TMS320C80 (MVP) Master Processor User's Guide

SPRU109A
March 1995

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.

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 master processor (MP). The primary role of the MP is to coordinate the MVP's on-chip processing resources. The MP also communicates real-time with external devices and performs floating-point calculations. 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.

Notational Conventions

This document uses the following conventions.

Term/Convention	Description
CM	Composite mask in shift operations
d, DP	64-bit floating-point double-precision value
dest	Destination data register
FP	Floating point
i, I	32-bit signed integer precision
IP	Instruction pointer
link	Link return address in r31
LSB, MSB	Least significant bit, most significant bit
offset	Formed as (target – IP)/ 4 in branch or jump instructions
OR	Bitwise inclusive OR
QNaN	Quiet not-a-number in floating-point
register⟦bit⟧	Identifies a bit within a register. For example, IE⟦ie⟧ refers to the ie bit in the IE register.
S	Signed integer
SP	32-bit floating-point single-precision value
SNaN	Signaling not-a-number in floating point
source1	Data from source register 1
source2	Data from source register 2
StkPtr	Stack pointer in R1 (8-byte boundary)
target	Label of branching point
u, 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
( )	Separate an operand and offset
;	Marks beginning of a single-line comment
→	Means *becomes the contents of*. In an instruction execution description, for example, $SP \rightarrow PC$ the single-precision value becomes the contents of (or replaces the contents of) the program counter.
&, AND	Bitwise AND

Term/Convention	Description
~, 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: **br**[**.a**] *target* You can enter either the instruction **br** or **br.a**.
italic text	In instruction syntax, *italics* identify "placeholders" that identify the type of information that you should enter for a parameter. For example, **cmp** *source1,source2,dest* The cmp instruction has three parameters: *source1, source2,* and *dest*. You must replace *source1, source2,* and *dest* with actual source and destination registers (**cmp r7,r8,r9**)
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.
{ }	Indicate a list. The symbol **|** (read as *or*) separates items within the list. Here's an example of a list: {**s**|**d**} Unless the list is enclosed in square brackets, you must choose one item from the list.
| |	Parallel vector load/store instruction used with vector floating-point instructions (for example, vadd)
⋘ ⋙	Denote parallel vector operations in instruction syntaxes.

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) 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) Parallel Processor User's Guide (literature number SPRU110) describes the 'C80 parallel processor (PP). This manual provides information about the PP features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

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

Figures

Tables

Examples

Chapter 1

Overview of the Master Processor

The master processor (MP) is a 32-bit RISC (reduced instruction set computer) processor with a built-in IEEE-754 floating-point unit. The MP resides on the TMS320C80 MVP, a single-chip multiprocessor device with the power to perform complex video and audio applications.

The MP's primary role is to perform the general-purpose computations necessary to direct the MP's on-chip processing resources. The MP also performs floating-point calculations and provides real-time response to external events.

This chapter presents an overview of the master processor's features and capabilities.

Topics

1.1 The Master Processor (MP)

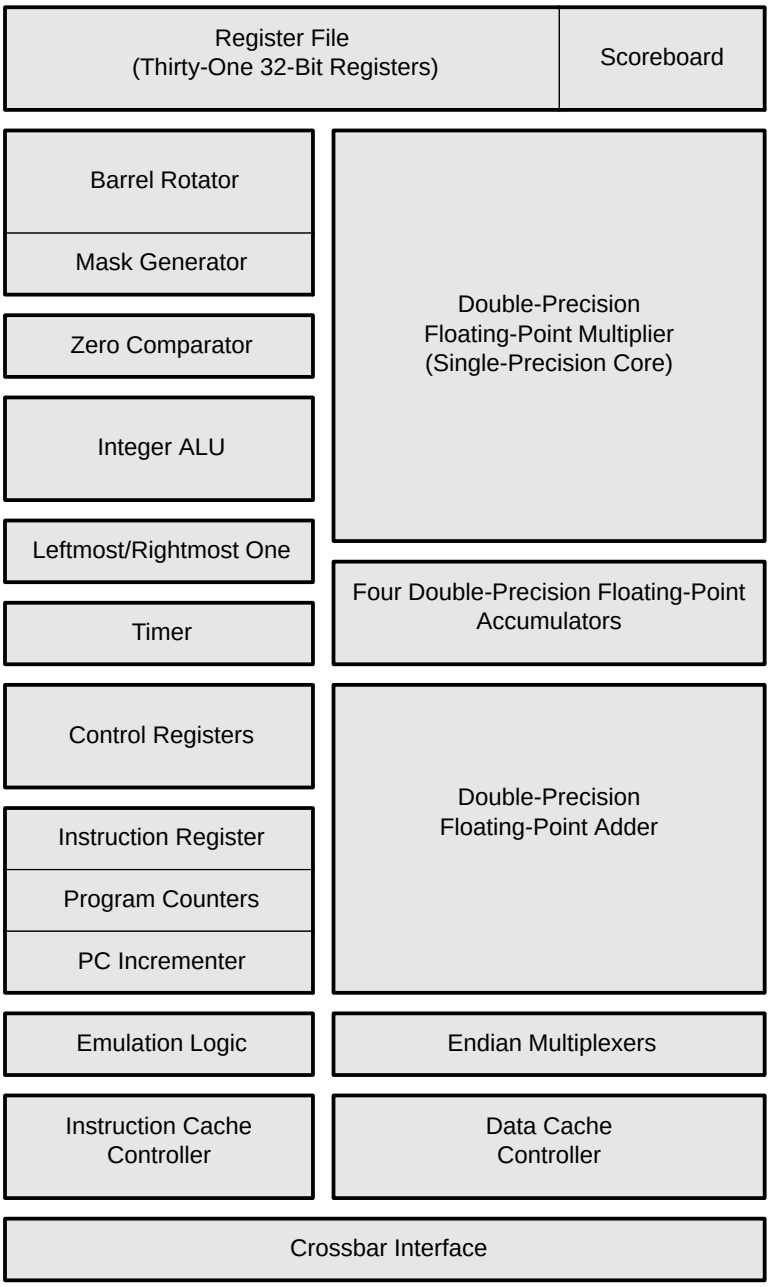
The MP is a 32-bit RISC processor with an integral IEEE-754 floating-point unit. As with other RISC processors, all accesses to memory are performed with load and store instructions, and most integer and logical operations on registers are performed in a single clock cycle. The MP has multiple-clock instructions to perform floating-point calculations and integer multiplications. The MP's floating-point hardware is pipelined to allow you to start a new single-precision multiply or any floating-point add instruction on each clock cycle. Floating-point unit operations use the same register file as the integer and logic unit. A register scoreboard ensures that correct register-access sequences are maintained.

Instructions and data are both fetched from on-chip cache memories. The MP's instruction and data caches are each 4K bytes in size. The control for these caches is an integral part of the MP design. The MVP's crossbar network allows the MP to access its two caches in parallel with other memory accesses.

The architecture of the MP is specifically designed for efficient execution of C code. As an example, the MP contains an r0 register, often called a zeroing register, used by C. Also, the MP instruction set is tailored to contain many of the primitives required for efficient C compilation.

Figure 1–1 is a block diagram of the major components internal to the MP.

Figure 1–1. Master Processor Block Diagram



1.2 Key Features

Key features of the MP that are shown in Figure 1–1 include:

- A 32-bit RISC processor
  - Load/store architecture
  - Three-operand arithmetic and logical instructions
- Thirty-one 32-bit general-purpose registers
- Four double-precision floating-point vector accumulators
- IEEE-754 floating-point hardware
- Data and instruction cache characteristics that include:
  - Four-way set associative
  - LRU replacement
  - Data write-back
  - No bus snooping or bus watching
- 4K-byte instruction cache
- 4K-byte data cache
- 2K-byte parameter RAM (not cached)
- Delayed branches with option to annul delay-slot instructions
- Explicit compare instructions—no dedicated status register
- Register and accumulator scoreboard
- 15-bit or 32-bit immediate constants
- Vector floating-point instructions
  - Initiate a floating-point operation and a parallel load or store in one instruction
  - Multiply and accumulate
- Scalable timer
- Leftmost-one and rightmost-one detection logic
- High performance. For example, at a processor clock frequency of N = 50 MHz:
  - $2 \times N$ MFLOPS peak = 100 MFLOPS
  - N MIPS = 50 MIPS
  - Over $2600 \times N$ Dhrystones = 130 000 Dhrystones
- Control registers used for vector data loads or stores, emulation, exceptions, and cache control
- 2^{32}-byte memory address space and 32-bit logical addresses that point to byte boundaries in memory

1.3 Interprocessor Communications

Communication between the MVP processors is implemented by the MP's cmnd (send command) instruction and a 32-bit command word. Interprocessor commands allow the MP to send interrupts to on-chip processors (including itself); reset, halt, or resume operation; or reset the instruction- or data-cache tags.

For more information about the 32-bit command word and interprocessor communications, see Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis*.

Chapter 2

Master Processor Registers

This chapter discusses the master processor registers. The MP has thirty-two general purpose registers; four double-precision floating-point accumulators and three I/O registers for vector loads and stores; and fifty-one control registers for functions such as checking the execution state of the instructions in the pipeline, managing data and instruction cache, and controlling system configuration.

The chapter concludes with an overview of the transfer controller and video controller on-chip registers that are mapped into the MP's memory map.

Topics

2.1 Reserved and Unnamed Bit Conventions

Future functions configured by bits that are currently reserved will be enabled by a value of 1. The following compatibility guidelines are suggested so that your code retains the same functionality on future versions of the MVP:

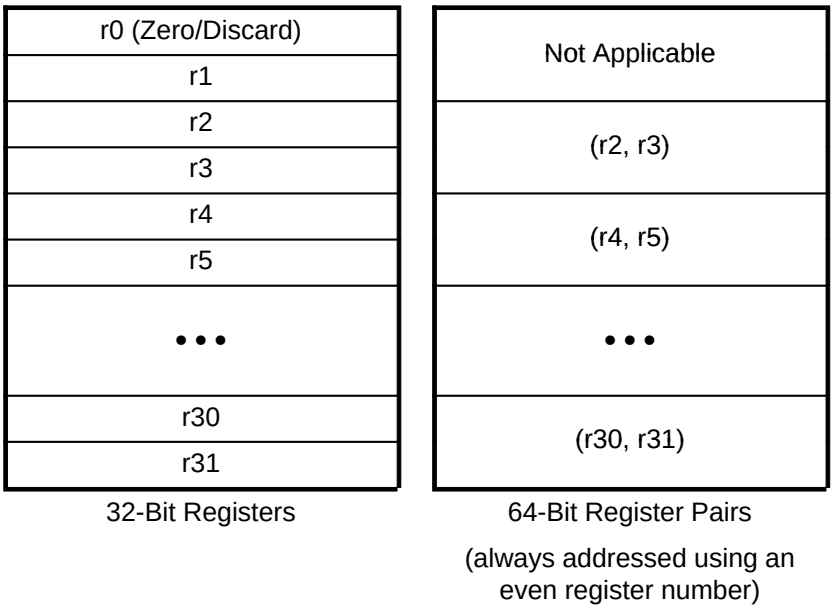
- During the MP's initialization sequence after reset, you should set reserved bits to 0. Note that at reset, the bits that are currently reserved are not set to any particular value.
- If it is necessary to write to any of the reserved bits in the process of setting currently implemented fields in a 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.

2.2 General-Purpose Registers

The MP contains thirty-one 32-bit general-purpose registers: r1 to r31 (see Figure 2–1). The thirty-second general-purpose register r0 is unique; reads from r0 always return the value zero, while writes to r0 do not affect r0's contents.

As indicated on the right side of Figure 2–1, each even/odd pair of 32-bit registers can be accessed as a single 64-bit register. For example, the MP's floating-point unit can write or read a double-precision value to or from register pair r2/r3 in a single clock cycle. Also, the MVP's doubleword-width internal memory bus allows 64-bit loads and stores of register pairs to complete in a single clock.

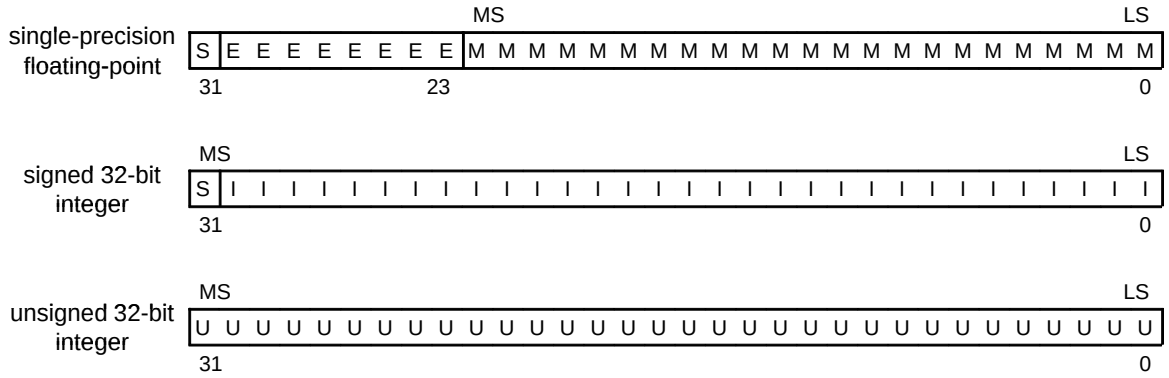
Figure 2–1. MP General-Purpose Registers



2.2.1 Individual Registers and Register Pairs

The 32-bit registers can contain signed-integer values, unsigned-integer values, single-precision floating-point values (see Figure 2–2), and double-precision floating-point values. The signed and unsigned bytes or halfwords are loaded into the 8 LSBs or 16 LSBs, respectively, of the 32-bit registers; the remaining register bits are filled either with zeros or by extending the sign bit of the value in the 8 or 16 lSBs (see Figure 2–3). Byte or halfword stores use the 8 LSBs or 16 LSBs, respectively, from the 32-bit register.

Figure 2–2. 32-Bit Data in MP Registers



Note: S = sign E = exponent
M = mantissa I = signed integer
U = unsigned integer

Figure 2–3. One Byte or Halfword in 32-Bit MP Registers

Note: S = sign I = signed integer
U = unsigned integer

Doublewords (64 bits) always use an even,odd register pair and are addressed using the even register. The odd register contains the 32 MSBs of the 64-bit doubleword from memory; the even register contains the 32 LSBs (see Figure 2–4).

Double-precision floating-point values are stored in an even-odd register pair (see Figure 2–5) and are referenced using the even register number. The odd register contains the sign bit, the exponent, and the most significant part of the mantissa. The even register contains the least significant part of the mantissa.

