset with modify	*xba,am=xba + offset
Postsubtraction of offset w/ modify	*xba,am=xba – offset
Preaddition of offset	*(xba+offset)
Presubtraction of offset	*(xba–offset)
Preaddition of offset w/ modify	*(am=xba + offset)
Presubtraction of offset w/ modify	*(am=xba – offset)

Notes: 1) The ***** symbol indicates address indirection.

2) The **am** in the syntax represents any address register.

3) The **xn** in the syntax indicates an index register in the same address unit as the base address register.

4) Offset indicates an immediate offset.

D.5 The Interprocessor Commands

You can issue an interprocessor command by setting up the desired 32-bit command word in a register and then moving the contents in that register to the **cmnd** keyword. Example D–1 shows a PP-independent command issued to halt the PP running the code and a message interrupt sent to the MP.

Example D–1. Using the cmnd Keyword

```
d7 = comm & 7       ;extract the PP number.
d7 = 0x1\\d7        ;convert the PP number to the
                    ;corresponding PP command word
                    ;designator bit.
d7 = 0x40002100 |d7;OR designator bits with command
                    ;to halt the PP and send a message
                    ;interrupt to the MP.
cmnd = d7
```

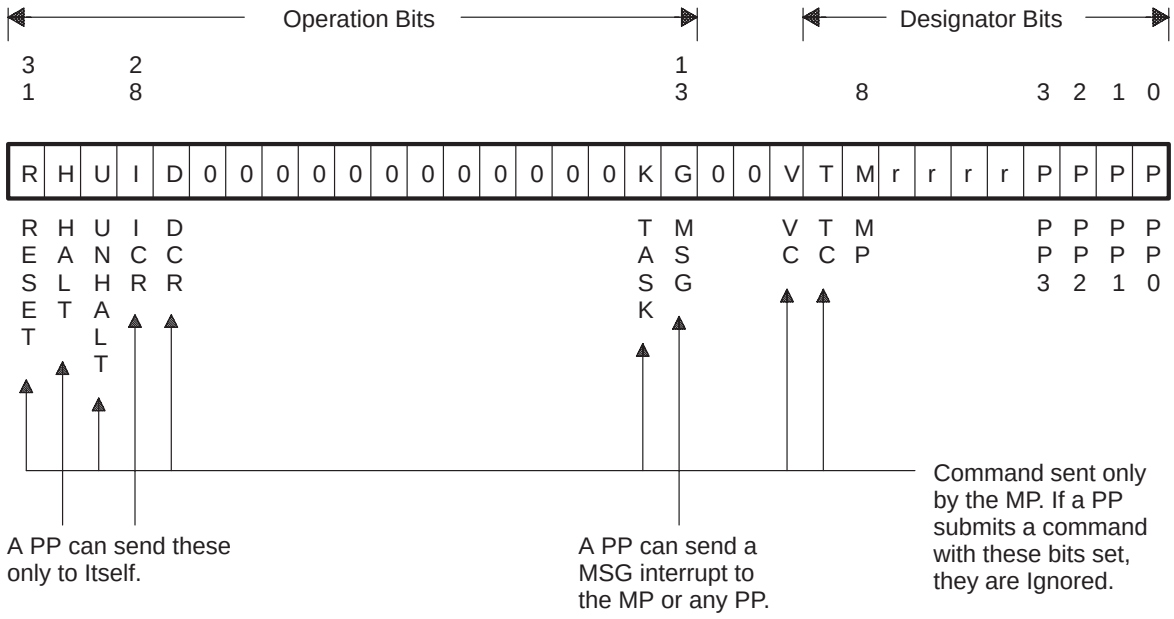
Example D–2 shows that sending a command word is a special case of a register-to-register move, where the destination register is a15.

Example D–2. A Special Case Move to the a15 Register

```
d4 = d5 & d2 || cmnd = d1 || d6 = b * (a4 + = [x0] )
```

The general format for a command word sent by the MVP processors is shown in Figure D–2. Note that a command word contains some bits indicating the desired operation(s) and other bits indicating the destination processor(s). Only those processor(s) identified by the designator bits will respond to the encoded operation.

Figure D–2. Command Word Bit Assignment



D.6 The Condition Codes

The 16 condition codes are shown in Table D–18. The syntax column specifies the symbol used in the assembly-language syntax to represent the condition. The condition is described in the center column, and the rightmost column shows the condition as a logical expression involving the N, C, V, and Z status bits of the sr register.

Table D–18. The Condition Codes

Syntax	Description	Status Bit Combination
u	unconditional (default)	—
p	positive	~N&~Z
ls	lower than or same	~C|Z
hi	higher than	C&~Z
lt	less than	(N&~V)|(~N&V)
le	less than or equal	(N&~V)|(~N&V)|Z
ge	greater than or equal	(N&V)|(~N&~V)
gt	greater than	(N&V&~Z)|(~N&~V&~Z)
hs, c	higher than or same, carry	C
lo, nc	lower than, no carry	~C
eq, z	equal, zero	Z
ne, nz	not equal, not zero	~Z
v	overflow	V
nv	no overflow	~V
n	negative	N
nn	not negative	~N

D.7 The Registers

The PP register codes are shown in Table D–19.

Table D–19. The Register Codes

Lower Registers (MSB of Bank = 0)						Upper Registers (MSB of Bank = 1)					
Coding			Coding			Coding			Coding		
Bank	Reg	Register	Bank	Reg	Register	Bank	Reg	Register	Bank	Reg	Register
0000	000	a0	0100	000	d0	1000	000	reserved	1100	000	lc0
0000	001	a1	0100	001	d1	1000	001	reserved	1100	001	lc1
0000	010	a2	0100	010	d2	1000	010	reserved	1100	010	lc2
0000	011	a3	0100	011	d3	1000	011	reserved	1100	011	reserved
0000	100	a4	0100	100	d4	1000	100	reserved	1100	100	lr0
0000	101	reserved	0100	101	d5	1000	101	reserved	1100	101	lr1
0000	110	a6 (sp)	0100	110	d6	1000	110	reserved	1100	110	lr2
0000	111	a7 (zero)	0100	111	d7	1000	111	reserved	1100	111	reserved
0001	000	a8	0101	000	reserved	1001	000	reserved	1101	000	lrse0
0001	001	a9	0101	001	sr	1001	001	reserved	1101	001	lrse1
0001	010	a10	0101	010	mf	1001	010	reserved	1101	010	lrse2
0001	011	a11	0101	011	reserved	1001	011	reserved	1101	011	reserved
0001	100	a12	0101	100	reserved	1001	100	reserved	1101	100	lrs0
0001	101	reserved	0101	101	reserved	1001	101	reserved	1101	101	lrs1
0001	110	a14 (sp)	0101	110	reserved	1001	110	reserved	1101	110	lrs2
0001	111	a15 (zero)	0101	111	reserved	1001	111	reserved	1101	111	reserved
0010	000	x0	0110	000	reserved	1010	000	reserved	1110	000	ls0
0010	001	x1	0110	001	reserved	1010	001	reserved	1110	001	ls1
0010	010	x2	0110	010	reserved	1010	010	reserved	1110	010	ls2
0010	011	reserved	0110	011	reserved	1010	011	reserved	1110	011	reserved
0010	100	reserved	0110	100	reserved	1010	100	reserved	1110	100	le0
0010	101	reserved	0110	101	reserved	1010	101	reserved	1110	101	le1
0010	110	reserved	0110	110	reserved	1010	110	reserved	1110	110	le2
0010	111	reserved	0110	111	reserved	1010	111	reserved	1110	111	reserved
0011	000	x8	0111	000	pc/call	1011	000	reserved	1111	000	reserved
0011	001	x9	0111	001	ipa/br	1011	001	reserved	1111	001	reserved
0011	010	x10	0111	010	ipe#	1011	010	reserved	1111	010	reserved
0011	011	reserved	0111	011	iprs	1011	011	reserved	1111	011	reserved
0011	100	reserved	0111	100	inten	1011	100	reserved	1111	100	tag0#
0011	101	reserved	0111	101	intflg	1011	101	reserved	1111	101	tag1#
0011	110	reserved	0111	110	comm	1011	110	reserved	1111	110	tag2#
0011	111	reserved	0111	111	lctl	1011	111	reserved	1111	111	tag3#

Notes: 1) **Bank** identifies the bank (or type) of register.

2) **Reg** represents the three LSBs of the register code, specifically, which register within a bank of registers. For a non-D register, these values indicate the companion D register.

3) **pc/call** is used to read the pc register. It can also be used to write to the pc register with an associated saving of the return address into the iprs register; thus, it is referred to as the call register because it can be used to perform subroutine calls.

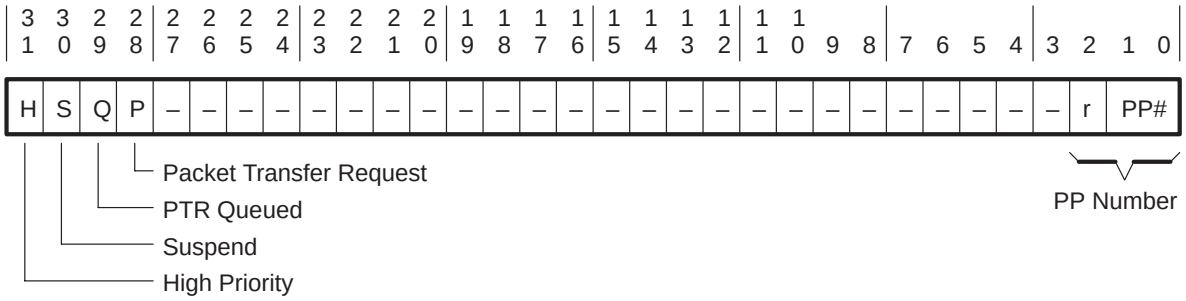
4) **ipa/br** is used to read ipa (a read-only register). It can also be used to write to the pc register without an associated saving of a return address into iprs; thus, it is referred to as the br register because it can be used to perform branches.

5) # symbol indicates read-only registers.

D.7.1 The comm Register

Figure D–3 shows the format for the comm register.

Figure D–3. The comm Register Format



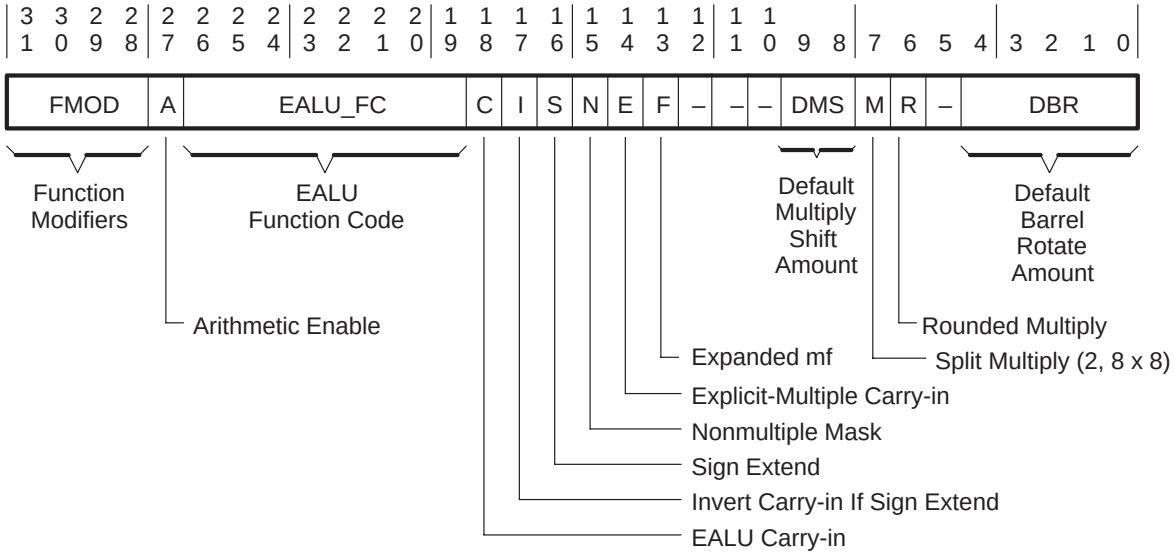
Note:

The comm register can be loaded only with 32 bits (it cannot be loaded by a sign- or zero-extended 8- or 16-bit value).

D.7.2 The d0 Register

Figure D–4 shows the format for d0, the EALU configuration register.

Figure D–4. The d0 Register Format



D.7.3 The inten and intflg Registers

Figure D–5 and Figure D–6 show the format for the inten and intflg registers.

Figure D–5. The inten Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r	r	r	r	E	E	E	E	–	–	–	E	E	E	E	–	–	E	–	–	–	–	–	–	–	–	–	–	–	–	–	W
Interrupts				PP3MSG	PPP2MSG	PPP1MSG	PP0MSG					MPMSG	PTENDR	PTERR	PTQ																
																	TASK														

Note: The **W** bit (bit 0) controls the effect of writing a 1 to bits in the intflg register. If W = 0, then writing a 1 to a bit in the intflg register clears that bit. If W = 1, then writing a 1 to a bit in the intflg register sets that bit.

Figure D–6. The intflg Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r	r	r	r	I	I	I	I	–	–	–	I	I	I	I	–	–	I	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Interrupts				PP3MSG	PPP2MSG	PPP1MSG	PP0MSG					MPMSG	PTENDR	PTERR	PTQ																
																	TASK														

Note: Bits that are labeled with an **r** are reserved for additional message interrupts.

D.7.4 The lctl Register

Figure D–7 shows the format for the lctl register.

Figure D–7. The lctl Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	E	LCD2			E	LCD1			E	LCD0		
																				le2				le1				le0			

Note: LCDx Coding: which loop counter to decrement when loop end is reached
000 – None. Always load loop start into the pc field.
001 – lc0. Allowed for LCD0, LCD1, and LCD2.
010 – lc1. Allowed only for LCD1.
011 – lc2. Allowed only for LCD2.
1xx – Reserved.

D.7.5 The sr register

Figure D–8 shows the format for the sr register.

Figure D–8. The sr Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	C	V	Z	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	MSS		R	Msize			Asize	
Status Bits																															

- The R bit specifies rotation when @mf is specified in the instruction.
  - When R = 1, rotation of mf is specified, and @mf uses the MSBs from mf for expansion.
  - When R = 0, mf is not rotated, and @mf uses the LSBs from mf for expansion.

  If @mf is not specified in the instruction, the R bit is ignored.
- The Msize field specifies the number of bits for the mf register expansion and the mf register rotation amount when using the expander.
  - 0xx – Reserved
  - 100 – Byte (4-bit rotation)
  - 101 – Halfword (2-bit rotation)
  - 110 – Word (1-bit rotation)
  - 111 – Reserved

- The Asize field specifies the size of the split ALU, the number of status bits saved to the mf register, and the number of bits the mf register is rotated left (using the mrc and mre modifiers).
  - 0xx – Reserved
  - 100 – Byte (4-bit rotation)
  - 101 – Halfword (2-bit rotation)
  - 110 – Word (1-bit rotation)
  - 111 – Reserved
- The MSS (mf status selection) field in the status register selects which type of ALU result information is used to set the mf register for the me, mre, and mze function modifiers. The function modifiers enable the setting of bits in the mf register, as shown in Table D–20.

Table D–20. MSS Field Values

MSS Field		
Bit 8	**Bit 7**	**Value/Action**
0	0	mf bit(s) set by zero result(s)
0	1	mf bit(s) set by sign bit(s)
1	0	mf bit(s) set by extended result bit(s) for signed arithmetic
1	1	Reserved

Note that the selection of the carry bits setting of the mf register are given explicitly by mc, mrc, and mzc function modifier options.

The rules for status setting are as follows:

- Only ALU operations set status.
- For conditional operations, status is set only if the condition is true.
- Software writes to the sr register dominate over normal status setting.
- Status protection dominates over software writes to the sr and status setting.

The individual status bits are set as follows:

- **N: Negative (sr bit 31)**
  - **Normal.** Set to the sign bit of the result of the ALU operation.
  - **Multiple Arithmetic.** Set to the AND of the zero compares for the split ALU.
  - **Bit Detection.** Set to the AND of the zero compares for the split ALU.
  - **Divide.** If you are executing the divi operation, the N bit is set according to the following expression:

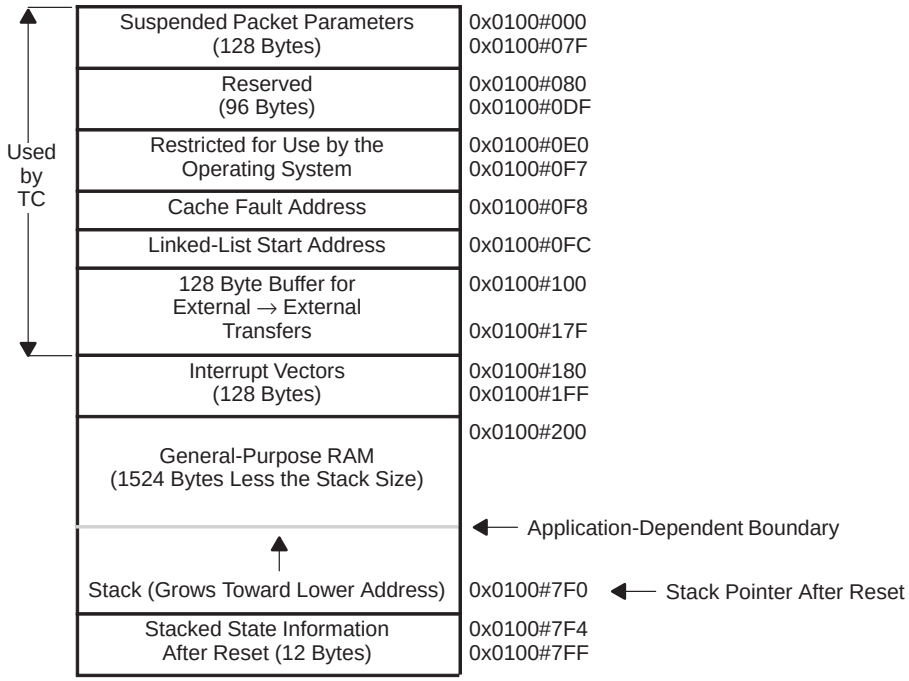
    carry-out | bit 31 of the input to the barrel rotator
- **C: Carry (sr bit 30)**
  - **Normal.** Set to the value of the carry-out ($\overline{\text{borrow}}$) from bit 31 of the ALU.
  - **Multiple Arithmetic.** Records the OR of the split-ALU carry-outs.
  - **Bit Detection.** Records the OR of the split-ALU carry-outs.
- **V: Overflow (sr bit 29)**
  - **Normal.** Set to the XOR of the carry-in and carry-out of bit 31.
  - **Multiple Arithmetic.** Set to the AND of the split-ALU carry-outs.
  - **Bit Detection.** For the lmo and rmo operations, V = 1 indicates that no 1s were present in the input word.

    For the lmbc and rmbc operations, V = 1 indicates that all bits in the source are the same.
- **Z: Zero (sr bit 28)**
  - **Normal.** Set to 1 if the result of the ALU operation was equal to 0; set to 0 if the result was nonzero.
  - **Multiple Arithmetic.** Set to the OR of the zero compares for the split ALU.
  - **Bit Detection.** Set to the OR of the zero compares for the split ALU.

D.8 The Memory Maps

Figure D–9 shows the memory map of a PP's parameter RAM.

Figure D–9.PP Parameter RAM Contents



Note: The # symbol represents the PP number.

When an enabled interrupt flag is set, the corresponding PP interrupt vector is fetched from the PP's local parameter RAM at the addresses shown in Figure D–10.

Figure D–10. Interrupt Vector Locations

	32 Bits	Address (# = PP Number)
Lowest	Reserved for PP7 Message	0x0100 #1FC
	Reserved for PP6 Message	0x0100 #1F8
	Reserved for PP5 Message	0x0100 #1F4
	Reserved for PP4 Message	0x0100 #1F0
	PP3 Message	0x0100 #1EC
	PP2 Message	0x0100 #1E8
	PP1 Message	0x0100 #1E4
	PP0 Message	0x0100 #1E0
	Reserved	0x0100 #1DC
	Reserved	0x0100 #1D8
	Reserved	0x0100 #1D4
	Master Processor Message	0x0100 #1D0
	Packet Transfer End	0x0100 #1CC
	Packet Transfer Error	0x0100 #1C8
	Packet Transfer Queued	0x0100 #1C4
	Reserved	0x0100 #1C0
	Reserved	0x0100 #1BC
	Task Interrupt	0x0100 #1B8
	Reserved	0x0100 #1B4
	Reserved	0x0100 #1B0
	Reserved	0x0100 #1AC
	Reserved	0x0100 #1A8
	Reserved	0x0100 #1A4
	Reserved	0x0100 #1A0
	Reserved	0x0100 #19C
	Reserved	0x0100 #198
	Reserved	0x0100 #194
	Reserved	0x0100 #190
	Reserved	0x0100 #18C
	Reserved	0x0100 #188
	Reserved	0x0100 #184
	Reserved	0x0100 #180

Priority

Highest

Appendix E

Glossary

A

address stage: The second stage of the PP's FAE pipeline during which addresses are calculated and supplied to the crossbar. See also *FAE pipeline sequence*

address unit: Hardware on the PP that computes a 32-bit address during each cycle. Each PP has two address units: a global address unit and a local address unit.

address unit arithmetic: The PP's use of the local and global address units to perform general-purpose arithmetics in parallel with the data unit. The computed address is not used for memory access, but is stored in the destination register.

addressing modes: The PP's methods of computing addresses to access memory. Each PP address unit supports twelve addressing modes, including preindexed and postindexed addressing.