Notes:

1) Loading or storing double-precision floating-point numbers (using the ld.d or st.d instructions, respectively) is independent of big- or little-endian external memory organization.
2) r1 is an illegal destination of any operation that the floating-point unit executes.
3) For register r0 and r1 usage, see subsection 10.9.2, *Special Treatment of Registers r0 and r1*.

Figure 2–4. 64-Bit Data in MP Registers

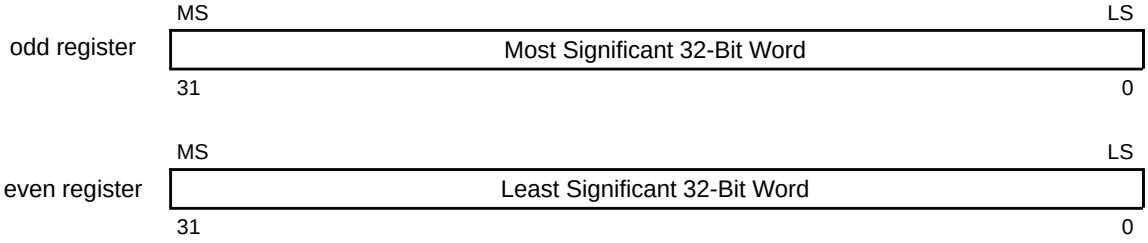
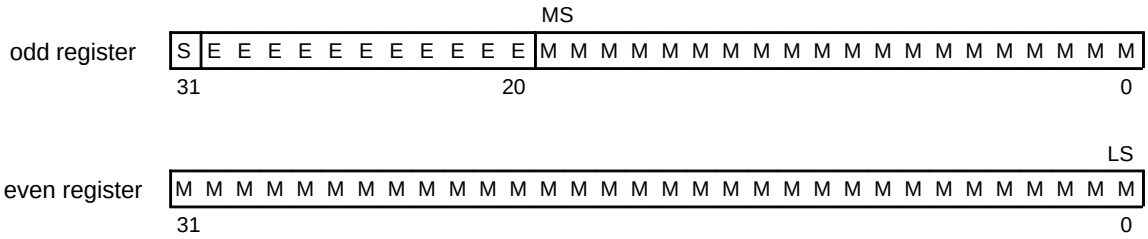


Figure 2–5. Double-Precision Floating-Point Data in MP Registers



Note: S = sign E = exponent
M = mantissa

2.2.2 Register Conventions

The following software conventions are recommended:

r0 Hardware zero

r1 Stack pointer on an 8-byte boundary (stack growing toward lower addresses)

r31 Return address link register

2.2.3 Register Scoreboarding

The scoreboard (SB) register indicates that a pipeline stall is required when either source register operands or the destination register is not available. Logically, the SB register has two flags for each register:

- ❑ The SB memory-load flag shows which registers are waiting on memory load completion.
- ❑ The SB floating-point-write flag shows which registers are waiting for the result of a previous floating-point unit instruction to be written.

The SB register has a scoreboard value for each register r1–r31; register r0 is not scoreboarded. The effective scoreboard bit for a register is given by the equation in Section C.4, *Freezing the Floating-Point Unit Pipeline*.

Reasons for a pipeline stall due to scoreboarding include:

- ❑ Operands are not loaded from data cache; this is indicated by an SB load flag (for example, awaiting service for a data cache-miss by the TC or a data read). Once the data is loaded or written into the destination register, the corresponding scoreboard flag is cleared, and the pipeline operation resumes.
- ❑ Operands are not written from a previous instruction that takes several cycles; this is indicated by an SB write flag (for example, all floating-point operations require multiple cycles to complete). The SB write flag allows the bits relating to floating-point operations to be temporarily removed from the scoreboard while the floating-point unit is stalled because of an exception. See Section C.4, *Freezing the Floating-Point Unit Pipeline*.

Note:

The scoreboard register (SB) is **not** a visible register.

2.3 Floating-Point Accumulators and Vector I/O Address Registers

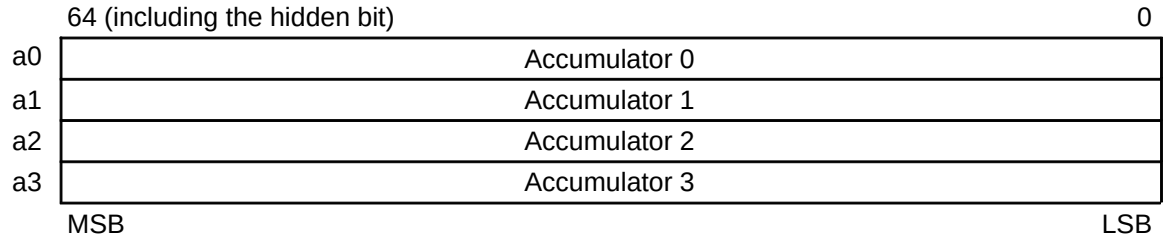
The MP's floating-point unit contains four double-precision floating-point registers (called **accumulators**) for intermediate floating-point results and three address registers to facilitate data I/O in parallel with vector floating-point operations.

2.3.1 Double-Precision Floating-Point Accumulators

Four double-precision floating-point registers are provided to accumulate intermediate floating-point results. These accumulator registers are named **a0, a1, a2,** and **a3**. All four accumulators are necessary to keep the floating-point pipeline in full use during calculations such as vector dot products or FFT butterflies. These registers improve efficiency of the floating-point pipeline by overlapping separate multiplies and adds for different computational elements. For example, all four output elements in a 1×4 vector that are used in three-dimensional 4×4 matrix multiply computation can be overlapped (for more information, see the example in Section 11.6, *Matrix Multiply Using Vector Floating-Point Instructions*). Accumulators a0, a1, a2, and a3 are shown in Figure 2–6.

See subsection 4.3.1, *Accumulator Scoreboarding*, for information about the scoreboarding activity for any one of the four accumulators.

Figure 2–6. Four Double-Precision Floating-Point Accumulators for the Floating-Point Unit



2.3.2 Vector I/O Address Registers

The vector I/O address registers (see Figure 2–7, *MP Control Registers*, or Table 2–1, *MP Control Register Numbers*) are used by the vector load (vld) or vector store (vst) instructions in parallel with other vector floating-point operations (for example, a vmac instruction, as shown in Example 2–1):

IN0P **Input address pointer 0** for vector load (vld0) of 32-bit or 64-bit words with IN0P postincremented by 4 or 8 bytes, respectively.

IN1P **Input address pointer 1** for vector load (vld1) of 32-bit or 64-bit words with IN1P postincremented by 4 or 8 bytes, respectively.

OUTP **Output address pointer** for vector store (vst) of 32-bit or 64-bit words with OUTP postincremented by 4 or 8 bytes, respectively.

Example 2–1. Vector Load/Store

```
vst.s    r3                          ; mem[OUTP++] = 32-bit r3

vld0.d   r4                          ; 64-bit (r4,r5) = mem [IN0P++]

vmac.ssd r2,r3,0,a1 || vst.d  r6     ; r2 × r3 + 0 → a1 and
                                     ;  mem[OUTP++] = 64-bit (r6,r7)

vadd.ss  r2,r3,r3 || vld1.s   r7     ; r2 + r3 → r3 and
                                     ;  32-bit r7 = mem[IN1P++]
```

2.4 Control Registers

Control registers represent the status of the processor (see Figure 2–7). These registers are divided into the following classes:

- FEA pipeline registers
- Packet request register
- State registers
- Cache registers
- Timing registers
- Configuration register
- System registers
- Memory fault registers
- Emulation registers

For a complete description of each of the control registers, see Chapter 3, *Master Processor Control Registers.*

You can address some of these registers directly (such as IN1P); other registers (for example, the program counter) are modified by hardware. Table 2–1 lists the numbers of the accessible registers that are required in branch (brcr), read (rdcr), swap (swcr), or write (wrcr) control register instructions.

Figure 2–7. MP Control Registers

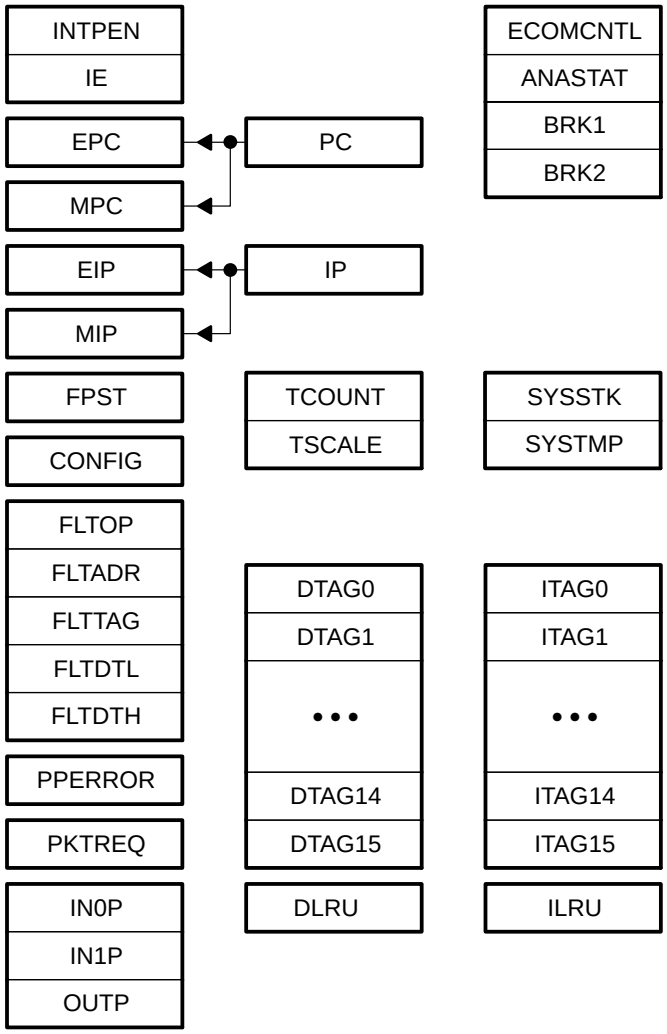


Table 2–1. MP Control Register Numbers

Register Number	Name	Description	See Page
0x0000	EPC	Exception program counter	MP:3-3
0x0001	EIP	Exception instruction pointer	MP:3-3
0x0002	CONFIG	Configuration register	MP:3-24
0x0004	INTPEN	Interrupts pending register (see note 2)	MP:3-7
0x0006	IE	Interrupts enabled register	MP:3-7
0x0008	FPST	Floating-point status register	MP:3-14
0x000A	PPERROR	PP error indicators	MP:3-18
0x000D	PKTREQ	Packet request register	MP:3-4
0x000E	TCOUNT	Current counter value	MP:3-23
0x000F	TSCALE	Counter reload value	MP:3-23
0x0010	FLTOP	Faulting operation register	MP:3-29
0x0011	FLTADR	Faulting address register	MP:3-32
0x0012	FLTTAG	Faulting tag register	MP:3-32
0x0013	FLTDTL	Faulting data low register (32 LSBs)	MP:3-32
0x0014	FLTDTH	Faulting data high register (32 MSBs)	MP:3-32
0x001X	FLTxxx	Reserved for fault	
0x0020	SYSSTK	Operating system stack pointer	MP:3-28
0x0021	SYSTMP	Operating system temporary register	MP:3-28
0x0030	MPC	Emulator exception program counter	MP:3-3
0x0031	MIP	Emulator exception instruction pointer	MP:3-3
0x0033	ECOMCNTL	Emulation analysis communication control register	MP:3-33
0x0034	ANASTAT	Emulation analysis status register	MP:3-33
0x0039	BRK1	Emulation analysis breakpoint 1 register	MP:3-33
0x003A	BRK2	Emulation analysis breakpoint 2 register	MP:3-33
0x020X	ITAG0–15	Instruction cache tags 0–15	MP:3-21
0x0300	ILRU	Instruction cache least recently used register	MP:3-21
0x040X	DTAG0–15	Data cache tags 0–15	MP:3-21
0x0500	DLRU	Data cache least recently used register	MP:3-21
0x4000	IN0P	Vector load address pointer 0	MP:2-8
0x4001	IN1P	Vector load address pointer 1	MP:2-8
0x4002	OUTP	Vector store address pointer	MP:2-8

Notes: 1) Register numbers below 0x4000 are read-only in MP user mode.

2) To clear a bit in the INTPEN register, use the wrcr instruction and a mask with a 1 in that bit position.

2.5 Transfer Controller and Video Controller On-Chip Registers

The MP controls the MVP's on-chip transfer controller (TC) and video controller (VC) through a number of user-accessible registers that are mapped into the MP's memory map. Section NO TAG, *Transfer Controller Registers*, in the *MVP Transfer Controller User's Guide* and Chapter NO TAG, *Frame Timer Registers*, in the *MVP Video Controller User's Guide* describe the TC and VC on-chip registers, respectively. However, excerpts from these sections are provided here a quick reference for several important registers. Registers highlighted in this section include:

- The TC's REFCNTL (refresh control) register, which controls DRAM refresh cycles.
- The TC's PTMIN (packet transfer minimum length) and PTMAX (packet transfer maximum length) registers, which control the duration of packet transfers before preemption is permitted.
- The TC's FLTSTS (memory fault status) register, which identifies the cause of a maskable memory fault.
- The VC's FMEMCTL0/1 (memory control) registers, which enable or disable VC frame timer-initiated packet transfer requests.

Note:

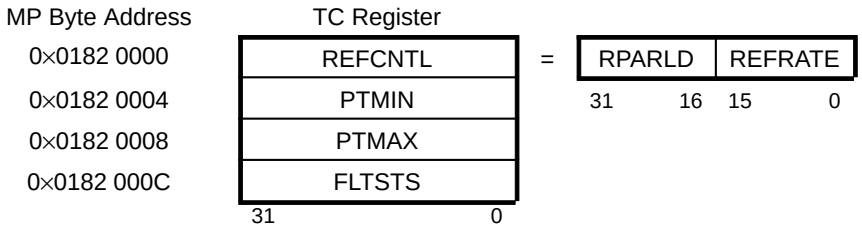
As a general rule, when reading or writing to a TC or VC register, you should specify the width of the access to match the width of the target register (16 or 32 bits). The following restrictions apply to load (ld and dld) and store (st and dst) instructions that access the MVP's on-chip memory-mapped registers:

- 32-bit stores are required to all 32-bit TC and VC registers.
- 16-bit stores are permitted only to 16-bit VC registers. Also, you can use 32-bit stores to access a pair of 16-bit VC registers.
- 64-bit loads are not supported from any TC or VC register. Likewise, 8-bit and 64-bit stores to any register are not supported.

2.5.1 Transfer Controller On-Chip Registers

The TC has four user-accessible registers that are mapped into the MP's memory map. You can access these with the load (ld) or store (st) instructions. The TC's on-chip registers are summarized in Figure 2–8 and in the following subsections.

Figure 2–8. Transfer Controller On-Chip Register Map



2.5.1.1 Refresh Control Register: REFCNTL

The REFCNTL register contains two 16-bit values that control the DRAM refresh cycles:

- **REFRATE**—bits 0–15 indicate the number of cycles between refresh requests. You can disable refresh by setting REFRATE to a value less than 0x0020. REFRATE contains the value 0x0020 at reset.
- **RPARLD**—bits 16–31 contain the reload value for the MVP's 16-bit refresh-address counter. The MVP's memory-control logic automatically decrements the counter by 1 each time it outputs a new refresh address to memory. When the counter reaches 0, the counter is reloaded with the RPARLD value. RPARLD contains the value 0xFFFF at reset.

2.5.1.2 Packet Transfer Minimum Length Register: PTMIN

In the PTMIN, bits 0–23 indicate the minimum number of clock cycles that a packet request must be serviced by the TC before it can be interrupted by a higher priority packet request. PTMIN contains the value 0x0001 0000 (64K cycles) at reset. Bits 24–31 are reserved.

2.5.1.3 Packet Transfer Maximum Length Register: PTMAX

Once a packet request has executed for the minimum number of clock cycles specified by PTMIN, PTMAX register bits 0–23 specify the maximum number of additional clock cycles for which the packet request can execute before it times out. Once PTMIN + PTMAX time has elapsed, the current packet request can be suspended for the execution of another packet request of the same priority. PTMAX does not affect when a transfer is suspended for a higher priority packet request. PTMAX contains the value 0x0001 0000 (64K cycles) at reset. Bits 24–31 are reserved.

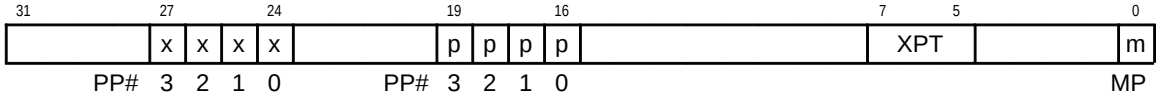
2.5.1.4 Fault Status Register: FLTSTS

The FLTSTS register contains status bits that indicate that a memory-access fault has occurred during an instruction cache fill, packet request, or direct external memory access (DEA). The interrupts generated by these faults are maskable; that is, software can enable or disable them. The FLTSTS register is shown in Figure 2–9.

When a packet request fault, PP instruction-cache fault, or PP load/store fault occurs, the FLTSTS register contains information about the origin of that fault. Additionally, setting any one of the FLTSTS bits signals the MP by setting the maskable-fault interrupt bit (INTPEN[mf] = 1). The interrupt service routine program can test the values in the FLTSTS register to determine the fault and initiate recovery attempts, if possible.

Clearing an FLTSTS packet transfer or cache-fill bit that has been set causes the associated transfer to be rescheduled. SImilarly, clearing the 3-bit XPT field allows the TC to begin servicing the current request at the MVP's XPT pins. You can clear FLTSTS bits by writing a 1 to the appropriate bit. Writing 0 to a bit has no effect.

Figure 2–9. Transfer Controller Fault Status Register—FLTSTS



Notes: 1) All unnamed bits are reserved. The following text describes each of the named bits.
2) A 1 indicates a fault.

- m (bit 0): Set if the last MP packet request had a fault. The TC suspends that packet request, saves its state in MP parameter RAM location 0x0101 0000, and leaves the MP linked-list pointer in 0x0101 00FC pointing to this suspended area. If the interrupt service routine resets this bit, the TC attempts this MP packet request again.
  - If m = 0, no memory fault has been caused by a packet transfer requested by the MP.
  - If m = 1, a packet transfer requested by the MP has caused a memory fault.
- XPT (bits 5–7): Indicates which externally initiated packet request (1–7) has faulted. The TC terminates that packet transfer immediately and ignores any new signals input on pins XPT[2:0] until the FLTSTS[XPT] field is cleared.

7	6	5	Faulted Request
1	1	1	External Packet Transfer 7 or VC Start of Field 0 Faulted
1	1	0	External Packet Transfer 6 or VC SAM Overflow 0 Faulted
1	0	1	External Packet Transfer 5 or VC Start of Field 1 Faulted
1	0	0	External Packet Transfer 4 or VC SAM Overflow 1 Faulted
0	1	1	External Packet Transfer 3 Faulted
0	1	0	External Packet Transfer 2 Faulted
0	0	1	External Packet Transfer 1 Faulted
0	0	0	No External Packet Transfer Fault

- p (bits 16–19): Set if that PP's last packet request had a fault. The TC suspends that packet request in PP parameter RAM location 0x0100 #000 with the PP linked-list pointer in 0x0100 #0FC pointing to this suspended area (where # is the PP number). If the interrupt service routine resets this p bit, the TC attempts this PP's packet request again.
  - bit 16 = PP0 packet transfer request fault
  - bit 17 = PP1 packet transfer request fault
  - bit 18 = PP2 packet transfer request fault
  - bit 19 = PP3 packet transfer request fault

  Bits 20–23 are reserved to indicate packet transfer faults in future MVP devices with eight PPs.

- x (bits 24–27): Set if that PP's last instruction-cache service or load/store instruction caused a fault. If the interrupt service routine resets the x bit, the TC attempts this PP's instruction-cache or load/store request again.
  - bit 24 = PP0 instruction-cache or load/store fault
  - bit 25 = PP1 instruction-cache or load/store fault
  - bit 26 = PP2 instruction-cache or load/store fault
  - bit 27 = PP3 instruction-cache or load/store fault

  Bits 28–31 are reserved to indicate PP instruction-cache or load/store faults in future MVP devices with eight PPs.

Note:

The instruction-cache or load/store faulting address is stored in location 0x0100 #0F8 of that PP's parameter RAM. The interrupt service routine must examine PPERROR bits 0–3, shown in Figure 3–4, *MP Register Showing PP Errors—PPERROR (Read Only)*, to determine if that PP's fault is instruction-cache fault or load/store fault:

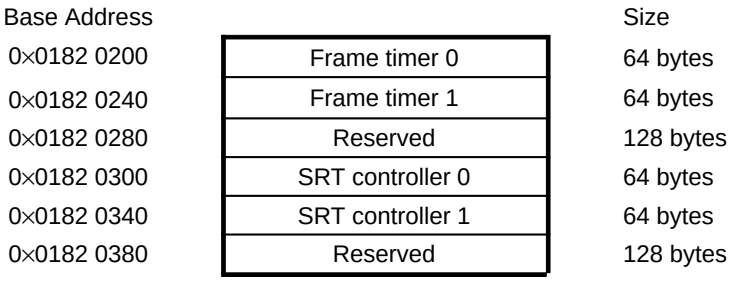
- 0 = instruction-cache fault
- 1 = load/store fault

2.5.2 Video Controller On-Chip Registers

The video controller (VC) has fourteen 32-bit and forty-two 16-bit registers that are mapped into the MP's memory address space. These registers control the frame timers, interlace mode, vertical and horizontal sync modes, and horizontal blanking and serration. You can access these registers by using the load (ld) or store (st) instructions.

The VC's on-chip registers are summarized in Figure 2–10. Each block in the figure represents a group of related registers.

Figure 2–10. VC On-Chip Register Map



Note: See Figure 2–11 and Figure 2–12 for frame timer and SRT information, respectively.

The VC frame timers are illustrated in Figure 2–11. There are 21 registers in each timer.

Figure 2–11. VC Frame Timers 0 and 1 Register Map

Address	Timer 0	Address	Timer 1
0×0182 0200	FTCTL0	0×0182 0240	FTCTL1
0×0182 0204	SETHCT0	0×0182 0244	SETHCT1
0×0182 0206	SETVCT0	0×0182 0246	SETVCT1
0×0182 0208	HESERR0	0×0182 0248	HESERR1
0×0182 020A	VFTINT0	0×0182 024A	VFTINT1
0×0182 020C	HESYNC0	0×0182 024C	HESYNC1
0×0182 020E	VESYNC0	0×0182 024E	VESYNC1
0×0182 0210	HEBLNK0	0×0182 0250	HEBLNK1
0×0182 0212	VEBLNK0	0×0182 0252	VEBLNK1
0×0182 0214	HSAREA0	0×0182 0254	HSAREA1
0×0182 0216	VSAREA0	0×0182 0256	VSAREA1
0×0182 0218	HEAREA0	0×0182 0258	HEAREA1
0×0182 021A	VEAREA0	0×0182 025A	VEAREA1
0×0182 021C	HSBLNK0	0×0182 025C	HSBLNK1
0×0182 021E	VSBLNK0	0×0182 025E	VSBLNK1
0×0182 0220	HTOTAL0	0×0182 0260	HTOTAL1
0×0182 0222	VTOTAL0	0×0182 0262	VTOTAL1
0x0182 0224	HALINE0	0x0182 0264	HALINE1
0x0182 0228	HBLINE0	0x0182 0268	HBLINE1
0×0182 023C	HCOUNT0	0×0182 027C	HCOUNT1
0×0182 023E	VCOUNT0	0×0182 027E	VCOUNT1
	15 0		15 0

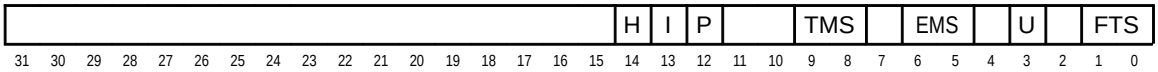
The VC SRT (serial register transfer) controllers are shown in Figure 2–12. There are 7 registers in each controller.

Figure 2–12. VC SRT Controllers 0 and 1 Register Map

Address	SRT Controller 0	Address	SRT Controller 1
0×01820300	FMEMCTL0	0×0182 0340	FMEMCTL1
0×0182 0304	F1STADR0	0×0182 0344	F1STADR1
0×0182 0308	F0STADR0	0×0182 0348	F0STADR1
0×0182 030C	LINEINC0	0×0182 034C	LINEINC1
0×0182 0310	SAMMASK0	0×0182 0350	SAMMASK1
0×0182 0314	NEXTADR0	0×0182 0354	NEXTADR1
0×0182 033C	CRNTADR0	0×0182 037C	CRNTADR1
	31 0		31 0

Figure 2–13 depicts the frame memory control register (FMEMCTL) in each set of SRT control registers. For more information about the FMEMCTL register and its bits, see the FMEMCTL0/1 discussion on page NO TAG of the *MVP Video Controller User's Guide.*

Figure 2–13. Video Controller Memory Control Register—FMEMCTL0/1



Note: All unnamed bits are reserved.

- FTS (bits 0–1)—Frame timer sequencer: Indicates which frame timer will generate the frame memory's serial register transfers (SRTs).

- U (bit 3)—Unblanked event disable: Controls whether the blanking or area registers of a frame timer generate the frame timer events.

- EMS (bits 5–6)—Event mode select: Determines which events will generate packet requests to the SRT. These events include:

  - Half-length serial access memories (SAMs) overflow events
  - Start of field (sof) events

  See Section NO TAG, SRT *Events*, and NO TAG, SRT *Event Modes (EMS),* in the *MVP Video Controller User's Guide* for more information about the EMS field.

- TMS (bits 8–9)—Transfer mode select: Determines how data will be transferred between the VRAM memory array and its serial access memory (SAM).

- P (bit 12)—Packet transfer select (PTS):
  - P = 0 selects the normal VC's serial register transfers
  - P = 1 enables the VC to initiate frame timer packet transfer requests when selected events occur in that frame timer. This can only be done if:
    - CONFIG[X] = 0 (see Figure 3–7, *MP Configuration Register—CONFIG*).
    - CONFIG[X] = 1 and XPT4–7 are not requested.
    - The program has set the correct linked-list values in the four MP parameter RAM locations 0x0101 00E0 to 0x101 00EC (see Figure 5–3, *Externally Initiated Packet Transfer Linked-List Address Pointers*) for externally or video controller frame timer-initiated packet requests.
    - The four packet transfer parameter lists that are enabled by the video controller frame timer are initialized.

  The P bit is not available on preproduction silicon version 1.

- I (bit 13)—Interlaced line repeat (ILR): When enabled, causes each horizontal line to occur once per field rather than once per frame.
- H (bit 14)—Half length serial access memory (SAM) select: Modifies the way in which serial register transfer addresses are generated to provide the proper tap point for half length SAM type devices.

Note:

The program must ensure that the four video controller-initiated packet requests and externally initiated XPT4–7 are mutually exclusive; that is, XPT7 from some other external device and video controller start-of-field 0 cannot both be active at the same time, and so on (see Figure 5–3 for other combinations).

Chapter 3

Master Processor Control Registers

This chapter describes the master processor's control registers. Control registers are used to monitor and control the status of the MP, PPs, and other on-chip mechanisms. These registers can be accessed only by the MP. Instead of being memory-mapped, the control registers reside in their own address space. Only programs running in supervisor mode can write to control register numbers below 0x4000. Section 2.4, *Control Registers,* provides an overview of the MP's control registers and lists the control register numbers.

For a description of the other MP registers, including the vector I/O address control registers (IN0P, IN1P, and OUTP), see Chapter 2, *Master Processor Registers.*

Topics

3.1 FEA Pipeline Registers

The master processor is implemented as a three-stage, FEA (fetch-execute-access) pipelined machine. The pipeline can overlap different stages of execution of three instructions in the same clock cycle. The FEA pipeline consists of three stages in the following order:

- **F**etch the next instruction, whose address is in the program counter (PC).
- **E**xecute the instruction, whose address is in the instruction pointer (IP).
- **A**ccess the external memory or on-chip SRAM for data (if needed).

The PC and IP registers are not user-visible. When an exception (interrupt or trap) occurs, the contents of the PC and IP registers are copied into the EPC and EIP registers, from which they can be accessed by software. The information in these registers is needed to return to the interrupted program after the exception has been serviced. Similarly, when an emulation trap occurs, the contents of PC and IP are copied into the MPC and MIP registers, from which they can be accessed by the emulation software. These registers are summarized in Table 3–1.

Table 3–1. MP FEA Pipeline Registers

	Program Execution Mode		
	Normal	**Exception**	**Emulation**
Program Counter	PC	EPC	MPC
Instruction Pointer	IP	EIP	MIP
Instruction Register	IR		

3.1.1 Program Execution Pipeline Registers

Registers in the normal program sequence flow are shown in Table 3–1 and can be altered by branch or trap instructions. The normal execution pipeline registers are the following:

- **Program counter** (**PC**) contains the address of the instruction being fetched.
- **Instruction pointer** (**IP**) contains the address of the instruction being executed.
- **Instruction register** (**IR**) contains the actual instruction being executed where IR is the instruction from memory location IP.