ALU: *Arithmetic logic unit*. Hardware that provides the logic for arithmetic and Boolean operations.

ALU function: For the PP, an action performed on the three inputs to the ALU, which includes any arithmetic or Boolean combination of the three inputs, as well as mixed arithmetic and Boolean functions.

ALU function modifier: For the PP, a 4-bit code that specifies modifications to the functions performed by the ALU data path (such as carry-in or multiple arithmetic). These function modifiers are specified in the opcode or in the d0 register, depending on the application.

ALU operation: For the PP, an action performed by the ALU data path (that is, the result of the ALU function, the operation class, and any function modifiers).

argument buffer: The memory block into which the argument values that accompany a command to a server PP are placed.

assembler: A software utility that creates a machine-language program from a source file. There are two assemblers associated with the MVP: a mnemonic-based RISC-type assembler for the MP and an algebraic assembler for the PP.

B

barrel rotator: A device that rotates the position of bits within a data word. It is similar to a barrel shifter except that bits shifted out are wrapped around to the vacated bits.

base set ALUs: The PP's fundamental set of ALU operations, which includes Boolean as well as mixed arithmetic and Boolean functions.

base set arithmetics: The fundamental set of PP instructions that specify an ALU arithmetic operation in the opcode, which includes eleven class-independent arithmetic operations and six class-specific arithmetic operations.

base set Booleans: The fundamental set of PP instructions that specify an ALU Boolean operation in the opcode, which includes the 256 possible Boolean functions.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *little endian*

bitBLT: *Bit-aligned block transfer*. Transfer of a block of pixels from one location in a bitmap to another.

bit detection: The special logic that supports leftmost-one, rightmost-one, leftmost-bit-change, and rightmost-bit-change detection.

bitmap: 1) A digital representation of an image in which bits are mapped to pixels. 2) A block of memory used to hold raster images in a device-specific format.

block miss: A cache miss in which the addressed block is not resident in the cache. The least recently used (LRU) algorithm determines which existing cache block is discarded. If the cache contains any modified data (MP data cache only), then any modified subblocks are written back to external memory before the requested subblock is brought into cache.

block write: A nonstandard packet transfer that allows the TC to perform multicolumn write operations.

bus watching: The processor's ability to see ahead and prepare for changes in address bus usage.

butterfly: A kernel function for computing N-point fast Fourier transform (FFT), where N is a power of 2. The combinational pattern of inputs resembles butterfly wings.

C

C compiler: A program that translates C source statements into assembly language source statements or object code.

cache: A fast memory into which frequently used data or instructions from slower memory are copied for fast access. Fast access is facilitated by the cache's high speed and its on-chip proximity to the CPU.

cache block: A section of cache memory. Each block has an associated tag register and is divided into four subblocks. Cache memory is allocated in block-size portions, but cache servicing is performed at the subblock level, with subblocks brought in as needed.

cache coherency: The state or condition in which the contents of one or more cache memories consistently and accurately represent the corresponding contents of the external memory.

cache flush: An MP instruction that updates external memory by writing modified (dirty) data-cache subblocks back to memory, thus resetting that subblock's present and dirty bits to 0.

cache miss: The state or condition in which the cache does not contain the requested instruction or data word.

cache subblock: One of four partitions of a cache block. Cache subblocks are the unit of memory brought into a cache on a subblock miss. Each subblock has a present bit (and a dirty bit for MP data cache only) in the tag register for that block.

cache tag register: A register containing the address of the block whose subblock(s) have been copied into cache. It also contains a present bit for each subblock indicating whether or not the subblock is present in the cache. For MP data cache, there is also a dirty bit for each subblock.

class: See *operation class*

class-independent arithmetics: The PP's eleven arithmetic ALU functions in the base set of ALU operations that are available with any of the eight operation classes.

class-specific arithmetics: The PP's arithmetic functions in the base set of ALU operations that are available only for a subset of the eight operation classes.

coherency: See *cache coherency*

conditional source: A source operand, either the odd or even register in a D register pair, depending on the negative status bit.

contention: A situation where two or more simultaneous access attempts for the same 2K-byte RAM are made. Contention is resolved automatically in hardware by arbitration, though a delay can occur.

crossbar: A generally configurable, high-speed bus switching network for a multiprocessor system, permitting any of several processors to connect to any of several memory modules.

D

data cache: The MP's two SRAM banks that hold cached data needed by the MP. Data RAMs for the PPs are not cached.

data RAM: On-chip RAM that is available for the general-purpose storage of data by the MP or PPs on the MVP.

data unit: The PP's data manipulation hardware unit that includes the ALU, the multiplier, the mf expander, and the barrel rotator.

data unit operation: For the PP, the operations performed by the hardware in the data unit. The data unit allows both a multiply and an ALU data path operation to be performed in a single cycle.

dba: The assembler keyword for the PP-relative base address to local RAM0 (LRAM0) (0x#000 where # is the PP identification number). The dba can be used to generate addresses in PP-independent code.

DCT: *Discrete cosine transform.* A fast Fourier transform used in manipulating compressed still and moving picture data. See also *FFT*, *JPEG standard*

DEA: *Direct external access.* A method of accessing off-chip (external) memory without having to issue a packet transfer request to the TC.

debugger: A window-oriented software interface that helps you to debug MVP programs running on an MVP emulator or simulator.

delta-guided transfer: A type of guided packet transfer in which the guide table consists of 32-bit delta values to be added to the starting address of the previous two-dimensional patch to form the starting address of the new patch. See also *guided transfer*

destination port: The message port to which a message is sent.

dimensioned transfer: A transfer consisting of sources and/or destinations that can be a simple contiguous linear sequence of data bytes or can consist of a number of such regions. See also *guided transfer*

direct external access: See *DEA*

double buffering: A method of using dual buffers to achieve efficient one-way data transmission between two processors or between a processor and a peripheral device. Each buffer is a block of storage through which data is transmitted from one processor (or device) to the other. The receiving processor reads the transmitted data from one buffer while the sending processor simultaneously prepares the data for the next transmission in the alternate buffer.

doubleword: A 64-bit value.

E

EALU: *Extended arithmetic logic unit.* The PP's set of operations that extend the normal ALU functions by specifying the ALU operation and/or multiplier features in the d0 register instead of in the opcode.

EALU||ROTATE: The PP's EALU variation that allows you to save the result of the barrel rotation to an additional destination register.

execute stage: The second stage of the MP's FEA pipeline and the third stage of the PP's FAE pipeline; these stages operate differently for the MP and the PPs. For the MP's execute stage, the instruction is decoded, source operands are read from the registers, the operation is performed, and the results are written into the destination register. For the PP, all data unit operations occur, as well as memory accesses (loads and stores) and register-to-register moves. See also *data unit operation*

executive: The portion of a multitasking software system that is responsible for executing application tasks, providing communications among tasks, and managing shared resources.

expander: For the PP, the ALU data path hardware that can expand 1-, 2-, or 4-bit values by respectively duplicating each bit by 32, 16, or 8 times to expand the bit(s) out to a 32-bit word. The value to be expanded can be contained in the multiple flags register (mf), the only register connected to the PP's expander.

extended ALU: See *EALU*

external address: See *off-chip address*

externally initiated packet transfer: See *XPT*

F

FAE pipeline sequence: *Fetch, address, execute.* The instruction-execution unit pipeline for the PP. The fetch stage includes instruction and operand fetch, the address stage includes address computation and possible crossbar access, and the execute stage includes data unit operations, memory accesses, and register-to-register moves.

fast Fourier transform: See *FFT*

fetch stage: First stage of the MP's FEA pipeline as well as the PP's FAE pipeline, during which instructions and their operands are fetched. See also *FAE pipeline sequence*

FFT: *Fast Fourier transform*. An efficient method for computing the discrete Fourier transform, which is used to transform functions between the time domain and frequency domain. The time-to-frequency domain is called the forward transform, and the frequency-to-time domain is termed the inverse transformation. See also *butterfly*

field extract move: A PP's register-to-register move that right-justifies a specified byte or halfword from the source register and writes the result with a 0 or sign extension to the destination register. The source register must be a D register, while the destination register can be any PP register.

field replicate move: A PP's register-to-register move that replicates the lower byte or halfword in the source register to fill 32 bits and writes the result to the destination register. The source register must be a D register, while the destination register can be any register.

fixed-patch guided transfer: Guided transfer that uses an on-chip guide table consisting of 32-bit word-aligned entries. See also *guided transfer*

flush: See *cache flush*

G

global address unit: PP hardware that generates addresses using a global address register (a8 – a12, a14, a15) with preaddition, postaddition, presubtraction, or postsubtraction of a global index register (x8 – x10) or an immediate offset.

global legal: A PP's access that can be performed over the global port because the address is in the shared RAMs. If an access is not globally legal, a direct external access (DEA) is performed by the TC. The PP's pipeline stalls until the DEA is completed.

global port: The data port with access to any of the shared RAMs (any PP data RAM or parameter RAM).

global transfer: A PP's operation performed by the global address unit. This can be either a memory access (load or store), an address unit arithmetic, or a register-to-register move.

guide table: A table of parameters describing individual patches within a packet transfer. See also *patch*

guided transfer: A transfer in which the sequence of dimension addresses is guided from an on-chip memory table, rather than calculated solely from values within the packet transfer parameters. See also *dimensioned transfer*

H

halfword: A 16-bit value.

I

instruction bus: A processor-dependent bus used to access instructions from on-chip SRAM. The PPs each use a 64-bit instruction bus, and the MP uses a 32-bit instruction bus.

instruction cache: An on-chip SRAM that contains current instructions being executed by one of the MVP processors. Cache misses are handled by the transfer controller.

instruction port: The port used by the PP for communication with its instruction cache.

internal address: See *on-chip address*

interprocessor command: A message sent via the crossbar to the other on-chip processors.

interrupt: An exceptional condition caused either by an event external to the processor or by a previously executed instruction that forces the current program to be interrupted. After the processor has serviced the interrupt, it typically resumes execution of the interrupted program at the instruction whose execution was interrupted.

interrupt latency: The time from an interrupt request to the execution of the first instruction of that interrupt's service routine.

IP: *Instruction pointer.* The MP register that points to the instruction currently in the fetch stage of the pipeline.

ipa: *Instruction pointer address.* The PP register that points to the current statement in the address pipeline stage of your program.

ipe: *Instruction pointer execution.* The PP register that points to the current statement in the execute pipeline stage of your program.

ISR: *Interrupt service routine*. A module of code that is executed in response to a hardware or software interrupt.

J

JPEG standard: *Joint Photographic Experts Group standard*. A standard used for compressed still-picture data.

L

linker: A software tool that combines object files to form an object module that can be allocated into system memory and executed by the device.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*

lmbc: *Leftmost-bit change.* The PP operation that returns the position of the leftmost bit that has a value different from the sign bit.

lmo: *Leftmost 1.* The MP or PP operation that returns the position of the leftmost bit that has a value of 1. In the MP, lmo is an instruction, while in the PP, it is a bit detection function.

local address unit: PP hardware that generates addresses using a local address register with preaddition, postaddition, presubtraction, or postsubtraction of an index register or an immediate offset.

local bus: A 32-bit bus allocated to each PP to access on-line SRAM data in a single cycle.

local legal: An access in a PP's local RAM that can be performed through the local port. Such access is done in a single cycle. If an access is attempted via the local port but is not local legal, access must be diverted to the global port, causing a pipeline stall.

local port: The PP data port with access restricted to a PP's local RAMs.

local RAM: The on-chip RAM that is associated with a particular PP in an MVP.

local transfer: A local address unit operation. This is a memory access that usually occurs over the local port.

long offset: An unsigned 15-bit offset for halfword or word transfers, or an unsigned 16-bit offset for byte transfers.

look-up table: A table, used during scan conversion of a digital image, that converts color-map addresses into the actual color values displayed.

LPRAM: The PP-relative reference to a PP's local 2K-byte parameter RAM.

LRAM0: The PP-relative reference to a PP's local 2K-byte data RAM bank with the lowest address.

LRAM1: The PP-relative reference to a PP's local 2K-byte data RAM bank with the second lowest address.

LRAM2: The PP-relative reference to a PP's local 2K-byte data RAM bank with the highest address.

LRU cache replacement: *Least recently used cache replacement*. A cache management strategy that replaces the least recently used cache block in memory (while retaining the blocks more recently used) when a cache block-miss occurs.

LSB: *Least significant bit.* The bit having the smallest effect on the value of a binary numeral, usually the rightmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 0 is the LSB.

M

mailbox: A 32-bit word in a PP's parameter RAM in which it places messages to clients (chiefly, the MP).

mask generator: A PP's ALU data path hardware device that takes a 5-bit input n and outputs a mask containing n 1s specified by the input when it is right-justified.

master processor: See *MP*

memory fault: An exception caused by an attempt to access an illegal or invalid address in memory.

memory map: A map of target system memory space that is partitioned into functional blocks.

MIMD: *Multiple instruction stream, multiple data stream.* A parallel processing structure composed of multiple independent processors.

miscellaneous operation: One of several operations that do not involve the data unit, including nop, eint, and dint. The data unit portion of the opcode is used to specify the operation.

MP: *Master processor*. A general-purpose RISC processor that coordinates the activity of the other processors on the MVP. The MP includes an IEEE-754 floating-point hardware unit.

MPEG standard: *Moving Picture Experts Group standard.* A proposed standard for compressed video data.

MPY||ADD: A PP's data unit opcode format that allows a multiply to be specified in parallel with an add or subtract by the ALU. See also *ALU*

MPY||EALU: A PP's data unit opcode format that allows a multiply to be specified in parallel with an extended ALU operation. See also *EALU*

MSB: *Most significant bit*. The bit having the greatest effect on the value of a binary numeral. It is the leftmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 31 is the MSB.

multimedia video processor: See *MVP*

multiple arithmetic: A PP's ALU operation in which the carry from bit to bit is disabled at certain points in the ALU, causing the ALU to act as multiple smaller ALUs. A status bit, either carry-out or 0, from each ALU segment is saved in the mf register. Multiple arithmetic is also referred to as split ALU.

multiple-byte arithmetic: Multiple arithmetic with ASize set to byte. The ALU acts like four parallel byte ALUs because the carry-path is broken between bits 24 and 23, bits 16 and 15, and bits 8 and 7.

multiple-halfword arithmetic: Multiple arithmetic with Asize set to halfword. The ALU acts like two parallel ALUs because the carry-path is broken between bits 16 and 15.

MVP: *Multimedia video processor*. A single-chip multiprocessor device that accelerates applications such as video compression and decompression, image processing, and graphics. The multimedia video processor contains a master processor and from one to eight parallel processors, depending on the device version. For example, the TMS320C80 device contains four PPs.

MVP multitasking executive: See *executive*

N

non-D operand: A non-D register used as an operand in a PP instruction.

O

off-chip address: An address external to the MVP chip. Addresses from 0x0200 0000 to 0xFFFF FFFF are off-chip addresses. See also *on-chip address*

on-chip address: An address internal to the MVP chip. Addresses from 0x0000 0000 to 0x1FFF FFFF are on-chip addresses. See also *off-chip address*

operation class: Specific multiplexer setting combinations that cause the same ALU function to perform various distinct operations by controlling routing of operands to the ALU. Eight operation classes are supported by the base set of ALU operations.

P

packed bytes: Four bytes contained in a 32-bit word. The individual bytes within a word of packed bytes are often operated on in parallel using multiple-byte arithmetic.

packed halfwords: Two halfwords contained in a 32-bit word. The individual bytes within a word of packed halfwords are often operated on in parallel using multiple-halfword arithmetic.

packet: A collection of patches of data. See also *patch*

packet transfer: See *PT*

packet transfer request: An I/O request submitted to the TC that is issued when a block of data is to be moved via packet transfer. Packet transfer requests can be submitted by the MP, the PPs, the VC, or an external device.

parallel processor: See *PP*

parallel transfers: The address unit operations specified in parallel with a data unit operation.

parameter RAM: A general-purpose 2K-byte RAM that is associated with a specific processor, part of which is dedicated to packet transfer information and the processor interrupt vectors.

patch: A group of lines of equal length whose starting addresses are an equal distance apart.

pba: The assembler keyword for the PP-relative parameter RAM base address that can be used to set up address registers for PP-independent code. pba corresponds to 0x0100 n000, where n is the PP number.

PC field: *Program counter field*. The 29-bit PP or 30-bit MP counter field within the 32-bit PC register that contains the address of the next instruction.

PC register: The 32-bit register that contains the address of the next instruction (PC field). In the PP, the PC register also includes the G and L control bits.

pipeline stall: Temporary halt to the normal fetching of operations. Events which cause a pipeline stall include: a cache-miss, an illegal operation detection, diversion of local port access to global port, a DEA, and crossbar contention.

pipelining: A design technique for reducing the effective propagation delay per operation by partitioning the operation into a series of stages, each of which performs a portion of the operation. A series of data is typically clocked through the pipeline in sequential fashion, advancing one stage per clock period.

PIXBLT: *Pixel-block transfer.* A pixel-array operation in which each pixel is represented by one or more bits. PIXBLTs are a superset of bitBLTs and include the commonly used Boolean functions as well as integer arithmetic and multibit operations. See also *bitBLT*

poll: A continuous test used by the program until a desired condition is met.

PP: *Parallel processor*. The MVP's advanced digital signal processor that is used for video compression/decompression (P × 64 or MPEG), still-image compression/decompression (JPEG), 2-D and 3-D graphic functions such as line draw, trapezoid fill, antialiasing, and a variety of high-speed integer operations on image data. An MVP single-chip multiprocessor device may contain from one to eight PPs, depending on the device version.

ppcl: A shell utility that invokes the MVP's PP compiler, assembler, and linker to create an executable object file version of your PP program.

PP command interface: The software interface through which the MP (or other client processor) issues commands to be executed by a server PP.

present flag: A bit in the cache tag register associated with a cache subblock that indicates whether the information in the subblock is present in the cache.

program flow control unit: A unit that manages the opcode fetches from the PP's instruction cache.

PT: *Packet transfer*. A transfer of data blocks between two areas of memory. The MVP supports packet transfers of one, two, or three dimensions. See also *dimensioned transfer*, *guided transfer*

PT options field: Packet-transfer parameter field which selects the form of transfer for source and destination. It determines if the packet will end the linked list and enables the selection of additional features such as special transfer modes.

R

reset: A means to bring processors to known states by setting registers and control bits to predetermined values and signaling execution to start at a specified address. At reset, the MP loads the address 0xFFFF FFF8 into the PC register.

RISC: *Reduced instruction set computer.* A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt response from a smaller, cost-effective chip.

rmbc: *Rightmost-bit change.* The PP operation that returns the position of the rightmost bit that has a value different from bit 0. The overflow status bit is set if all bits in the source are identical.

rmo: *Rightmost 1.* The operation that returns the bit position of the rightmost 1. In the MP, it is an actual instruction, while in the PP, it is a bit-detection function.

S

shared RAMs: Memory that can be shared by MVP's processors. This includes the 12 PP data RAMs and 4 PP parameter RAMs.

short offset: Unsigned 3-bit immediate index for halfword or word transfers; unsigned 4-bit immediate index for byte transfers. A short offset can be used in an addressing mode for any other operations specified in parallel in the instruction.

software write: A write to the destination register specified in the MP or PP instruction. Software writes take precedence over writes performed automatically by hardware, such as the increment of the program counter or the setting of ALU status.

split ALU: See *multiple arithmetic*

split multiply: A PP operation that performs two simultaneous 8-bit by 8-bit multiplies. The PP multiplier can perform two 8-bit unsigned by 8-bit unsigned multiplies or two 8-bit unsigned by 8-bit signed multiplies.