3.1.2 Program Exception Pipeline Registers: EPC and EIP

Normal program flow can be interrupted by an exception (for example, integer overflow). If an exception occurs, the normal state of the pipeline registers (PC and IP) are saved in the exception context registers (EPC and EIP, respectively). While the exception is being serviced by an interrupt service routine, the PC, IP, and IR are used as in the normal program flow. The exception pipeline registers in Table 3–1 are the following:

- **Exception instruction pointer** (**EIP**) contains the address of the instruction that would have been executed if the exception had not occurred. This is the first instruction to be fetched after returning from an interrupt.

- **Exception program counter** (**EPC**) contains the address of the instruction that would have been fetched if the exception had not occurred. This is the second instruction to be fetched after returning from an interrupt and the first instruction to be fetched after returning from a trap.

3.1.3 Program Emulation Pipeline Registers: MPC and MIP

There are dedicated context registers similar to EIP and EPC called MIP and MPC, respectively. These registers are used in the same manner as EIP and EPC (for example, to save the IP and PC registers at the time of emulation exceptions). Emulation exceptions are caused by emulation trap (etrap) instructions or analysis events (for example, an address breakpoint) generated by a simulator or debugger. While the emulation exception is being serviced by an interrupt service routine, the PC, IP, and IR are used as in the normal program flow. The emulation pipeline registers shown in Table 3–1 are the following:

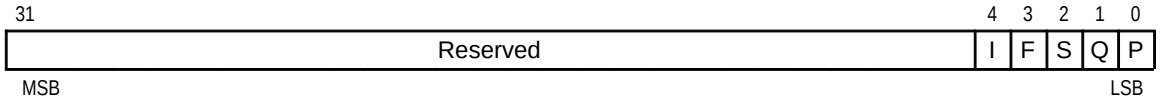
- **Emulation instruction pointer** (**MIP**) contains the address of the instruction that would have been executed if the emulation exception had not occurred. This is the first instruction to be fetched after returning from an interrupt.

- **Emulation program counter** (**MPC**) contains the address of the instruction that would have been fetched if the emulation exception had not occurred. This is the second instruction to be fetched after returning from an interrupt and the first instruction to be fetched after returning from a trap.

3.2 Packet Request Register: PKTREQ

You can use the **packet request register** (**PKTREQ**) to issue a packet request to the transfer controller (TC) from the MP or to check the status of a previous packet request. Refer to Figure 3–1. Note that bits are read/write unless otherwise noted.

Figure 3–1. MP Packet Request Register—PKTREQ



- P (bit 0): If PKTREQ[Q] = 0 and PKTREQ[S] = 0 (or PKTREQ[S] is simultaneously cleared to 0 while P is being set), writing a 1 to the P bit submits a packet request to the TC (see Section 7.4, *Packet Transfer Handshake Signals*, for more information). Before submitting a request, the program should write the address of the packet request to the linked-list pointer in the MP parameter RAM at location 0x0101 00FC.
- Q (bit 1): Packet request queue is active (read-only bit). This bit indicates when the TC is still busy performing a packet transfer that was requested previously by the MP.
- S (bit 2): Suspend packet request.
- F (bit 3): Foreground priority is selected.
- I (bit 4): Immediate (urgent) priority is selected.

The use of packet requests is discussed in Chapter 7, *Understanding the Packet Transfer Request Protocol*.

Table 3–2 shows I, F, S, and P bit values for submitting and suspending packet transfer requests.

Table 3–2. Setting the PKTREQ Bits

	Set These Bits			
To...	**I**	**F**	**S**	**P**
Submit various priority packet requests:				
❑ Background (low priority)	0	0	0	1
❑ Foreground (high priority)	0	1	0	1
❑ Immediate (urgent priority)	1	X†	0	1
Change packet request priorities:				
❑ Set to background (low priority)	0	0	0	0
❑ Set to foreground (high priority)	0	1	0	0
❑ Set to immediate (urgent priority)	1	X†	0	0
Suspend packet requests:				
❑ Suspend at the existing priority	‡	‡	1	0
❑ Set to low priority and suspend	0	0	1	0
❑ Set to high priority and suspend	0	1	1	0
❑ Set to urgent priority and suspend	1	X†	1	0

†X = 0 for production silicon version 3 and earlier; X = 1 is reserved for future MVP products.
‡Rewrite the bit with its current value.

3.3 State Registers

State registers reflect the execution state of the instructions in the pipeline; they control the flow of instructions down the pipeline. This includes enabling interrupts and flag bits that show which interrupts are pending. The MP's state registers are listed in Table 3–3 and are described in the following subsections.

Table 3–3. MP State Registers

Register	See page
Scoreboard register: SB	MP:3-6
MP interrupt registers: IE and INTPEN	MP:3-7
Floating-point status register: FPST	MP:3-14
PP error register: PPERROR	MP:3-18

3.3.1 Scoreboard Register: SB

The **scoreboard** (**SB**) register indicates which registers are waiting on memory load completion, or which registers are waiting for the result of a previous floating-point unit instruction to be written. SB is not a visible control register. Note that register R0 is not scoreboarded. For more information about the SB register, see subsection 2.2.3, *Register Scoreboarding.*

3.3.2 MP Interrupt Registers: IE and INTPEN

You can use the MP's interrupt enable (IE) and interrupt pending (INTPEN) registers to control and monitor interrupts.

- The **IE** register contains interrupt enable bits for the interrupts that are maskable. For example, you can enable interrupts that detect errors, synchronize multiprocessor messages, or signal the program when I/O or computational events have completed.

  - If an enable bit is 1, program control transfers to the address specified in the transfer vector array should that specific interrupt occur.
  - If an enable bit is 0, the interrupt transfer is not performed.

  The exception to this is that not all memory faults are maskable. Pending nonmaskable faults are not reflected in the INTPEN register and are always enabled, regardless of the contents of the IE register. Both maskable and nonmaskable faults, however, share the interrupt vector at address 0x0101 01B8 (see subsection 3.3.2.6).

  IE's LSB is the highest priority interrupt, and IE's MSB is the lowest priority interrupt.

  Note that interrupts must be enabled (IE⟦ie⟧ = 1) to service interrupts and to perform any floating-point unit operations. See Figure 3–2.

  You can modify the IE register with the wrcr or swcr instruction in the supervisor mode.

- The **INTPEN** register contains bits that are set by hardware to indicate interrupts awaiting service (see Figure 3–2). Interrupts are serviced if interrupts are enabled globally (IE⟦ie⟧ = 1) **and** the bit that corresponds to the interrupt is also a 1 in register IE. Otherwise, the program must poll the INTPEN register to determine if interrupt service is required.

  Bits in the INTPEN register can be reset using the wrcr instruction when the MP is in the supervisor mode.

  - Writing a 0 to an INTPEN bit that is a 0 has no effect.
  - Writing a 1 to an INTPEN bit that is a 0 has no effect.
  - Writing a 1 to an INTPEN bit that is a 1 resets that bit and turns off the interrupt service request corresponding to that bit.

INTPEN register bits are latched. Once an INTPEN bit has been set to 1 to indicate a pending interrupt, the bit remains 1 until it the software clears it to 0 by writing a 1 to that bit. The one exception to this rule is the INTPEN[x4] bit, which is a read-only bit that is updated continuously to follow changes in the signal input to the MVP's $\overline{\text{LINT4}}$ pin.

Note that you cannot change an INTPEN bit from 0 to 1 by writing to INTPEN.

Maskable interrupt *N* is serviced when:

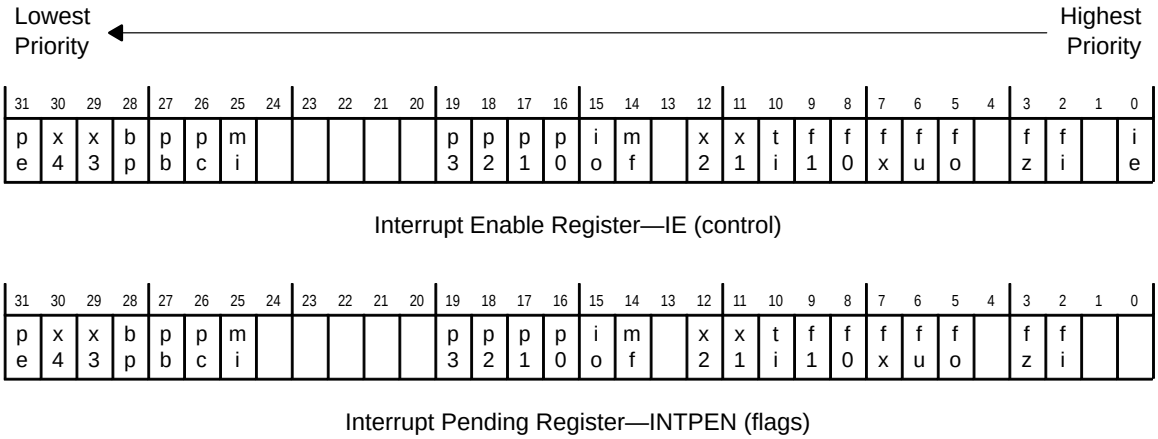
- Interrupts are globally enabled (IE[ie] = 1).
- *N* is the highest priority interrupt that is both enabled in the IE register and pending in INTPEN.
- The address of interrupt *N*'s service routine is stored in the MP's interrupt vector address 0x0101 0180 + 4**N* (pre-initialized by software).

When all of these conditions are true, the interrupt *N* service routine is executed.

For example, assume that the floating-point divide-by-zero exception is enabled (IE[ie] = 1 and IE[fz] = 1; note that fz is bit *N*=3 of the IE register). A divide-by-zero exception causes the PC to be loaded with the vector address contained in location 0x0101 0180 + 4**N* = 0x0101 018C, where *N* = 3. This is the same vector address that is loaded into the PC by a trap-3 instruction. A trap-3 instruction, however, always causes a trap to occur, regardless of the state of IE's ie and fz bits. Interrupt vector addresses are further described in Table 9–1, *Maskable Interrupt Priorities and Vector Addresses*.

Bits in the IE and INTPEN registers are shown in Figure 3–2 and are described in the following subsections.

Figure 3–2. MP Interrupt Registers—IE and INTPEN



Note: All unnamed bits are reserved.

Legend:

ie	global interrupt enable	fi	floating-point invalid
fz	floating-point divide-by-zero	fo	floating-point overflow
fu	floating-point underflow	fx	floating-point inexact
f0	frame timer 0	f1	frame timer 1
ti	MP timer	x1	external interrupt 1
x2	external interrupt 2	mf	memory fault
io	integer overflow	p0	PP0 message
p1	PP1 message	p2	PP2 message
p3	PP3 message	mi	MP self-interrupt
pc	packet complete	pb	packet busy
bp	bad packet	x3	external interrupt 3
x4	external interrupt 4	pe	PP error

3.3.2.1 Global Interrupt Enable

Interrupts (and floating-point operations) are globally enabled by the IE register's bit 0 (ie).

- ie = 1 enables interrupts.
- ie = 0 disables interrupts.

Bits 1–31 of the IE register enable individual interrupts. An interrupt is enabled only if its individual enable bit and the global-enable bit (ie) are both set in the IE register. Note that INTPEN bit 0 is not used to post interrupt 0.

3.3.2.2 Floating-Point Pipeline Exceptions

The floating-point exceptions are posted in INTPEN bits 2, 3, 5, 6, and 7 if the corresponding bits are enabled in register IE. The following floating-point exceptions are monitored by INTPEN:

- fi (bit 2): Floating-point invalid
- fz (bit 3): Floating-point division by zero
- fo (bit 5): Floating-point overflow
- fu (bit 6): Floating-point underflow
- fx (bit 7): Floating-point inexact

See Section C.5, *Results From Floating-Point Unit Exceptions*, for more information about floating-point exceptions.

3.3.2.3 Frame Timer Interrupts

The two frame timer interrupts are used to synchronize programs to specified scan lines on display devices controlled by the VC. INTPEN bits 8 and 9 are dedicated to frame timer interrupts from VC frame timers 0 and 1, respectively. A frame timer generates an interrupt when its VCOUNT counter reaches the value contained in its VFTINT register. For more information, refer to the VFTINT0/1 discussion on page NO TAG and Section NO TAG, *Programming VSAREA, VEAREA, and VFTINT in Interlaced Mode,* in the *MVP Video Controller User's Guide.*

- f0 (bit 8): Frame timer 0 interrupt when VCOUNT0 = VFTINT0
- f1 (bit 9): Frame timer 1 interrupt when VCOUNT1 = VFTINT1

3.3.2.4 MP Timer Interrupt

The MP has a timer that can signal an interrupt to the MP when the 32-bit counter decrements by 1 until it reaches 0 (counts processor clock cycles). This is bit 10 (ti) in the INTPEN register. An MP timer interrupt is posted when TCOUNT (decrements by 1 on each clock) reaches 0 and is 0 for one clock cycle. TCOUNT is then reloaded with the TSCALE value. See Section 3.5, *Timing Registers*.

3.3.2.5 External Interrupts

Four bits in the INTPEN register are dedicated to monitoring interrupt requests from devices external to the MVP. These interrupts are referred to as external interrupts 1, 2, 3, and 4. An external device requests an interrupt by signaling one the MVP's four interrupt-request pins ($\overline{\text{EINT1}}$, $\overline{\text{EINT2}}$, $\overline{\text{EINT3}}$, and $\overline{\text{LINT4}}$). See Section 9.5, *Reset*, for more information about how the MVP runs or halts at reset using $\overline{\text{EINT3}}$.

- x1 (bit 11): External interrupt 1 is signaled
- x2 (bit 12): External interrupt 2 is signaled
- x3 (bit 29): External interrupt 3 is signaled
- x4 (bit 30): External interrupt 4 is signaled

3.3.2.6 Memory Fault Interrupts

MVP memory faults fall into two categories:

- Nonmaskable faults caused by the MP
- Maskable faults caused by the TC and PPs

Maskable memory faults are indicated by INTPEN bit 14; nonmaskable memory faults have no effect on this bit. Nonmaskable memory faults always take interrupt 14 service, even if IE⟦ie⟧ = 0 or IE⟦mf⟧ = 0. Status information for maskable and nonmaskable faults is contained in the FLTSTS and FLTOP registers, respectively. FLTSTS is one of the TC's memory-mapped registers; FLTOP is an MP control register. For more information about the maskable and nonmaskable operation of memory faults, refer to Section 5.6, *Accessing Illegal Addresses*.

3.3.2.7 Integer Overflow Interrupts

Integer overflows for addition/subtraction instructions are signaled in bit 15 (io) of the INTPEN register.

Preproduction silicon versions 1 and 2 have interrupt io as bit 4 (not bit 15) in both IE and INTPEN registers.