SRAM banks: *Static random access memory banks*. These include parameter and data RAM and instruction and data caches.

stack pointer: A special-purpose 32-bit register that contains (points to) the address of the top of the system stack.

subblock: See *cache subblock*

subblock miss: A cache miss where the desired block is present but the desired subblock is not. Results in a pipeline stall until the required subblock is brought into cache.

T

tag: 1) An optional type name that can be assigned to a structure, union, or enumeration. 2) A register holding the address of the cache block. See also *cache tag register*

task interrupt: An MVP hardware mechanism through which the MP cmnd instruction can signal a task interrupt to one or more PPs if their interrupts are enabled. The MP uses this mechanism to spawn tasks on the various PPs.

TC: *Transfer controller*. The MVP's on-chip DMA controller for servicing the cache and for transferring one-, two-, and three-dimensional data blocks between each processor on the MVP and its external memory.

transfer controller: See *TC*

transparency: A pixel attribute that renders a source pixel invisible so that portions of the destination array show through portions of the source array.

V

variable-patch guided transfer: A type of guided transfer in which all patch size information is specified within the guide table rather than in the packet transfer parameters, allowing each patch within the transfer to have different dimensions. See also *delta-guided transfer*, *offset-guided transfer*, *guided transfer*

VC: *Video controller*. The portion of the MVP responsible for the video interface.

video controller: See *VC*

W

word: A sequence of 32 adjacent bits that constitutes a register or memory value. The PP supports 32-bit words. The MP also supports doublewords of 64 bits for loads and stores.

X

xba: The assembler keyword for a PP-relative base address in local RAM, either data or parameter RAM. After memory allocation is performed by the linker, xba is changed to either dba or pba, depending on where space is allocated by the linker.

XPT: *Externally initiated packet transfer*. A packet transfer initiated by an external device through the MVP's $\overline{\text{XPT}}$ [2:0] inputs.

Index

A

A (address computation)
  stage PP:5-3

A0–A4 registers PP:7-5

A14 register
  mapping with the stack pointer
    PP:6-3

A15 register
  address unit arithmetic PP:8-141
  read as 0 PP:6-3

A6 register
  mapping with the stack pointer
    PP:6-3

A7 register
  address unit arithmetic PP:8-141
  read as 0 PP:6-3

A8–A12 registers PP:7-6

absolute branching. *See* branching, absolute

absolute value PP:4-52
  class-specific arithmetics PP:8-42
  EALU operations PP:8-79
  multiple absolute value PP:8-42

add-with-saturate PP:4-63

addition
  adder/subtracter unit PP:6-4
  adding two masked inputs PP:4-64
  carry-in functions PP:4-16
  conditional PP:4-66, PP:8-80

addition (continued)
  shift operations
    See also shift operations
    shift left PP:4-42
    signed shift right PP:4-50
    unsigned shift right PP:4-46
    using PP:4-18, PP:4-39

address registers PP:6-11
  modifying PP:5-7

address stage instruction pointer
  See also IPA register
  program counter PP:1-14

address unit PP:6-1 to PP:6-22
  See also global address unit; local address unit
  adder/subtracter unit PP:6-4
  block diagram PP:6-2
  overview PP:1-10 to PP:1-11, PP:6-2
  registers PP:6-3
    accessible PP:1-17
    summary PP:1-16

address unit arithmetic PP:1-11, PP:6-21, PP:8-139 to PP:8-141
  conditional PP:8-141
  syntax PP:8-139
  writing to any PP register PP:6-10

addressing
  indexed PP:6-3
  key features PP:1-3

addressing (continued)
modes
long offset PP:8-117
short offset PP:8-117
postindexed PP:6-4
PP-relative addressing PP:8-124
restrictions in parallel PP:8-159
xba keyword PP:8-124
preindexed PP:6-4

advanced EALU. *See* EALU, advanced

algorithms
BitBlt (bit-aligned block transfer) PP:4-54
cache replacement PP:3-6

ALU
See also arithmetic operations; base set ALU operations; Boolean operations; EALU
base set PP:8-30 to PP:8-60
branch tables PP:10-15
deriving functions PP:4-24
description PP:1-8
multiplexers PP:4-31
natural arithmetic functions PP:4-14
rules in function code PP:4-20
split ALU PP:4-9
matching pixel values PP:4-56
three-input PP:4-11, PP:4-18
EALU operators PP:8-64
rotation PP:8-86
two-input PP:10-17

AND operator PP:4-53

applications
bitstream decoding PP:15-32 to PP:15-40
graphics PP:15-2 to PP:15-14
Bresenham line drawing PP:15-6
color expand PP:15-3

applications, graphics (continued)
pixel block transfers (PIXBLTs) PP:15-7 to PP:15-15
add-with-saturate PP:15-11
fills PP:15-8
MAX/MIN PP:15-9
pixel processing PP:15-9
plane masking PP:15-13
subtract-with-saturate PP:15-12
transparency PP:15-14
image-processing PP:15-15 to PP:15-31
binary morphology PP:15-18 to PP:15-24
cosine transform PP:15-28 to PP:15-32
mean absolute difference PP:15-24 to PP:15-28
thresholding PP:15-16
parallel processor PP:1-4, PP:15-1 to PP:15-40

architecture
overview PP:1-5 to PP:1-20

arithmetic operations
See also ALU; Boolean operations; EALU; function; modifiers; operations; shift operations
class-independent arithmetics PP:8-40
class-specific arithmetics PP:8-41
equations PP:4-21
functions PP:4-11
multiple PP:4-28
See also add-with-saturate; MAX function; MIN function; multiple arithmetic; transparency
multiple-byte data flow diagram PP:4-29
natural arithmetic functions PP:4-14
opcode formats PP:A-15
operators. *See* operators

arithmetic operations (continued)
quick reference PP:D-16
table of useful arithmetic functions PP:4-19

assembly language instruction set. *See* instruction set

AUA. *See* address unit arithmetic

B

barrel rotator
combining two or more functions PP:4-38
description PP:1-9
right-justifying fields PP:4-53
shift operations PP:4-39
using PP:4-7

base set ALU operations PP:4-12, PP:8-30 to PP:8-60
See also ALU
class-independent PP:8-47
class-specific PP:8-48
opcode formats PP:A-15
operands PP:8-32 to PP:8-37
quick reference PP:D-10 to PP:D-14

big-endian
addressing PP:2-9
dimensioned packet transfer parameters PP:12-19
ordering PP:2-10

binary number system PP:C-1 to PP:C-10
signed numbers PP:C-4
unsigned numbers PP:C-4

bit-detection
logic PP:15-38
modifiers. *See* modifiers
status setting PP:8-55
using PP:4-10

bit fields
Asize
MF register PP:7-29
SR register PP:7-41
clearing PP:7-31
expanding in the MF register PP:7-30
Msize
MF register PP:7-30
SR register PP:7-40
setting in the MF register PP:7-28
status bits in the SR register PP:7-37

BitBlt algorithm PP:4-54

bitstream decoding
applications PP:15-32 to PP:15-40

bitwise logical operations PP:4-11
bitwise AND PP:4-23
using PP:4-14

block diagram
address units PP:6-2
data unit PP:4-2
PP PP:1-6
program flow control unit PP:5-2

Boolean operations PP:4-11, PP:13-4 to PP:13-5
base set ALU operations PP:8-38 to PP:8-42
bitwise operators. *See* operators
explicit EALU expressions PP:8-74 to PP:8-76
products PP:13-3
quick reference PP:D-15
three-input ALU PP:8-39
truth table PP:13-4
two-input ALU PP:10-17
using three-input PP:4-14

br register code PP:5-4, PP:7-32, PP:10-3

branching PP:10-1 to PP:10-18
absolute PP:10-11
branch tables PP:10-15

branching (continued)
conditionally PP:10-13
See also conditional operations
conditional assignment PP:8-146
hardware branches PP:11-26 to PP:11-28
software branches PP:11-24 to PP:11-25
zero-overhead PP:5-13
FAE pipeline PP:5-8
hardware PP:5-13, PP:11-21 to PP:11-23
relative PP:10-12
software PP:10-2
within a loop PP:5-14

bus
global address port PP:6-6
global data port PP:6-6
global destination PP:6-7, PP:6-10
global source PP:6-7, PP:6-10
instruction address port PP:6-7
instruction port PP:6-7
local address port PP:6-6
local data port PP:6-6
local destination or source PP:6-6, PP:6-9
structure PP:6-6
overview PP:1-12

byte addressability PP:8-120

byte ordering
diagram PP:2-10

C

cache
See also cache controller
architecture PP:3-2 to PP:3-4
detecting an interrupt message PP:3-14
diagram PP:3-3
hit PP:3-4
management PP:3-1 to PP:3-14
miss PP:3-5
stall condition PP:5-10

cache (continued)
replacement algorithm PP:3-6
reset PP:3-6, PP:3-13
self-modifying code PP:3-7
subblocks PP:3-2

cache controller PP:5-15 to PP:5-16
overview PP:1-14

cache service requests PP:3-5
resetting PP:9-19

cache tag register. *See* TAG register

call register code PP:5-4, PP:7-32, PP:10-3

calls PP:10-1 to PP:10-18
conditional PP:10-13
See also conditional operations
conditional assignment PP:8-146
nested PP:10-10
subroutine
return sequence PP:10-8 to PP:10-10

carry-in PP:4-16
See also modifiers
addition PP:4-16
EALU operations PP:13-7
explicit-multiple (E bit) PP:13-11
invert (I bit) PP:13-9
specification PP:13-13
starting value PP:13-8
instruction set PP:4-17

carry-out PP:4-16
addition PP:C-2
subtraction PP:4-16, PP:C-2
threshold detection PP:4-57

carry status bit PP:4-13, PP:7-38, PP:8-150

cin function modifier PP:8-58
See also modifiers

classes. *See* operation classes

cmnd keyword PP:3-8

code
alignment PP:3-7
pipelined tight loop PP:14-2
address unit arithmetic PP:14-13
mean squared error (MSE) PP:14-8
two computations PP:14-10
straight-line PP:14-3
address unit arithmetic PP:14-9
mean squared error (MSE) PP:14-7
multiple arithmetic PP:14-11

color
expanding PP:4-54 to PP:4-55
transparency PP:4-55

COMM register PP:7-7
polling PP:12-10
summary PP:1-16

command word
bit assignments PP:3-11
sending PP:3-9
using PP:3-11 to PP:3-14

communications
interprocessor
See also interprocessor communications
keyword PP:3-9

companion registers
base set ALU operands PP:8-36

conditional branches. *See* branching, conditionally

conditional operations PP:8-144 to PP:8-155
See also function; operations; status bit protection
assignment PP:8-146 to PP:8-147
branches PP:1-13
condition codes PP:8-145
masking PP:4-24
source selection PP:8-148 to PP:8-149

conditional operations (continued)
status bit protection PP:8-150 to PP:8-156
swap PP:8-147
using PP:4-65

contention
resolving PP:6-17

crossbar
contention
stall condition PP:5-10
description PP:2-7 to PP:2-8
diagram PP:2-8
ports
global port PP:2-7
instruction port PP:2-7
local port PP:2-7