3.3.2.8 PP Message Interrupts

Each PP can send a message interrupt to the MP. INTPEN bits 16–19 are dedicated to message interrupts from PPs 0–3. INTPEN bits 20–23 are reserved to accommodate future MVP devices with up to eight PPs on a chip. A message interrupt typically indicates that a PP has completed the assigned task previously requested by the MP.

- p0 (bit 16): Message interrupt from PP0 to the MP
- p1 (bit 17): Message interrupt from PP1 to the MP
- p2 (bit 18): Message interrupt from PP2 to the MP
- p3 (bit 19): Message interrupt from PP3 to the MP
- bits 20–23: Reserved for future MVP configurations for PP4–PP7 message interrupts to the MP

3.3.2.9 MP Message Interrupts

With the cmnd instruction, the MP can send a message interrupt to itself (a self interrupt). This interrupt is posted in INTPEN bit 25 (mi).

3.3.2.10 Packet-Transfer Status Interrupts

MP packet request information is signaled in INTPEN bits 26–28.

- pc (bit 26)—Packet transfer completed: The last MP packet transfer request has been completed by the TC (see bit 31 in the PT Options field described in Section NO TAG, *The PT Options Field*, in the *MVP Transfer Controller User's Guide*). Also, if the PT Options bit 28 = 1, an intermediate packet transfer request sets pc = 1 when that request is complete.
- pb (bit 27)—Packet transfer queued: The TC is busy with an earlier MP packet transfer request, and the new transfer cannot begin. This MP transfer request can be resubmitted by the MP once the TC completes the previous packet transfer request.
- bp (bit 28)—Packet transfer error: The TC cannot service this new MP packet transfer request; a parameter error has been detected in the packet transfer list. Examples include:
  - The number of source bytes is less than the number of destination bytes.
  - The packet transfer parameter list is not in on-chip SRAM.
  - The packet transfer parameters do not begin on a 64-byte boundary.

For a complete list of packet transfer errors, see Section NO TAG, *Packet Transfer Errors*, in the *MVP Transfer Controller User's Guide*.

3.3.2.11 PP Illegal Instruction Interrupt

The PP illegal instruction interrupt is posted in INTPEN bit 31. The PP number is set in one of PPERROR bits 8–11 (see Figure 3–4). The location of the PP's illegal instruction is contained in that PP's ipa register. Location 0x0100 #7F8 contains this ipa value after the PP is reset (where # = PP number 0, 1, 2, or 3).

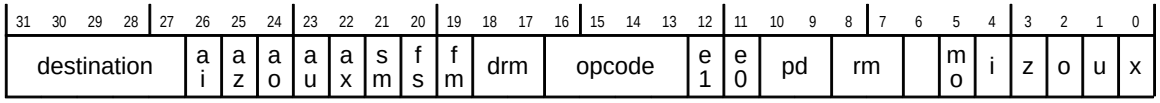
3.3.3 Floating-Point Status Register: FPST

The **floating-point status** (**FPST**) register contains bits that are set by the floating-point unit hardware to report the results of every floating-point unit instruction and to reflect floating-point status or errors since FPST was last reset. A floating-point instruction is not considered complete until FPST has been updated. Note that each floating-point instruction writes to the FPST register once and only once.

The bits in the FPST register are shown in Figure 3–3 and are described in the following subsections.

At reset, all bits in the FPST register that are not shown as read-only are cleared to 0, except the fm (fast mode) bit; the fm bit is set to 1 upon reset.

Figure 3–3. MP Floating-Point Status Register—FPST



Note: All unnamed bits are reserved.

Floating-point exceptions are discussed in Appendix C, *Floating-Point Unit Exceptions.*

Note:

Your program cannot write to FPST (or any other control register) while the floating-point unit is not empty.

3.3.3.1 Current Floating-Point Status

Bits 0–16 and 27–31 of the FPST report the final status of the last floating-point instruction. The current-status fields are defined as follows:

- x (bit 0): Floating-point inexact; this read-only bit is written at the completion of each floating-point instruction.
- u (bit 1): Floating-point underflow; this read-only bit is written at the completion of each floating-point instruction.
- o (bit 2): Floating-point overflow; this read-only bit is written at the completion of each floating-point instruction.
- z (bit 3): Floating-point division by zero; this read-only bit is written at the completion of each floating-point instruction.
- i (bit 4): Floating-point invalid; this read-only bit is written at the completion of each floating-point instruction.

  Refer to subsection 8.1.3, *Floating-Point Status Register: FPST*, for more information about the x, u, o, z, and i floating-point status bits.
- mo (bit 5): Floating-point integer multiply overflowed 32 LSBs; read-only bit
- rm (bits 7–8): Rounding mode of the last instruction to write to the FPST register; read-only bits

Bits		
8	**7**	**Rounding Mode**
0	0	Round to nearest
0	1	Round to zero
1	0	Round toward positive infinity
1	1	Round toward negative infinity

- pd (bits 9–10): Destination precision of the last instruction to write to the FPST register; read-only bits

Bits		
10	**9**	**Destination Precision**
0	0	Single-precision floating point
0	1	Double-precision floating point
1	0	Signed integer
1	1	Unsigned integer

- e0 (bit 11): The ninth MSB of the 11-bit exponent of the last instruction to write to the FPST register; this read-only bit can be set only for double-precision rounds when either the overflow or underflow trap is enabled.

- e1 (bit 12): The tenth MSB of the 11-bit exponent of the last instruction to write to the FPST register; this read-only bit can be set only for double-precision rounds when either the overflow or underflow trap is enabled.
- opcode (bits 13–16): Four opcode bits of the last instruction to write to the FPST register; this read-only bit field is the opcode field from the floating-point instruction, where instruction bits 13–15 are copied directly into FPST's bits 13–15, and instruction bit 17 is inverted and copied into FPST's bit 16.
- destination (bits 27–31): Five-bit destination register address of the last instruction to write to the FPST register; read-only bit field (destination = 0 = r0 when results go to accumulators).

3.3.3.2 Floating-Point Control

You can control operation of the floating-point unit by writing values to bits 17–21 of the FPST register. The control fields are defined as follows:

- drm (bits 17–18): Selects the default rounding mode if the rounding mode is not specified in the individual floating-point instruction.

Bits		
18	**17**	**Rounding Mode**
0	0	Round to nearest
0	1	Round to zero
1	0	Round toward positive infinity
1	1	Round toward negative infinity

- fm (bit 19): Determines how denormals are handled in the floating-point unit.
  - When fm = 0, fast mode is enabled, where denormals are forced to zero.
  - When fm = 1, IEEE mode is enabled, where denormals are returned as specified in IEEE-754.
- sm (bit 21): Forces the floating-point unit to run in the sequential mode so that there is at most one floating-point instruction in the floating-point pipeline. For example, a second floating-point operation stalls the FEA pipeline waiting for completion of the first floating-point operation in the sequential mode.
  - 0 = floating-point unit runs in pipeline mode
  - 1 = floating-point unit runs in sequential mode

3.3.3.3 Floating-Point Stall

When the floating-point unit encounters an enabled floating-point exception, the floating-point unit stalls and sets the fs bit (bit 20). This permits a service routine to examine the exception and attempt corrective action.

3.3.3.4 Floating-Point Accumulated Status

The floating-point accumulated-status bits in bits 22–26 of the FPST register keep a cumulative record of exceptional conditions that are indicated by the current-status flags in bits 0–4 of FPST. For example, ax is the accumulated version of the x status bit, au is the accumulated version of the u status bit, and so on. Whereas current-status bits 0–4 are continually updated to reflect the status of the last floating-point operation, accumulated-status bits 22–26 keep a persistent record of any exceptional conditions that have occurred since the last time they were explicitly cleared by the software. Each accumulated flag represents the logical-OR of the past values of the corresponding current-status flag. Once a current-status flag takes on a nonzero value, the corresponding accumulated-status bit is set to 1 and remains 1 until explicitly reset. For example:

new ax = old ax OR current x

- ax (bit 22): Accumulated floating-point inexact
- au (bit 23): Accumulated floating-point underflow
- ao (bit 24): Accumulated floating-point overflow
- az (bit 25): Accumulated floating-point division by zero
- ai (bit 26): Accumulated floating-point invalid

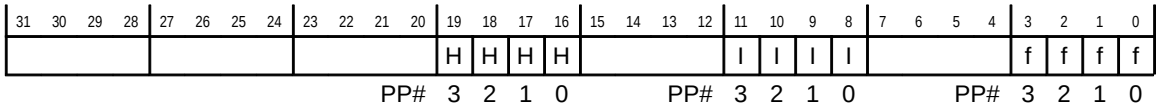
These bits are set by the floating-point unit; however, you can set or clear these bits by writing to FPST.

3.3.4 PP Error Register: PPERROR

The PPERROR register indicates the current status of the PPs. The contents of this register are read-only and are updated on a clock-by-clock basis to reflect status changes in the PPs. As shown in Figure 3–4, the PPERROR register contains the following status information:

- The type of memory fault when a PP has an instruction-cache fault or a load, store, or direct external access (DEA) address fault
- If a PP has executed an illegal instruction
- If a PP is halted

Figure 3–4. MP Register Showing PP Errors—PPERROR (Read Only)



Notes: 1) Illegal PP instructions set the PP error bit INTPEN[pe] (see Figure 3–2), and PP memory faults set INTPEN[mf].

2) All unnamed bits are reserved.

3.3.4.1 PP Memory Fault

PPERROR bits 0–3 contain the memory-fault status bits for PPs 0–3, respectively. The value of a PP's fault bit is 1 while the PP is performing a DEA or accessing an on-chip RAM location via a load or store instruction. Otherwise (while the PP is requesting instruction-cache service or not performing an access), the fault bit is 0. When a memory fault occurs, the PP is stalled, causing the fault bit to freeze at a value that indicates the type of access that caused the fault:

- f = 0—The fault occurred during instruction-cache servicing.
- f = 1—The fault occurred during a DEA or a load/store access of an illegal or invalid address below 0x0200 0000.

Load, store, and DEA fault addresses are stored in 0x0100 #0F8 (where # is the PP number). Instruction-cache fault addresses are also stored in 0x0100 #0F8. For more illegal address information, see Section 5.6, *Accessing Illegal Addresses.*

- bit 0: PP0 memory fault
- bit 1: PP1 memory fault
- bit 2: PP2 memory fault
- bit 3: PP3 memory fault

Note that a PP memory fault sets INTPEN[[mf]].

3.3.4.2 PP Illegal Instruction

Bits 8–11 in the PPERROR register indicate which PP attempted to execute an illegal PP instruction. The location of the illegal PP instruction (the value in the PP's ipa register) can be determined only if that PP is reset. The location of the illegal PP instruction is stored in location 0x0100 #7F8 (where # is the PP number) only after that PP has been reset.

- bit 8: PP0 illegal instruction
- bit 9: PP1 illegal instruction
- bit 10: PP2 illegal instruction
- bit 11: PP3 illegal instruction

Note that a PP illegal instruction sets INTPEN[[pe]].

3.3.4.3 PP Halt

Bits 16–19 in the PPERROR register indicate whether a PP is running (0) or halted (1). The MP can issue a reset to a PP to halt that PP (with the MP's cmnd instruction); the MP or the PP can issue a halt command to that PP (with a cmnd instruction).

- bit 16: PP0 halted
- bit 17: PP1 halted
- bit 18: PP2 halted
- bit 19: PP3 halted

When a PP halts, that bit is set in PPERROR⟦H⟧. This does not produce an INTPEN⟦pe⟧ (PP error) signal.

The reasons for a PP to be halted are:

- The PP was reset.
- The PP has halted itself or has been halted by the MP.

There is a difference between halting and stalling. The following conditions stall the PP but do not set the PP's halt bit in the PPERROR register:

- The PP attempted to execute an illegal instruction.
- The PP caused a memory fault.

When the MP sends a reset command (via its cmnd instruction) to the PP, the PP may require several cycles to complete its reset sequence. The MP can safely assume that the PP has completed reset only after the corresponding halt bit in the PPERROR register has changed to 1.

In the event that the PP was already halted at the time that the MP sent the reset command, logic associated with the PPERROR register forces the corresponding halt bit to 0 until the PP completes reset. Due to the latency in the 1-to-0 transition of the halt bit following execution of the cmnd instruction that resets the PP, the MP should allow one delay slot following the cmnd instruction before attempting to read the halt bit. In other words, the MP cannot reliably read the halt bit in the instruction slot that immediately follows the cmnd instruction.

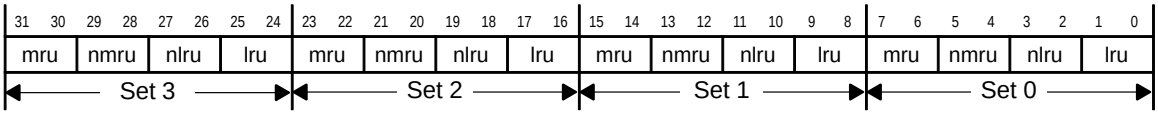
3.4 Cache Registers

The MP's instruction cache and data cache are almost identical in their organization. Both have sixteen tag registers and a least recently used (LRU) register. Each 4K-byte cache is divided into sixteen blocks (four sets of four blocks), each of which has a tag register with the 22 MSBs of its associated external memory address; each block is subdivided into four subblocks. The usage of each block is maintained in the LRU register; when a cache miss occurs, that register determines which block is the oldest block in this set and, therefore, a candidate to be loaded with the needed information. The cache is described in greater detail in Chapter 6, *Cache Management*. The cache registers are:

- **DLRU**, **ILRU**—Data (DLRU) or instruction (ILRU) least recently used information for all sixteen cache blocks (see Figure 3–5).
- **DTAG0–15**, **ITAG0–15**—Data (DTAG0–15) or instruction (ITAG0–15) tag information for each of the sixteen cache blocks. The tag contains the MSBs of the block's address in external memory (see Figure 3–6). The register contains a present (P) bit for each of the four subblocks in this block.

  The data cache tag also includes a dirty (D) bit for each of the four subblocks to indicate if the data in cache has been altered. The dirty bit indicates that that particular subblock must be written back into external memory before a cache replacement occurs for that block or subblock.

Figure 3–5. MP Cache LRU Register



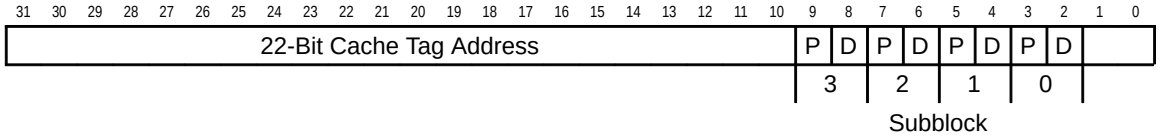
Key:

mru	most recently used block within this set.
nmru	next most recently used block within this set.
nlru	next least recently used block within this set.
lru	least recently used block within this set.

Notes: 1) Sixteen cache segments are divided into four sets, with each set containing four blocks; each block contains four subblocks.

2) mru, nmru, nlru, and lru are mutually exclusive for each set. These fields can have a value of 0, 1, 2, or 3, representing the block number in the set.

Figure 3–6. MP Cache Tag Registers



Key:

P Subblock is present when bit = 1 or not resident when bit = 0.

D Subblock has been modified when bit = 1 or not modified when bit = 0. Bit D is used for MP data cache only.

Notes: 1) All unnamed bits are reserved.

2) If P = 0, then D = 0.

Note:

Although you can modify the tag and LRU registers in the supervisor mode, this is not recommended. Any modifications can produce unpredictable behavior.

COMPATIBILITY

Certain silicon work-arounds on preproduction silicon versions 1 and 2 may require DTAG register modifications.

Since these control registers are visible to emulation, a software debugger can manipulate the contents of the instruction cache by inserting and removing breakpoints. Also, when communicating with other processors or peripherals, you can use the dcachef and dcachec instructions to ensure cache coherency and the integrity of data in external memory.

See the example in Section 11.10, *Clean Data Cache Using the dcachec Instruction.*

3.5 Timing Registers

The MP has a timer that provides on-chip time reference using the following registers:

- **TCOUNT** is the current 32-bit value of the timer (decrements by 1 at each clock cycle). When TCOUNT decrements to 0 and is 0 for one clock cycle, interrupt 10 (ti) is signaled (INTPEN[[ti]] = 1).
- **TSCALE** is the 32-bit reload value for the timer when TCOUNT decrements to 0 and is 0 for one clock cycle.

Example 3–1 issues a timer interrupt (ti) every 500 cycles. Note that the timer interrupt service routine is not shown.

Note:

When TCOUNT decrements by 1 until it reaches 0, INTPEN[[ti]] = 1, and if IE[[ie]] = IE[[ti]] = 1, the MP timer interrupt is issued.

Example 3–1. TCOUNT/TSCALE Sample Code

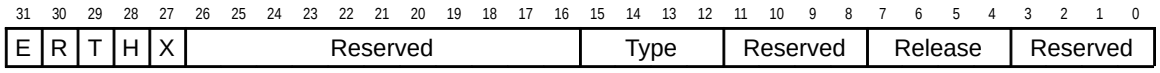
```
                    ; set TCOUNT = TSCALE for 500
                    ;  cycles
addu  499,r0,r5     ; constant = 499 → r5
swcr  TSCALE,r5,r7  ; r7 = TSCALE old, TSCALE new = r5
swcr  TCOUNT,r5,r6  ; r6 = TCOUNT old, TCOUNT new = r5
```

Note: MP must be in the supervisor mode to swap control registers (swcr instruction).

3.6 Configuration Register: CONFIG

The **CONFIG** register controls or reflects the state of options, as shown in Figure 3–7. In addition, the CONFIG register provides information about the current version of MVP silicon.

Figure 3–7. MP Configuration Register—CONFIG



The program may test or select modes of operation by the values in the CONFIG register. Software can write to the R, T, H, and X bits, but the remaining fields in the CONFIG register are read-only. The following subsections describe the bits and fields in the CONFIG register.

3.6.1 Find Information About Current Version of MVP Silicon

The Release and Type fields provide information about the current version of the MVP silicon. These fields are read-only and are not modified by writes to the CONFIG register:

- Release (bits 4–7): Current release of the MVP.
  - For preproduction silicon release number 1, the value is 0001.
  - For preproduction silicon release number 2, the value is 0010.
  - For production silicon release number 3, the value is 0011.
- Type (bits 12–15): Number of PPs included on the MVP.
  - For a four-PP device, the value is 0000.
  - For a one-PP device, the value is 0001.
  - For a two-PP device, the value is 0010.
  - For an eight-PP device, the value is 0011.

3.6.2 Enable/Disable Externally Initiated Packet Requests

The X bit (bit 27) is used to enable or disable externally initiated packet requests (XPT1–7) from external hosts or peripheral devices. See the P bit (bit 12) in Figure 2–13, *Video Controller Memory Control Register—FMEMCTL0/1*, for video controller enable or disable frame timer packet request signals. XPT4–7 share the linked-list addresses with the VC frame timer packet requests.

- If X = 0, externally initiated packet requests are disabled.
- If X = 1, externally initiated packet requests are enabled.

Figure 5–2, *MP Parameter RAM Contents*, and Figure 5–3, *Externally Initiated Packet Transfer Linked-List Address Pointers*, illustrate the seven linked-list packet request addresses and the MP parameter RAM buffer required for those externally initiated packet requests.

Bit X is not available on preproduction silicon version 1.

3.6.3 Enable/Disable High-Priority Mode

The H bit (bit 28) is used to enable or disable the high-priority MP cache, DEA, and packet requests at a level just below that of externally initiated packet requests (see NO TAG, *Transfer Controller's Request Prioritization*, in the *MVP Transfer Controller User's Guide*).

- If H = 0, high-priority MP requests are disabled.
- If H = 1, high-priority MP requests are enabled.

Bit H is not available on preproduction silicon version 1.

3.6.4 Disable TC Packet Request Priority Round-Robin

The T bit (bit 29) is the round-robin disable bit for TC service requests. This bit controls how the TC arbitrates among competing, equal-priority requests from the MP and PPs for TC services. The bit affects requests for cache service, DEAs, and packet transfer requests (urgent, high, and low priority). By default, competing requests at the same priority level are selected on the basis of a token that circulates in a round-robin fashion among the processors requesting service. An alternative mode is to assign fixed subpriority levels within each priority level. This mode fixes MP requests at the highest subpriority, with successively lower fixed subpriorities assigned to PP0, PP1, PP2, and PP3; MP cache and DEA requests are given a higher subpriority than urgent MP requests for packet transfers. Refer to NO TAG, *Transfer Controller's Request Prioritization*, in the *MVP Transfer Controller User's Guide*.

- If T = 0, the TC grants packet transfer requests of equal priority in a round-robin fashion (that is, the token circulates to the next processor with a request). T is 0 at reset.
- If T is 1, the packet request priorities are fixed and the round-robin tokens begin with PP3's global bus.

3.6.5 Enable Crossbar Round-Robin

The R bit (bit 30) is the crossbar round-robin enable bit for the PPs' on-chip data and parameter RAMs. This bit controls how the crossbar network arbitrates among competing, equal-priority requests from the PPs to access the same bank of RAM. Each RAM bank is physically connected to the local port of a single PP but can be accessed through any PP's global port. By default, competing requests are assigned fixed subpriority levels, with a PP local-port access given highest subpriority, and with successively lower subpriorities assigned to global-port accesses by PP0, PP1, PP2, and PP3. An alternative mode is to select among the PPs' access requests on the basis of a token that circulates in a round-robin fashion among PP access requests. Refer to NO TAG, *Crossbar Access Priority,* in the *MVP System-Level Synopsis*.

- If R = 0, the priorities are fixed, and the round-robin tokens begin with PP3's global bus. R is 0 at reset.
- If R is 1, the crossbar grants access requests of equal priority in a round-robin fashion (that is, the token circulates to the next processor with a request).

3.6.6 Endian Mode Indicator

The E bit (bit 31) is the endian read-only bit that indicates whether the MVP powered-up in big- or little-endian mode.

- E = 0 for big endian.
- E = 1 for little endian.

3.7 System Registers

These 32-bit registers are used by the operating system:

- **SYSSTK** represents the system stack pointer (8-byte boundary).
- **SYSTMP** is a temporary register (for example, the user r1 stack pointer is saved, as shown in Example 3–2).

Example 3–2 saves r1–r31 in a context switch by using the SYSSTK and SYSTMP registers.

Example 3–2. Sample MP Context Register Save Code

```
wrcr  SYSTMP,r1        ; SYSTMP = user's stack pointer in r1
rdcr  SYSSTK,r1        ; r1 = system stack pointer (64–bit boundary)
st.d  –16*8(r1:m),r30  ; push (r30,r31), and also modify
                       ; r1 = SYSSTK – 16 doublewords
st.d  1*8(r1),r28      ; push (r28,r29)
st.d  2*8(r1),r26      ; push (r26,r27)
.
.                      ; push (r4 – r25) onto stack
.
st.d  14*8(r1),r2      ; push (r2,r3)
rdcr  SYSTMP,r2        ; r2 = user's stack pointer in SYSTMP
st.d  15*8(r1),r2      ; push user's original r1 onto stack
```

Note: The program must be in supervisor mode to issue a wrcr instruction for control registers < 0x4000.

3.8 Memory Fault Registers

When a maskable or nonmaskable memory address exception occurs (a maskable exception sets the INTPEN[mf] bit shown in Figure 3–2), information about the memory fault is captured in the following registers:

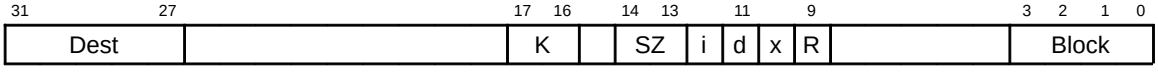
- Memory fault operation register: FLTOP
- Memory fault tag register: FLTTAG
- Memory fault address and data registers: FLTADR, FLTDTH, FLTDTL

See subsection 5.6.1, *MP Memory Fault Service Routine Outline*, for additional information about memory faults.

3.8.1 Memory Fault Operation Register: FLTOP

The FLTOP register is used by the memory fault interrupt service routine to determine the type of memory error. Figure 3–8 shows the FLTOP register.

Figure 3–8. Memory Fault Operation Register—FLTOP



Note: All unnamed bits are reserved.

The FLTOP register is cleared to 0 at power-up reset. When a nonmaskable memory fault occurs, machine logic sets FLTOP's i, d, or x bit to 1 to indicate the nature of the fault. The memory-fault service routine checks these bits to detect a nonmaskable fault. If the fault is recoverable, the fault routine should write 0s to the i, d, and x bits before returning control to the interrupted program.

- Block (bits 0–3): Four bits that represent the set and block numbers that identify which DTAG*n* register (where *n* = 0, 1, ..., 15) is loaded into register FLTTAG if the fault is the result of a data-cache fault (see Figure 6–2, *MP Cache Structure—Tag Registers and LRU Stacks*). FLTOP⟦block⟧ and FLTTAG register values are not valid for direct external memory access faults (FLTOP⟦x⟧ = 1).

- R (bit 9): Used by the memory fault interrupt service routine for normal (R = 0) or modified (R = 1) return sequence. The modified return is used when a long-immediate operand is in progress or when a branch annul was interrupted in the branch-delay slot. (For more information, see Section 9.3, *Returning From Interrupts and Traps*).

- x (bit 10): Used to signal a direct external memory access fault or when an on-chip load/store is treated as dld/dst instructions. The illegal MP address is in the FLTADR register.

- d (bit 11): Signals that a data-cache fault has occurred and the MP faulted address is held in the FLTADR register.

Note:

If a fault occurs in performing a subblock write-back from data cache, the FLTTAG register contains the original P and D bits before the fault; however, all of that block's D bits are reset in its DTAG register. This ensures that the same memory fault does not re-occur if the same block is replaced during the interrupt service routine. However, even if the block is read successfully, writing to it may not be possible. For example, if the block is write-protected, recovery from that memory fault error is not possible.

- i (bit 12): Signals that an MP instruction-cache fault has occurred. The EPC register contains the faulted MP instruction address.

- SZ (bits 13–14): Copies the data size of the MP load/store operation as byte, halfword, word or doubleword.

Bits		
14	**13**	**Data Size**
0	0	8-bit byte
0	1	16-bit halfword
1	0	32-bit word
1	1	64-bit doubleword

- K (bits 16–17): Helps determine the type of MP memory operation.

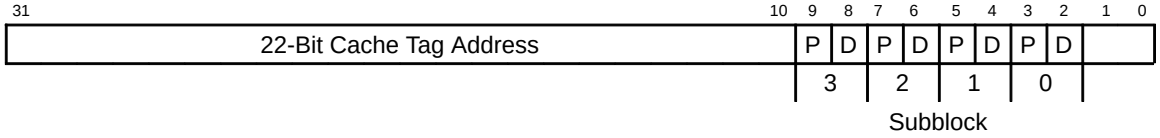
Bits		
17	**16**	**Memory Operation**
0	0	Load
0	1	Unsigned load
1	0	Store
1	1	Cache clear (dcachef/dcachec)

- Dest (bits 27–31): A copy of the faulting MP instruction's *dest* field as a register number 0–31. If this is a vector instruction, then:
  - If the result is to an accumulator with a vld, then Dest = vld's destination.
  - If the result is to an accumulator with a vst, then Dest = vst's source.

3.8.2 Memory Fault Tag Register: FLTTAG

When a memory fault occurs as a result of a cache load or write-back, the FLTTAG register contains a copy of the tag register at the point at which the memory fault occurred. The tag includes the faulting address if the fault occurred during an attempt to write back a modified (dirty) subblock. See Figure 3–9.

Figure 3–9. Memory Fault Tag Register—FLTTAG



Notes: 1) All unnamed bits are reserved.

2) FLTTAG is not valid for DEA errors (FLTOP⟦X⟧ = 1).

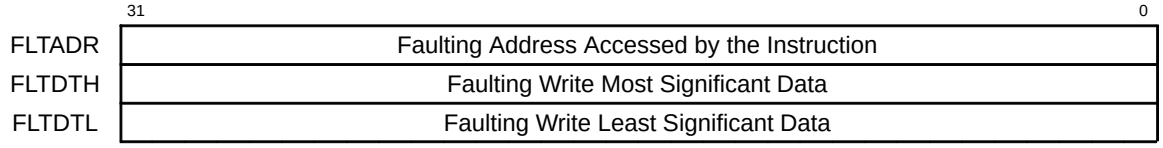
- D (bits 2, 4, 6, and 8): Subblock has been modified when bit = 1 or not modified when bit = 0. The dirty (D) bit is used for MP data cache only.
- P (bits 3, 5, 7, and 9): Subblock is present when bit = 1 or not resident when bit = 0.

3.8.3 Memory Fault Address and Data Registers: FLTADR, FLTDTH, FLTDTL

The following registers (shown in Figure 3–10) contain information about the memory fault address and data:

- **FLTADR** contains the faulting address that was accessed by the MP instruction.
- **FLTDTH** contains the 32 MSBs of the data value if a write fault occurred.
- **FLTDTL** contains the 32 LSBs of the data value if a write fault occurred.

Figure 3–10. Memory Fault Address and Data Registers—FLTADR, FLTDTH, FLTDTL



Note: For word, halfword, or byte data, FLTDT* has the faulting write data replicated in the 64-bit data register two, four, or eight times, respectively.