D

D registers PP:4-3
base set ALU operands PP:8-33
conditional source selection PP:8-148
D0 register PP:7-10 to PP:7-14
D0–D7 registers PP:7-9
description PP:1-7
load/store via the local destination or source bus PP:6-9
parallel write priority PP:8-171 to PP:8-174
summary PP:1-16

D0 register PP:7-10 to PP:7-14
multiply fields PP:8-90

D0–D7 registers PP:7-9

data
formats
16x16 multiply PP:4-68
rounded PP:4-69
signed split multiply PP:4-71
unsigned split multiply PP:4-72

data path
combining two or more functions PP:4-38
hardware PP:4-6
multiplier PP:4-67
operation classes PP:4-32
base set diagram PP:4-34
six-operand diagram PP:4-35
using multiplexers PP:4-31

data RAM
accessing PP:2-2
dba keyword PP:8-124
description PP:2-3
off-chip memory PP:2-4

data registers. *See* D registers

data streams
controlling multiple data streams PP:4-59 to PP:4-63

data unit
architecture PP:4-1 to PP:4-28
block diagram PP:1-6, PP:4-2
definition PP:8-2
expander PP:4-9
hardware PP:4-6
inputs and outputs PP:4-4
key features PP:1-2
multiplier
description PP:1-8
opcode PP:A-2
formats PP:A-8 to PP:A-14
mnemonics PP:A-5
operations
overview PP:8-3
quick reference PP:D-5 to PP:D-13
restrictions in parallel PP:8-158
specifying in parallel PP:10-15
summary PP:8-4 to PP:8-7
operators PP:8-12
See also modifiers; operators
overview PP:1-7 to PP:1-9
registers PP:4-3
accessible PP:1-17

dba keyword PP:8-124

DEA (direct external access)
PP:2-2 to PP:2-6
latency PP:6-18
resetting service requests PP:9-19
stall condition PP:5-11

delay-slot instruction PP:3-10
branching PP:10-4 to PP:10-7
FAE pipeline PP:5-8

designator bit
halt command PP:3-12

dimensioned transfers PP:12-16 to PP:12-26
double buffering PP:12-23
row-based processing PP:12-24
setting up
packet transfer parameters PP:12-19

DINT operation
syntax PP:8-113

direct external access. *See* DEA

divi operation
encoding PP:8-103
syntax PP:8-98 to PP:8-102

division PP:8-96 to PP:8-102
See also divi operation
MF register PP:7-31

E

E (execute) stage PP:5-3

EALU PP:4-12
See also ALU; arithmetic operations; Boolean operations; EALU||ROTATE operation; EALUF operation; MPY||EALU operation
advanced PP:13-1 to PP:13-14
data path PP:8-63 to PP:8-64
expressions
explicit PP:8-74
function code PP:13-2

EALU (continued)
operations PP:8-61 to PP:8-86
arithmetic EALU operations PP:13-6 to PP:13-7
conditional PP:8-77, PP:8-80
explicit EALU expressions PP:8-77 to PP:8-78
quick reference PP:D-14
special operations PP:8-79
unrelated rotate PP:8-86
operators PP:8-64
rules in function code PP:4-20
syntax PP:8-65 to PP:8-72
explicit syntax PP:8-69 to PP:8-73
generic syntax PP:8-66 to PP:8-68

EALU||ROTATE operation
data path PP:8-63
explicit EALU syntax PP:8-69
field extract move PP:8-84
generic EALU syntax PP:8-66
operation classes PP:4-33
six-operand diagram PP:4-36

EALUF operation PP:8-95 to PP:8-102
image-processing applications PP:15-30

EINT operation
syntax PP:8-114

equations
arithmetic operations PP:4-21
masking both sides PP:4-26
subfunctions PP:4-21

execute stage instruction pointer
See also IPE register
program counter PP:1-14

executing code
sequential program execution PP:1-13

expander
color expansion PP:4-54 to PP:4-55
data size PP:7-40
description PP:1-9
multiple arithmetic PP:4-59 to PP:4-63
R bit in the SR register PP:7-39
using PP:4-9

explicit EALU syntax
EALU||ROTATE operation PP:8-69
MPY||EALU operation PP:8-69

expressions
masking PP:4-15
merge PP:4-15
reducing with bitwise AND PP:4-23

extended set of ALU operations. *See* EALU

F

F (fetch) stage PP:5-3

FAE pipeline PP:5-3
See also pipelines
delay-slot instruction PP:5-8
implications PP:5-6
modifying registers PP:5-7

field move operations
See also move operations; register-to-register transfers (move)
conditional opcode format PP:A-34
field extract move PP:6-20, PP:8-84, PP:8-136 to PP:8-137
operation classes PP:4-33
overview PP:1-11
right-justifying fields PP:4-53
syntax PP:8-136
field replicate move PP:6-20, PP:8-138 to PP:8-140
overview PP:1-11
syntax PP:8-138

function
See also absolute value; arithmetic operations; Boolean operations; operations
absolute value PP:4-27, PP:4-52
arithmetic PP:4-18
ALU PP:4-24
equations PP:4-21
multiple PP:4-28
multiple-byte data flow diagram PP:4-29
combining two or more PP:4-38
conditional PP:4-24
definition PP:4-6
deriving PP:4-24
factoring PP:4-22
mask generator PP:4-8
masking PP:4-24
MAX PP:4-62
MIN PP:4-62
natural arithmetic PP:4-14
rules in ALU and EALU operations PP:4-20
sign-extend PP:4-27
syntaxes for the PP PP:8-10

function modifiers. *See* modifiers

G

GAU. *See* global address unit

general-purpose registers
global address (A8–A12) PP:7-6
local address (A0–A4) PP:7-5

generic EALU syntax
EALU||ROTATE operation PP:8-66
MPY||EALU operation PP:8-66

global address unit
A15 register PP:6-3
forcing a move PP:8-134
g modifier PP:8-134
global address registers (A8–A12) PP:7-6
overview PP:1-10

global buses
address port PP:6-6
data port PP:6-6
destination PP:6-7, PP:6-10
source PP:6-7, PP:6-10

global port PP:1-12, PP:2-7

global transfers PP:8-116
restrictions in parallel PP:8-161

graphics
applications PP:15-2 to PP:15-14
Bresenham line drawing PP:15-6
color expand PP:15-3
pixel block transfers (PIXBLTs) PP:15-7 to PP:15-15
add-with-saturate PP:15-11
fills PP:15-8
MAX/MIN PP:15-9
pixel processing PP:15-9
plane masking PP:15-13
subtract-with-saturate PP:15-12
transparency PP:15-14
color expansion PP:4-54 to PP:4-55

guided transfers PP:12-27 to PP:12-32
fixed-patch PP:12-30
line draw
fill-with-value PP:12-28
pixel processing PP:12-31
pixel fill PP:12-30

H

halting
command word PP:3-8
halting a PP PP:3-12
stall condition PP:5-11

hardware
branching PP:5-13
See also branching, conditionally, hardware branches
parallel processor
global destination or source bus PP:6-10
local destination or source bus PP:6-9
replicate hardware PP:6-8

history
program counter history PP:5-5

I

image-processing
applications PP:15-15 to PP:15-31
binary morphology PP:15-18 to PP:15-24
cosine transform PP:15-28 to PP:15-32
mean absolute difference PP:15-24 to PP:15-28
thresholding PP:15-16

immediate index range PP:8-121

immediate operands
32-bit immediate PP:8-37
format PP:1-19
5-bit immediate PP:8-37

#include
files
packetpp.i PP:B-1 to PP:B-6

index registers PP:6-11
modifying PP:5-7
parallel transfers PP:8-118
scaling PP:6-4
summary PP:1-16

index scaler PP:6-4

indexing
immediate index range PP:8-121
negative immediate index PP:8-122
scaled indexing PP:8-121

input
acceptable inputs for the data unit PP:4-4
adding with two masked inputs PP:4-64
barrel rotator PP:4-7

instruction-cache PP:3-2
controller
overview PP:1-13
description PP:2-3
diagram PP:3-3
hit. *See* cache
program flow control unit PP:5-15
reset PP:3-13
command word PP:3-8

instruction fetch PP:3-4

instruction pipeline
stalling PP:1-11, PP:6-7

instruction pointer
IPA register PP:5-4, PP:7-17
IPE register PP:5-4, PP:7-18
IPRS register PP:5-5, PP:7-19
registers
summary PP:1-16

instruction port PP:1-12, PP:2-7, PP:6-7
overview PP:6-7

instruction set PP:8-1 to PP:8-3
overview PP:1-18 to PP:1-20, PP:8-2 to PP:8-3

instructions
pop instruction PP:8-142
push instruction PP:8-142

INTEN register PP:7-14
interrupt-related register PP:9-4

interprocessor communications PP:3-1 to PP:3-14
commands PP:3-8 to PP:3-14
delay-slot instruction PP:3-10
issuing a command PP:3-9
keyword PP:3-9
quick reference PP:D-23

interrupt flags PP:9-10 to PP:9-12
clearing PP:9-10
saving/storing PP:9-11

interrupt registers PP:9-2 to PP:9-7
INTEN register PP:7-14
interrupt bits PP:3-14
INTFLG register PP:7-15
summary PP:1-16

interrupt service routine
guidelines PP:9-16
returning from PP:9-17

interrupts PP:9-1 to PP:9-22
disabling PP:9-8
enabling PP:9-8
halt command effects PP:3-12
packet transfer
PTEND flag PP:12-12
PTQ flag PP:12-12
priorities PP:9-13
processing PP:9-15
vector locations PP:9-13

INTFLG register PP:7-15
interrupt-related register PP:9-6

inverse operations
ealuf operation PP:8-95
shift-and-add function PP:8-95
shift-and-subtract function PP:8-95

IPA register PP:7-17
instruction pointer address
stage PP:5-4

IPE register PP:7-18
instruction pointer execute
stage PP:5-4

IPRS register PP:7-19
instruction pointer return from
subroutine PP:5-5
interrupt service routine PP:9-16