3.9 Emulation Registers

In addition to the other control registers in this section, various registers in the analysis logic are related to breakpoints and can act as control registers to provide software debugging if an emulator is not available. They include:

- **Emulation communication control** (**ECOMCNTL**) register
- **Emulation analysis status** (**ANASTAT**) register
- **Emulation breakpoint 1** (**BRK1**) register
- **Emulation breakpoint 2** (**BRK2**) register

Note:

These registers are reserved for the debugger software. Do **not** modify these registers, or unpredictable results may occur.

Chapter 4

Master Processor Pipelines

To improve performance, the MP uses several pipelines for overlapped operations. Several instructions can be in successive stages of completion in a given clock cycle as they flow through a pipeline. This chapter discusses MP pipelines: the instruction-execution pipeline (where most of the integer operations are performed) and the floating-point unit pipelines.

Topics

4.1 FEA (Instruction-Execution) Pipeline

Recall from subsection 3.1, *FEA Pipeline Registers*, that program execution uses a three-stage FEA (fetch-execute-access) pipeline. The program pipeline can overlap successive stages of execution of three different instructions in the same cycle.

Example 4–1 illustrates the FEA pipeline for a program that reads the last stack entry, then generates the sum of five integers. Note that the load instruction (ld) does not complete until the end of the access memory pipeline stage A in cycle 3.

Example 4–1. FEA Pipeline in the Sum-of-Five Integers—Normal Case

```
ld 0(r1), r10       ; r10 = Mem (r1)
add r5, r6, r4      ; r4 = r5 + r6
add r7, r4, r4      ; r4 = r7 + r4
add r8, r4, r4      ; r4 = r8 + r4
add r9, r4, r4      ; r4 = r9 + r4
```

Cycle	Fetch (F)	Execute (E)	Access (A)
1	ld r10	— — —	— — —
2	add r5	ld r10	— — —
3	add r7	add r5	ld r10
4	add r8	add r7	— — —
5	add r9	add r8	— — —
6	— — —	add r9	— — —

Note: Example assumes no data or instruction-cache misses, contention, or interrupts.

4.1.1 Instruction Fetch (F)

The PC (program counter) resides in the F stage of the FEA pipeline and contains the address of the next 32-bit instruction to be fetched. If this instruction is resident in the MP's instruction cache, FEA pipeline acquisition of the 32-bit instruction continues. However, if the instruction is not resident in the MP's instruction cache, then an instruction-cache miss is issued to the transfer controller (TC), and the FEA pipeline stalls. After the TC has completed the instruction-cache service (see Example 4–2), the FEA pipeline resumes acquisition of the next 32-bit instruction.

Example 4–2. FEA Pipeline in the Sum-of-Five Integers—Fetch Instruction Cache Miss

```
add r5, r6 r4      ; r4 = r5 + r6
add r7, r4, r4     ; r4 = r7 + r4
add r8, r4, r4     ; r4 = r8 + r4
add r9, r4, r4     ; r4 = r9 + r4
```

Cycle	Fetch (F)	Execute (E)	Access (A)
1	add r5	— — —	— — —
2	add r7	add r5	— — —
3	add r8	add r7	— — —
	Instruction-Cache Miss		
K	add r9	add r8	— — —
K+1	— — —	add r9	— — —

Note: Example assumes an instruction-cache miss but no contention or interrupts.

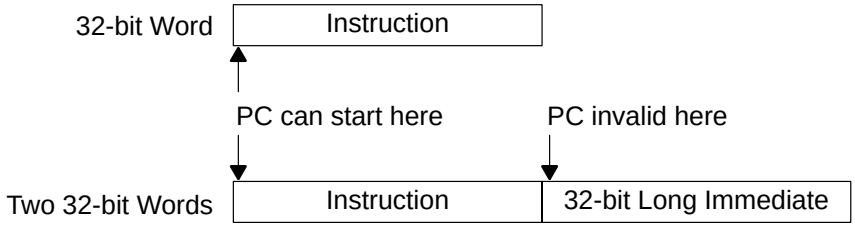
If the fetched instruction requires a long 32-bit immediate operand (indicated by a 1 in instruction bits 12, 20, and 21), the fetch (F) pipeline stage acquires the next 32-bit word as an operand from the instruction cache at byte address PC+4. The 32-bit long-immediate operand adds one extra cycle (or more if there is an instruction-cache miss) to the F pipeline. The execute (E) pipeline stage for this instruction will not start until the optional 32-bit long-immediate operand is available at the end of stage F.

If the fetched instruction is a trap instruction (trap or etrap), then the next F stage is inhibited. As a result, there is no **delay slot instruction** (the first instruction that follows a branch instruction) associated with traps; delay slot instructions are associated with branch instructions other than traps (for example, br, brcr, bcnd, bbz, etc.). When the transfer address of a trap is loaded into the PC, there is the potential for one or two cache misses if:

- The trap transfer address is not resident (a cache miss is issued to the TC). However, in most cases, the trap or interrupt transfer address is found in the MP's parameter RAM, which is accessed directly, bypassing the data cache.
- The first instruction of the service routine is not resident in the instruction cache.

Note:

MP instructions are either one 32-bit word or, when a 32-bit long-immediate operand is included, two 32-bit words. The target of a branch or jump instruction should always be the first word of an instruction and never the 32-bit immediate operand in the second word of a two-word instruction. If this should occur, the MP operation is unpredictable.



The cache-miss service time by the TC is variable and is a function of the following:

- The type of external memory (1-, 2-, or 3-cycle RAMs)
- The bus size of external memory (64, 32, 16, or 8 bits)
- The number of rows per page in external memory (1, 2, or 4)
- The round-robin mode selected for the TC's packet transfers (fixed or variable)
- Data cache flush write-back time for 0–4 subblocks (if needed after data modifications)

4.1.2 Instruction Execute (E)

The IP (instruction pointer) contains the address of the instruction that is in the instruction execute (E) pipeline stage. The following events occur during this stage:

1) The instruction is decoded.
2) Source operands are read from registers.
3) The specified operation is performed.
4) Results are written into the destination register.

Pipeline stalls can occur as the result of register scoreboarding (see subsection 2.2.3, *Register Scoreboarding*, for more information about register scoreboarding). Once the registers are available (for example, the scoreboard load and write flags are both 0 for the source and destination registers), the E pipeline stage resumes operation, as shown in Example 4–3.

Any result that is not written into the destination register at the end of the E pipeline stage is flagged in the SB (scoreboard) register. An example is the single-precision floating-point square root instruction (fsqrt), which requires multiple cycles and sets the SB's write flag for the destination register. Also, any load register instruction (for example, ld, dld, or vld) sets the SB's load flag for the destination register, indicating that the load will not complete until the end of a subsequent access memory (A) pipeline stage.

Example 4–3.FEA Pipeline in the Sum-of-Five Integers—Load Data Delay

```
add r5, r6, r4          ; r4 = r5 + r6
ld 0(r1), r7            ; r7 = Mem (r1)
add r7, r4, r4          ; r4 = r7 + r4
add r8, r4, r4          ; r4 = r8 + r4
add r9, r4, r4          ; r4 = r9 + r4
```

Cycle	Fetch (F)	Execute (E)	Access (A)	
1	add r5	— — —	— — —	
2	ld r7	add r5	— — —	
3	add r7	ld r7	— — —	
	Data-Cache Miss			
K	(add r8)	(add r7)	ld r7	← (**add r7** is stalled, waiting for the **ld r7** instruction to complete)
K+1	add r8	add r7	— — —	
K+2	add r9	add r8	— — —	
K+3	— — —	add r9	— — —	

Note: Example assumes a data-cache miss but no instruction-cache misses, contention, or interrupts.

4.1.2.1 Branch Delay Slots and the E Pipeline Stage

You can specify whether or not the pipeline delay slot that follows a branch or jump instruction is annulled.

The instruction (called a branch delay slot instruction) that follows a nonannulled branch or jump instruction is executed before the instruction at the target address of the branch. Branch (and jump) instructions load the PC with the branch target address at the end of the execute (E) stage of the FEA pipeline. The MP fetches the delay slot instruction at the same time it executes the branch; in other words, the E stage of the branch coincides with the delay slot instruction's F stage. A delay slot instruction must be a single-word instruction.

When a branch or jump instruction is annulled, the MP automatically inserts a nop instruction into the IR register in lieu of fetching a delay slot instruction. This adds a minimum of one clock to the execution time of the next branch instruction, regardless of whether the branch is taken. The next instruction executed after the branch instruction is either the instruction at the branch target address (if the branch is taken) or the instruction that follows the branch instruction (if the branch is not taken).

4.1.3 Memory Data Access (A)

When the instruction execute (E) pipeline stage detects a memory load or store instruction, the memory request is passed to the memory data access (A) pipeline stage after the integer ALU forms the address in the E pipeline stage. The SB memory load flag is also set during the E pipeline stage (see subsection 2.2.3, *Register Scoreboarding*, for more information about the SB flags).

When a memory access is required, the A pipeline stage completes the process. If the required address is in the on-chip RAM or resident in the MP's data cache, the load or store operation is performed without assistance from the TC. In the case of a load, the SB (scoreboard) register's load bit for the target register is reset when the load operation is completed.

If the required external memory address is not resident in the data cache, the MP signals its data-cache miss to the transfer controller (TC) and the A pipeline stalls; however, the fetch (F) stage and E stage pipelines continue operation unless they encounter a stall condition (for example, a source operand is not available in stage E or an instruction-cache miss occurs in stage F). After the TC has completed the data-cache service, the memory operation resumes in the A pipeline stage.

The TC handles DEA loads in a manner similar to data-cache misses.

Notes:

1) Not all instructions require the A pipeline stage.

2) There is a one-cycle latency delay between data load and data availability in the E (execute) stage of the pipeline. To avoid this, you can place a useful instruction in this slot, rather than allowing the SB register to stall the E pipeline stage while waiting for source operand and destination register availability.

4.2 Floating-Point Unit Pipelines

The MP's floating-point unit has two pipelines:

- A floating-point multiply pipeline for multiplies
- A floating-point add pipeline for adds

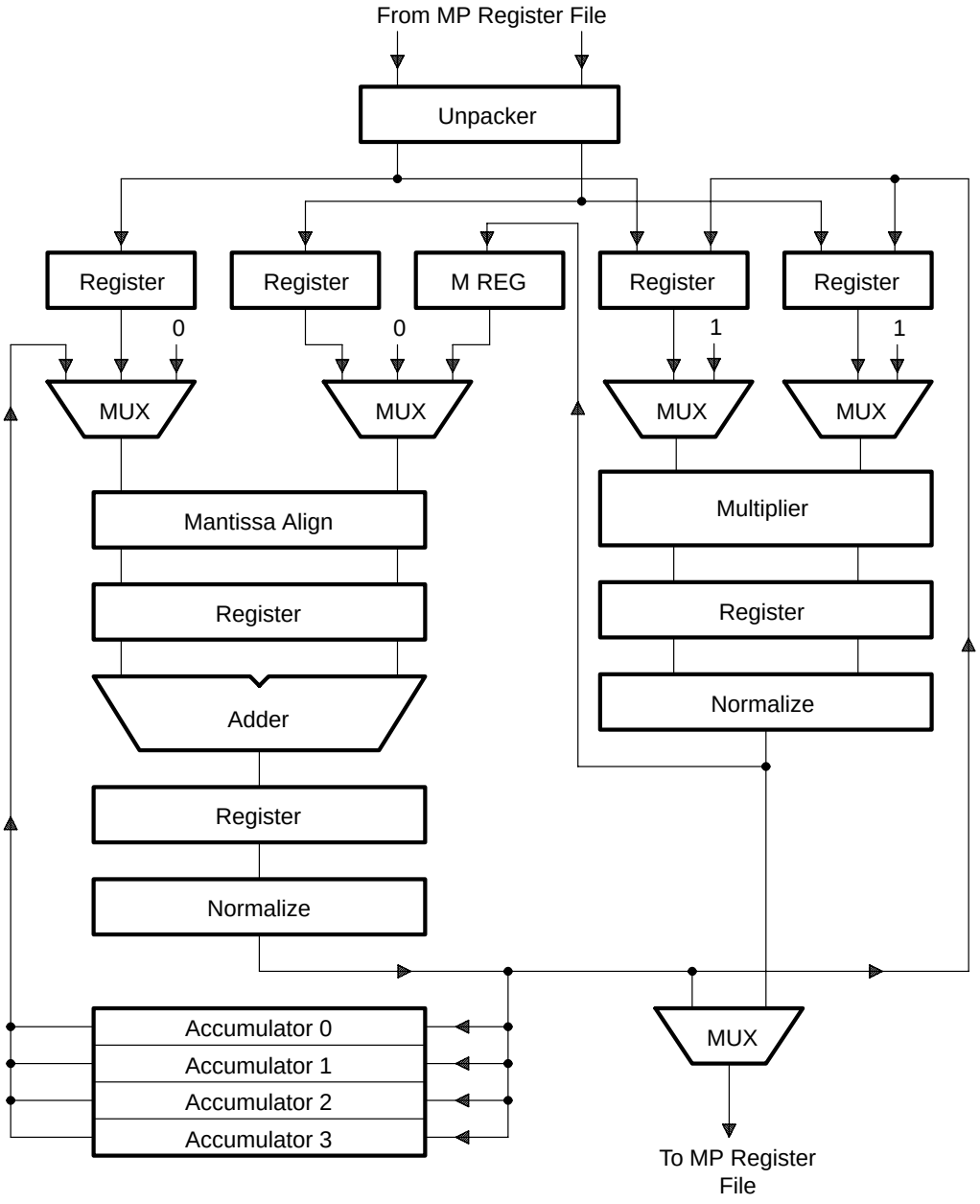
These pipelines are separate from the FEA pipeline described in Section 4.1. The FEA pipeline is used primarily for integer operations. The components involved in floating-point operations, however, include:

- A double-precision **floating-point multiply pipeline** with a single-precision core
- A double-precision **floating-point add pipeline**
- Four **double-precision floating-point accumulators** (with scoreboarding)
- A **double-precision floating-point path** from the output of the floating-point multiply pipeline to the input of the floating-point add pipeline and a similar path from the output of the add pipeline to the input of the multiply pipeline.

In the execute stage (E) of the FEA pipeline, the floating-point unit acquires operands as input. The instruction pipeline continues in the normal manner, while the floating-point multiply and add pipelines perform their respective floating-point operations in parallel.

A general flow diagram of the floating-point unit operation is shown in Figure 4–1.