ISR. *See* interrupt service routine

K

keyword
cmnd PP:3-8
dba PP:8-124
pba PP:8-124
sp PP:7-34, PP:8-142
xba PP:8-124

L

LC0–LC2 registers PP:7-20

LCN register
initializing loop values PP:11-8

LCTL register PP:7-21
designating associated loop counter
PP:11-9
enabling loop ends PP:11-9

LE0–LE2 registers PP:7-23
loop end priority PP:11-16

linked list
pointer
setting PP:12-4
Q bit polling PP:12-10

linker
PP-relative addressing PP:8-124

little-endian
addressing PP:2-9
ordering PP:2-10

lmbc operation PP:4-31
bit-detection modifier PP:8-52

lmo operation PP:4-31
bit-detection modifier PP:8-52

load operations PP:1-10,
PP:8-126 to PP:8-129
access range PP:8-126
conditional load PP:8-127
syntax PP:8-127 to PP:8-130
via the local destination or source
bus PP:6-9

local address unit
A7 register PP:6-3
local address registers (A0–A4)
PP:7-5
overview PP:1-10

local buses
address port PP:6-6
data port PP:6-6
destination or source PP:6-6, PP:6-9

local port PP:1-12, PP:2-7
local address registers (A0–A4) PP:7-5

local RAM
accessing PP:1-11, PP:2-2
description PP:2-4
local address port bus PP:6-6
pipeline stalls PP:6-7

local transfers PP:8-116
restrictions in parallel PP:8-161
specifying in parallel PP:10-15

logical operations
unsigned shift right PP:4-44 to PP:4-45
with addition PP:4-46
with subtraction PP:4-47

long offset PP:6-3, PP:8-117
parallel transfer opcode format PP:A-29 to PP:A-32
restrictions in parallel PP:8-159

loop control logic PP:11-1 to PP:11-28
See also loop control registers; loop controller
flexibility PP:11-19
hardware branching PP:11-21 to PP:11-23
shortform initialization
multi-instruction loop PP:11-12
single-instruction loop PP:11-10

loop control registers
See also loop control logic; loop controller
LCTL register PP:7-21
LE0–LE2 registers PP:7-23

loop control registers (continued)
loop counter
decrement and reload PP:11-5
enabling PP:11-9
LC0–LC2 registers PP:7-20
multiple ends/same counter PP:11-19
LR0–LR2 registers PP:7-24
LRS0–LRS2 registers PP:7-25
LRSE0–LRSE2 registers PP:7-26
LS0–LS2 registers PP:7-27
shortform initialization PP:11-10
summary PP:1-16

loop controller PP:5-12 to PP:5-14
See also loop control logic; loop control registers
loop end comparison PP:11-5
overview PP:1-14
registers PP:11-3
initialization PP:11-6 to PP:11-13

looping PP:11-2
initializing
counter value PP:11-8
start and end addresses PP:11-8
loop end priority PP:11-16
multiple nested loops PP:5-13
nested loops PP:11-14
single loops PP:5-12

lower registers
base set ALU operands PP:8-35
register code PP:7-2

LR0–LR2 registers PP:7-24

LRN register
initializing loop values PP:11-8

LRS0–LRS2 registers PP:7-25

LRSE0–LRSE2 registers PP:7-26

LS0–LS2 registers PP:7-27

LUT
individual access PP:14-3 to PP:14-4
pipelined access PP:14-5
tight loop PP:14-2

M

mask generator
  description PP:1-9
  operator (%) PP:8-24 to PP:8-25
  right-justifying fields PP:4-53
  shift operations PP:4-39
  using PP:4-8

masking PP:4-18, PP:4-24
  adding with two masked inputs PP:4-64
  both sides of equations PP:4-26
  expressions PP:4-15

master processor. *See* MP

MAX function PP:4-62

memory
  See also off-chip memory
  addressing
    key features PP:1-3
  organization PP:2-1 to PP:2-10

memory accesses
  aligned access PP:8-120
  generating addresses PP:6-14
  loading PP:6-13
  local RAM PP:6-11
  off-chip accesses PP:6-18
  resolving contention PP:6-17
  storing PP:6-13
  transferring data PP:6-15

memory mapping PP:2-2 to PP:2-6
  See also memory accesses

memory-to-register transfers
  (load) PP:1-10, PP:8-126 to PP:8-129
  See also load operations

merge operations
  merging control PP:4-15

message interrupt
  command word PP:3-8
  detecting PP:3-14
  setting the message interrupt flags PP:3-14

MF register PP:4-3, PP:7-28 to PP:7-32
  bit expansion PP:7-30
  bit setting PP:7-28
  clearing PP:7-31
  color expansion PP:4-54
  description PP:1-8
  divides PP:7-31
  matching pixel values PP:4-56
  rotation PP:7-30
  using with the expander PP:4-9

MIMD model PP:2-2, PP:3-2

MIN function PP:4-62

miscellaneous operations PP:8-113 to PP:8-115
  See also DINT operation; EINT operation; NOP operation

modifiers PP:8-49 to PP:8-59
  See also operators
  A port = 0 PP:8-57, PP:8-82
  bit detection PP:8-52 to PP:8-56, PP:8-81
  carry-in (cin) PP:8-58, PP:8-81
  EALU operations PP:8-81 to PP:8-84
  mask generator (%) PP:8-24
  move done with global unit (=g) PP:8-134
  multiple arithmetic PP:8-50 to PP:8-51, PP:8-82
  right-shift-mask (!) PP:8-24, PP:8-59, PP:8-81
  summary of codes PP:8-60

morphology
  applications PP:15-18 to PP:15-24

move operations PP:8-134 to PP:8-138
  See also field move operations; register-to-register transfers (move)
  conditional opcode format PP:A-32 to PP:A-36
  conditional swap PP:8-135
  field moves
    overview PP:1-11

move operations (continued)
forcing global unit to perform move PP:8-134
replicating bytes or halfwords PP:6-8
syntax PP:8-134
ambiguous move syntax PP:8-135

MP
halting
unhalting a PP PP:3-12

MPY||ADD operation
See also multiply; parallel multiply
operation classes PP:4-32
six-operand diagram PP:4-35
syntax PP:8-88 to PP:8-89

MPY||EALU operation
data path PP:8-63
data routing PP:8-91
explicit EALU syntax PP:8-69
generic EALU syntax PP:8-66
operation classes PP:4-33
parallel PP:8-90
six-operand diagram PP:4-35, PP:4-36

MPY||SADD operation
See also multiply; parallel multiply
operation classes PP:4-32
six-operand diagram PP:4-35
syntax PP:8-88 to PP:8-89

multiple arithmetic PP:4-11, PP:4-28
add-with-saturate PP:4-63
controlling multiple data streams PP:4-59 to PP:4-63
data size PP:7-41
MAX and MIN functions PP:4-62
multiple-byte data flow diagram PP:4-29
transparency PP:4-60
using PP:4-65

multiple data streams PP:4-59 to PP:4-63

multiple flags register. *See* MF register

multiple-instruction-multiple-data. *See* MIMD model

multiplexers
C port PP:4-31
data path operation classes PP:4-32
index addressing PP:6-3
mask generator PP:4-31
preindex/postindex PP:6-4
rotate amount PP:4-31
using PP:4-31

multiplier PP:4-67
description PP:1-8

multiply PP:8-87 to PP:8-95
See also MPY||ADD operation; MPY||EALU operation; MPY||SADD operation; parallel multiply; rounded multiply; split multiply
16x16 PP:4-68
default shift PP:8-92
rounding PP:4-69
scaling PP:4-69
split multiply PP:4-71

MVP
crossbar diagram PP:2-8
memory map PP:2-5

N

natural arithmetic functions. *See* arithmetic operations, natural arithmetic functions

negative status bit PP:4-13, PP:7-37, PP:8-150

negative terms PP:4-21, PP:4-22

nested calls. *See* calls, nested

nested loops. *See* looping, nested loops

non-D registers
base set ALU operands PP:8-34
opcode format PP:A-36
operand format PP:1-19
restrictions in parallel PP:8-160

NOP operation
syntax PP:8-115

O

off-chip memory PP:2-4
See also memory
accesses PP:6-18
specifying in the instruction PP:2-2

offsets PP:6-3
See also long offset; short offset
scaled PP:8-121
unscaled PP:8-121

on-chip memory PP:2-3
See also data RAM; instruction-cache; memory; shared RAM

opcode PP:A-1 to PP:A-2
fetches PP:3-2
formats
32-bit immediate PP:1-19
base set ALU PP:8-30
conditional PP:1-20
data PP:4-72
general PP:1-19
long offset PP:1-20
non-D ALU operand PP:1-19
summary PP:A-3
illegal
stall condition PP:5-10
overview PP:1-18
restrictions PP:8-175

operation bits PP:3-14

operation classes PP:4-6 to PP:4-10, PP:8-43 to PP:8-48
ALU data path PP:4-32
base set diagram PP:4-34
six-operand diagram PP:4-35
summary PP:8-43 to PP:8-44

operations
See also address unit arithmetic; conditional operations; parallel operations; parallel transfers; stack, operations
base set PP:4-12
definition PP:4-6
EALU PP:4-12
fixed PP:14-13
lmbc PP:4-31
lmo PP:4-31
rmbc PP:4-31
rmo PP:4-31
selectable PP:14-13

operators
See also modifiers
& operator PP:8-20
arithmetic PP:8-18
assignment (=) PP:8-15
bitwise Boolean PP:8-20
AND operator (&) PP:8-20
OR operator (|) PP:8-20
unary/NOT operator (~) PP:4-8, PP:8-21
XOR operator (^) PP:8-20
conditional source selection PP:8-17
data unit PP:8-12 to PP:8-29
EALU operators PP:8-64
expand (@mf) PP:8-22
mask generator (%) PP:4-8, PP:8-24
multiplication PP:8-19
operator precedence PP:8-13
rotate (\\) PP:8-26
shift PP:8-27 to PP:8-31
shift left PP:8-27
signed shift right PP:8-29
unsigned shift right PP:8-27

optimization PP:14-1 to PP:14-20
mean squared error (MSE) PP:14-6

output
acceptable outputs for the data unit PP:4-4

overflow
signed number PP:C-7
unsigned number PP:C-5 to PP:C-6
addition PP:C-5
subtraction PP:C-6

overflow status bit PP:4-10, PP:4-13, PP:7-38, PP:8-150

P

packet transfer handshake
signals PP:12-6 to PP:12-9

packet transfer parameters
setting up PP:12-3

packet transfers PP:12-1 to PP:12-32
interrupts PP:9-6
multidimensional transfers PP:2-2
priority PP:12-6
processing loop flow PP:12-12
polling PP:12-13
request
issuing PP:12-5
nested loops PP:11-14
protocol PP:12-2, PP:12-8
resetting PP:9-19
suspending PP:12-7
waiting to complete PP:12-9 to PP:12-15

packetpp.i include file PP:B-1 to PP:B-6

parallel multiply PP:8-83 to PP:8-87
See also parallel operations; parallel transfers
EALU operations PP:8-90

parallel operations PP:6-12, PP:8-156 to PP:8-161
See also parallel multiply; parallel transfers
basic rules PP:8-157
legal PP:8-165
parallel write priority PP:8-171 to PP:8-174

parallel operations (continued)
restrictions
data unit operations PP:8-158
global transfers PP:8-159
local transfers PP:8-160
two parallel transfers PP:8-161
summary PP:8-162

parallel processor. *See* PP

parallel transfers PP:8-116 to PP:8-143
See also address unit arithmetic; load operations; memory-to-register transfers (load); move operations; parallel multiply; parallel operations; register-to-memory transfers (store); register-to-register transfers (move); store operations
addressing modes PP:8-117
definition PP:8-2
global transfers
overview PP:8-3
local transfers
overview PP:8-3
opcode PP:A-2
formats PP:A-21 to PP:A-36
mnemonics PP:A-6
operators PP:8-118 to PP:8-120
overview PP:8-3
quick reference PP:D-18
summary PP:8-8
formats PP:8-162

parallel write priority PP:8-171 to PP:8-174
discarding results PP:8-172
fixed PP:8-173 to PP:8-176

parameter RAM
accessing PP:2-2
branch tables PP:10-15
contents PP:2-6
description PP:2-3
off-chip memory PP:2-4
pba keyword PP:8-124

pba keyword PP:8-124

PC field PP:7-3, PP:7-32

PC register PP:7-3, PP:7-32
branch delay-slot instructions PP:10-5
interrupt-related register PP:9-3
overview PP:1-13
PC field PP:7-3, PP:7-32
related registers PP:5-4

PFC. *See* program flow control unit

pipelines PP:5-3
See also FAE pipeline
effects after interrupts PP:9-9
implications PP:5-6
restrictions PP:8-175
stalling PP:1-11, PP:5-10, PP:6-7
cache miss PP:3-5
with interprocessor commands PP:3-10

pointers
address stage instruction pointer PP:1-14
execute stage instruction pointer PP:1-14

polling PP:2-2
conditional branches and calls PP:10-14
packet transfers PP:12-10
PTEND flag PP:12-11

pop instruction PP:8-142

PP
See also address unit; data unit; program flow control unit
applications PP:1-4
architecture PP:1-5 to PP:1-20
expander PP:1-9
global address unit PP:1-10
local address unit PP:1-10
replicate hardware PP:6-8
block diagram PP:1-6
key features PP:1-2 to PP:1-6
overview PP:1-1 to PP:1-20
description PP:1-2 to PP:1-3

PP (continued)
registers
accessible PP:1-17
key features PP:1-2
MF register
description PP:1-8
overview PP:1-15 to PP:1-17
PC register
overview PP:1-13
SR register
description PP:1-8
summary PP:1-16, PP:7-1 to PP:7-4

PPERROR register
PP halt latch value PP:3-12

processors
halting PP:3-12
resetting PP:3-11
unhalting PP:3-13

program counter register. *See* PC register

program flow control unit PP:5-1 to PP:5-16
block diagram PP:5-2
cache controller
overview PP:1-14
decoding PP:1-13
instruction controller
overview PP:1-13
instruction fetches PP:1-13
key features PP:1-3
loop controllers
overview PP:1-14
looping. *See* looping
overview PP:1-13 to PP:1-14, PP:5-2
PC register
overview PP:1-13
registers
accessible PP:1-17
stalling the address and data units PP:5-10

push instruction PP:8-142

Q

quick reference PP:D-1 to PP:D-2

R

RAM. *See* data RAM; instruction-cache; local RAM; off-chip memory; on-chip memory; parameter RAM; shared RAM

reduction of expressions PP:4-23

register code PP:5-4, PP:7-32
  branches vs calls PP:10-3

register-to-memory transfers (store) PP:1-10, PP:8-130 to PP:8-133
  See also store operations

register-to-register transfers (move) PP:6-20, PP:8-134 to PP:8-138
  See also move operations
  global data port bus PP:6-6
  replicating bytes or halfwords PP:6-8

registers
  address unit PP:6-3, PP:6-11
  data unit PP:4-3
  index PP:6-11
  modifying PP:5-7
  parallel processor
    See also companion registers; D registers; index registers; individual register names; lower registers; non-D registers; upper registers
    accessible PP:1-17
    D registers PP:7-9 to PP:7-14
      description PP:1-7
    general-purpose PP:7-5 to PP:7-7
    instruction pointers PP:7-17 to PP:7-20
    interrupt registers PP:7-14 to PP:7-17
    key features PP:1-2

registers, parallel processor (continued)
    loop counters PP:7-20 to PP:7-27
    multiple flags register PP:7-28 to PP:7-32
      color expansion PP:4-54
      description PP:1-8
    overview PP:1-15 to PP:1-17
    program counter register PP:7-32
      description PP:1-13
      related registers PP:5-4
    register codes PP:7-2
    special-purpose registers PP:7-7, PP:7-34
    status register PP:7-36
      description PP:1-8
    summary PP:1-16, PP:7-1 to PP:7-4

relative addressing. *See* addressing, PP-relative addressing

relative branching. *See* branching, relative

replicate hardware
  overview PP:6-8
  register-to-register transfers (move) PP:6-8

reset
  activity during PP:9-19
  activity following PP:9-21
  hardware PP:9-18
  initial state PP:9-20
  resetting the instruction cache PP:3-13
  software PP:9-18

resetting
  command word PP:3-8, PP:3-11
  resetting a processor PP:3-11

restrictions
  programming PP:8-175 to PP:8-176

reti operation PP:9-17

right-align hardware PP:6-8

rmbc operation PP:4-31
bit-detection modifier PP:8-52

rmo operation PP:4-31
bit-detection modifier PP:8-52

rotation
barrel rotator PP:4-7
MF register PP:7-30
R bit in the SR register PP:7-39
unrelated EALU rotate PP:8-86

rounded multiply PP:8-70, PP:8-94
See also MPY||EALU operation

rounding PP:4-69

rules
ALU operations PP:4-20
EALU operations PP:4-20

S

scaling PP:4-69
index scaler PP:6-4
MPY||EALU operations PP:8-90
scaled indexing PP:8-121

scoreboarding
FAE pipeline PP:5-6

self-modifying code PP:3-7

shared RAM PP:2-2
accessing PP:2-2
description PP:2-4
global address port bus PP:6-6

shift operations PP:4-18, PP:4-39 to PP:4-51
default multiply PP:8-92
EALU operations PP:8-80
inverse functions
shift-and-add PP:8-95
shift-and-subtract PP:8-95
shift left PP:4-40 to PP:4-41
addition PP:4-42
subtraction PP:4-43

shift operations (continued)
signed shift right PP:4-48 to PP:4-49
addition PP:4-50
class-specific arithmetics PP:8-41
subtraction PP:4-51
unsigned shift right PP:4-44 to PP:4-45
addition PP:4-46
subtraction PP:4-47

short offset PP:6-3, PP:8-117

shortform
addressing PP:8-119
initialization PP:11-10
multi-instruction loop PP:11-12
single-instruction loop PP:11-10

sign-extend hardware PP:6-8
memory accesses PP:6-15

sign extension PP:4-18
EALU operations PP:13-7
selection (S bit) PP:13-10
specification PP:13-13
sign-extend hardware PP:6-5

signed split multiply PP:4-71

software branching. *See* branching, software

sp keyword PP:8-142

SP register PP:7-34
interrupt service routine PP:9-16

special-purpose registers
COMM register PP:7-7
SP register (A6 or A14) PP:7-34

split multiply PP:4-71, PP:8-70, PP:8-93
See also MPY||EALU operation

SR register PP:4-3, PP:7-36 to PP:7-41
description PP:1-8
EINT operation affects PP:8-114
result status bits PP:4-13
rotation PP:7-39
setting PP:4-13

SR register (continued)
status bits PP:7-37
status bit protection PP:8-154
status setting PP:7-36
summary PP:1-16

stack
operations PP:8-142
parallel PP:8-143
pointer PP:6-3

stalling
address and data units PP:5-10
PP pipeline PP:3-12

status bit protection PP:8-150 to PP:8-156
restrictions PP:8-153
software writes to the SR register PP:8-154
status bits PP:8-150
syntax PP:8-150

status registers. *See* SR register

status setting. *See* bit-detection, status setting

store operations PP:1-10, PP:8-130 to PP:8-133
access range PP:8-130
conditional store PP:8-131
syntax PP:8-132
via the local destination or source bus PP:6-9

subblock (of memory) PP:3-2

subfunctions PP:4-21

subroutine
calling PP:10-8 to PP:10-10
FAE pipeline PP:5-8
returning from PP:5-5, PP:10-8

subtraction
adder/subtracter unit PP:6-4
base set ALU operations
class-independent PP:8-47
class-specific PP:8-48
carry-out functions PP:4-16
conditional PP:4-66, PP:8-80

subtraction (continued)
shift operations
See also shift operations
shift left PP:4-43
signed shift right PP:4-51
unsigned shift right PP:4-47
using PP:4-18, PP:4-39

T

TAG register PP:3-5
summary PP:1-16

task interrupt PP:9-6
command word PP:3-8, PP:3-11
detecting PP:3-14
setting the task interrupt flags PP:3-13

three-input ALU. *See* ALU, three-input

threshold
detecting values PP:4-57

transparency PP:4-60

U

unhalting
command word PP:3-8, PP:3-11
unhalting a PP PP:3-13

upper registers
base set ALU operands PP:8-35
register code PP:7-2

V

vector
addresses
interrupt priorities PP:9-13
interrupt vector locations PP:9-13

X

xba keyword PP:8-124

Z

zero-extend hardware PP:6-8
zero-overhead branching PP:5-13
zero-overhead looping PP:5-12 to PP:5-14, PP:11-2
program flow control unit PP:1-13
zero status bit PP:4-13, PP:7-3, PP:8-150