Figure 4–1. General Floating-Point Unit Flow



4.2.1 Floating-Point Multiply Pipeline

The floating-point multiply pipeline performs the following operations:

- Signed and unsigned integer multiplication
- Single- and double-precision floating-point multiplication
- Single- and double-precision floating-point division
- Single- and double-precision floating-point square root
- Multiply portion of vector multiply-and-accumulate operations

The single-precision floating-point multiply pipeline has the following three stages:

1) The unpack stage partitions the floating-point numbers into their respective fields for each operand: the sign field, exponent field, and the mantissa field. Input exceptions are detected in this stage. Special cases such as NaNs (not-a-numbers), zero, and infinities are also detected in this stage. A floating-point instruction in the unpack stage of the floating-point pipeline simultaneously resides in the E stage of the FEA pipeline.

2) The multiply stage calculates the product of the two mantissas.

3) The normalize stage packs the sign field, the exponent field, and mantissa field of the product after normalization and rounding into standard IEEE-754 floating-point format. Output exceptions are detected in this stage.

As shown in Example 4–4, a new single-precision floating-point multiply can begin on each cycle. Note that the last instruction must remain in the M stage for four clocks to calculate the double-precision result.

Example 4–4. Floating-Point Multiply Pipeline

```
fmpy.iii r4, r5, r8              ; I r8 = r4 × r5
fmpy.uuu r5, r4, r9              ; U r9 = r5 × r4
fmpy.sss r6, r7, r10             ; SP FP r10 = r6 × r7
fmpy.ssd r7, r6, r12             ; SP FP r7 times r6 with DP FP
                                 ; output in r12
```

Cycle	Unpack	Multiply	Normalize	
1	fmpy r4	— — —	— — —	
2	fmpy r5	fmpy r4	— — —	
3	fmpy r6	fmpy r5	fmpy r4	→ r8
4	fmpy r7	fmpy r6	fmpy r5	→ r9
5	— — —	fmpy r7	fmpy r6	→ r10
6	— — —	fmpy r7	— — —	
7	— — —	fmpy r7	— — —	
8	— — —	fmpy r7	— — —	
9	— — —	— — —	fmpy r7	→ r12

Notes: 1) Example assumes no exceptions or denormal values and no instruction-cache misses, contentions, or interrupts.

2) **SP** = single precision
DP = double precision
FP = floating point
I = signed integer
U = unsigned integer

Microcode sequences that use the floating-point multiply unit provide double-precision floating-point multiplication, floating-point division, and floating-point square root operations. These operations require additional cycles to return an answer. Example 4–5 illustrates some of these pipelines. Table 8–2, *Latencies of Floating-Point Operations*, illustrates the latency time (cycles to obtain the first answer) and the start-next time (when another floating-point operation of the same type can begin operation in this floating-point pipeline).

The output of the floating-point multiply unit is passed directly to the floating-point add unit when a vector multiply-and-accumulate (vmac) instruction type is submitted.

Example 4–5. Floating-Point Multiply Pipeline

```
fmpy.ddd r4, r6, r8              ; r8 = r4 × r6 all in DP FP
fdiv.sss r2, r3, r10             ; r10 = r2 / r3 all in SP FP
fdiv.ddd r6, r8, r12             ; r12 = r6 / r8 all in DP FP
fsqrt.ss r3, r11                 ; r11 = sqrt (r3) all in SP FP
fsqrt.dd r6, r4                  ; r4 = sqrt (r6) all in DP FP
```

Cycle	Unpack	Multiply	Normalize	
1	fmpy r4	— — —	— — —	
	Double-Precision Floating-Point Multiply Sequence			
5	fdiv r2	fmpy r4	— — —	
6	— — —	fdiv r2	fmpy r4	→ r8
	Single-Precision Floating-Point Divide Sequence			
11	fdiv r6	fdiv r2	— — —	
12	— — —	fdiv r6	fdiv r2	→ r10
	Double-Precision Floating-Point Divide Sequence			
31	fsqrt r3	fdiv r6	— — —	
32	— — —	fsqrt r3	fdiv r6	→ r12
	Single-Precision Floating-Point Square Root Sequence			
40	fsqrt r6	fsqrt r3	— — —	
41	— — —	fsqrt r6	fsqrt r3	→ r11
	Double-Precision Floating-Point Square Root Sequence			
67	— — —	— — —	fsqrt r6	→ r4

Notes: 1) Example assumes no exceptions or denormal values and no instruction-cache or data-cache misses, contention, or interrupts.

2) This example primarily shows the generation of results; most of the multiple-cycle multiply stages are not shown.

3) **SP** = single precision
DP = double precision
FP = floating point

4.2.2 Floating-Point Add Pipeline

The floating-point add pipeline performs the following operations:

- ❑ Single- and double-precision floating-point addition
- ❑ Single- and double-precision floating-point subtraction
- ❑ Single- and double-precision floating-point comparison
- ❑ Conversions for integer and for single- and double-precision floating point
- ❑ The add/subtract portion of vector multiply-and-accumulate operations

A new single-precision or double-precision floating-point add-class operation can begin on each cycle (see Example 4–6).

The double-precision floating-point add unit pipeline has the following four stages:

1) Unpack stage partitions the floating-point numbers into their respective fields for each operand: the sign field, exponent field, and the mantissa field. Input exceptions are detected in this instruction execution (E) stage. Special cases such as NaNs (not-a-numbers), zero, and infinities are also detected in this stage. The vmac-type instructions bypass this stage of the floating-point add unit pipeline.
2) The align stage aligns the binary point of the two mantissas by right-shifting the mantissa with the smaller exponent.
3) The add stage adds or subtracts the two mantissas.
4) The normalize stage packs the sign field, exponent field, and mantissa field of the result after normalization and rounding into standard IEEE-754 floating-point format. Also, output exceptions are detected in this stage.

Example 4–6. Floating-Point Add Pipeline

```
fadd.sss r2, r3, r8          ; SP r8 = r2 + r3
fsub.sss r3, r2, r9          ; SP r9 = r3 – r2
fadd.ddd r6, r4, r10         ; DP r10 = r6 + r4
fcmp.sd r3, r6, r12          ; SP r3 – DP r6 as status in r12
frndn.id 17, r14             ; DP r14 = int (17)
; Fetch of the frndn.id 17, r14 instruction requires two cycles;
; the extra cycle for the long immediate constant 17.
```

Cycle	Unpack	Align	Add	Normalize	
1	fadd r2	— — —	— — —	— — —	
2	fsub r3	fadd r2	— — —	— — —	
3	fadd r6	fsub r3	fadd r2	— — —	
4	fcmp r3	fadd r6	fsub r3	fadd r2	→ r8
5	frndn	fcmp r3	fadd r6	fsub r3	→ r9
6	Long 17	— — —	fcmp r3	fadd r6	→ r10
7	— — —	frndn 17	— — —	fcmp r3	→ r12
8	— — —	— — —	frndn 17	— — —	
9	— — —	— — —	— — —	frndn 17	→ r14

Note: Example assumes no exceptions or denormal values and no instruction-cache misses, contentions, or interrupts.

Table 8–2, *Latencies of Floating-Point Operations*, illustrates the latency time (cycles to obtain the first answer) and the start-next time (when another floating-point operation of the same type can begin operation in the add unit pipeline).

4.2.3 Double-Precision Floating-Point Path Between Floating-Point Multiply and Add Pipelines

The floating-point multiply pipeline provides a double-precision floating-point path to pass a double-precision floating-point value to the floating-point add pipeline when using the vector multiply-and-accumulate (vmac) instruction class and denormal outputs. Example 4–7 shows a matrix multiply that performs these operations:

1) The floating-point multiply pipeline performs the product of two single-precision floating-point numbers to produce a double-precision floating-point answer.

2) The floating-point add pipeline performs the summation of two double-precision floating-point numbers to generate a double-precision floating-point result.

3) The double-precision floating-point sums are written into accumulators.

4) The final double-precision floating-point sums are converted to single- or double-precision floating-point numbers and written into registers.

Example 4–7.[1x3] x [3x3] Matrix Multiply—vmac Floating-Point Pipelines

r4	r5	r6	×	r7	r8	r9	=	r16	r17	r18
				r10	r11	r12				
				r13	r14	r15				

```
vmac.ssd r7, r4, 0, a0        ; r7 × r4 + 0   = a0
vmac.ssd r8, r4, 0, a1        ; r8 × r4 + 0   = a1
vmac.ssd r9, r4, 0, a2        ; r9 × r4 + 0   = a2
vmac.ssd r10, r5, a0, a0      ; r10 × r5 + a0 = a0
vmac.ssd r11, r5, a1, a1      ; r11 × r5 + a1 = a1
vmac.ssd r12, r5, a2, a2      ; r12 × r5 + a2 = a2
vmac.sss r13, r6, a0, r16     ; r13 × r6 + a0 = r16 (SP FP out)
vmac.sss r14, r6, a1, r17     ; r14 × r6 + a1 = r17 (SP FP out)
vmac.sss r15, r6, a2, r18     ; r15 × r6 + a2 = r18 (SP FP out)
```

	Floating-Point Multiply			Floating-Point Add			
Cycle	Unpack	Multiply	Normalize	Align	Add	Normalize	
1	vmac r7	— — —	— — —	— — —	— — —	— — —	
2	vmac r8	vmac r7	— — —	— — —	— — —	— — —	
3	vmac r9	vmac r8	vmac r7	— — —	— — —	— — —	
4	vmac r10	vmac r9	vmac r8	vmac r7	— — —	— — —	
5	vmac r11	vmac r10	vmac r9	vmac r8	vmac r7	— — —	
6	vmac r12	vmac r11	vmac r10	vmac r9	vmac r8	vmac r7	▶ a0
7	vmac r13	vmac r12	vmac r11	vmac r10	vmac r9	vmac r8	▶ a1
8	vmac r14	vmac r13	vmac r12	vmac r11	vmac r10	vmac r9	▶ a2
9	vmac r15	vmac r14	vmac r13	vmac r12	vmac r11	vmac r10	▶ a0
10	— — —	vmac r15	vmac r14	vmac r13	vmac r12	vmac r11	▶ a1
11	— — —	— — —	vmac r15	vmac r14	vmac r13	vmac r12	▶ a2
12	— — —	— — —	— — —	vmac r15	vmac r14	vmac r13	▶ r16
13	— — —	— — —	— — —	— — —	vmac r15	vmac r14	▶ r17
14	— — —	— — —	— — —	— — —	— — —	vmac r15	▶ r18

Notes: 1) Example assumes no exceptions or denormal values and no instruction-cache or data-cache misses, contentions, or interrupts.

2) **SP** = single precision
FP = floating point

4.3 Pipeline Implications

The pipelines described in this chapter imply the following:

- An instruction-cache miss stalls the instruction fetch (F) and execute (E) stages until the TC cache service is completed.
- 32-bit long immediates require an extra F cycle (minimum).
- There is a one-cycle branch delay slot.
- A data-cache miss or DEA load stalls the memory access (A) stage until the TC completes the access and the target register is updated.
- There is a minimum of one cycle for the load data delay slot.
- If the transfer address is not an MP parameter RAM address and is not resident, trap instructions stall the F stage. A TC cache-miss service is required before the F stage can continue.
- In back-to-back memory references, the second memory reference stalls the A stage until the first memory reference is completed.
- A floating-point unit instruction requires several cycles to complete.
  - Double-precision multiply or any divide or square root keeps the floating-point multiply unit busy for additional cycles.
  - Denormal input/output operands in the floating-point multiply unit issue a request to the floating-point add unit to correct the denormal values.
  - If in the sequential mode (FPST⟦sm⟧ = 1 as shown in Figure 3–3, *MP Floating-Point Status Register—FPST*), a second floating-point operation stalls while waiting for the first floating-point operation to complete.
  - If the floating-point multiply unit and floating-point add unit produce an output on the same cycle, the floating-point add result is written first, then the floating-point multiply unit writes its answer on the next clock, as shown in Example 4–8.
  - An accumulator that is scoreboarded may delay a subsequent vector instruction that requires that accumulator's data as input (see subsection 4.3.1).

- If the floating-point unit is not empty and a swap or write to a control register (swcr or wrcr instruction, respectively) is issued when not in a floating-point exception trap service routine, the FEA pipeline stalls until the floating-point pipeline is empty.

Example 4–8. Simultaneous Results From fadd and fmpy

```
fadd.sss r5, r6, r7        ; r7 = r5 + r6 all in SP FP
fmpy.sss r8, r9, r10       ; r10 = r8 × r9 all in SP FP
fadd.ddd r14, r14, r16     ; r16 = r14 + r14
```

	Floating-Point Multiply			Floating-Point Add				
Cycle	Unpack	Multiply	Normalize	Unpack	Align	Add	Normalize	
1	— — —	— — —	— — —	fadd r5	— — —	— — —	— — —	
2	fmpy r8	— — —	— — —	— — —	fadd r5	— — —	— — —	
3	— — —	fmpy r8	— — —	fadd r14	— — —	fadd r5	— — —	
4	— — —	— — —	(fmpy r8)	— — —	fadd r14	— — —	fadd r5	▶ r7
5	— — —	— — —	▶ r10	— — —	fadd r14	— — —	— — —	

Notes: 1) Both the fadd and fmpy instructions produce an answer at the end of clock 4.

2) The fadd instruction stores its answer first at the end of clock 4; both pipelines in the floating-point unit do not advance for one cycle during cycle 4.

3) The fmpy instruction stores its answer second at the end of clock 5.

4) **SP** = single precision
FP = floating point

5) Example assumes no exceptions or denormal values and no instruction-cache or data-cache misses, contentions, or interrupts.

4.3.1 Accumulator Scoreboarding

The four floating-point unit accumulators (a0, a1, a2, and a3) provide intermediate double-precision floating-point sums and support overlapped vector multiply and accumulate operations. Writes to each accumulator are scoreboarded to prevent use of the value in that accumulator until the value is valid.

Example 4–9 demonstrates how scoreboarding preserves the logical order of execution of a sequence of vmac instructions with data dependencies. Assume that the values in registers r4 through r9 are all available at the time the example begins. For the sake of brevity, refer to the first, second, and third vmac instructions in Example 4–9 as vmac1, vmac2, and vmac3, respectively. Note that vmac2 depends on the value loaded into accumulator a0 by vmac1.

In clock cycle 2 of Example 4–9, vmac2 is loaded into the MP's instruction register, where it is held during the unpack stage of the floating-point pipeline. At this time, the scoreboard logic detects vmac2's dependency on vmac1's a0 value. A hardware interlock prevents vmac2's operands from propagating through the multiply pipeline as long as vmac1 remains in the multiply pipeline. Only when vmac1 enters the add pipeline in cycle 4 are vmac2's operands unpacked, allowing vmac2 to begin propagating through the multiply pipeline. The delay introduced into vmac2 during cycles 2 and 3 is sufficient to ensure that vmac1's result, which is loaded into a0 at the end of cycle 6, is available to be used by vmac2 when it enters the add pipeline at the start of cycle 7.

Similarly, vmac3 is dependent on the value written to a0 by vmac2.

Example 4–9 also points out the importance of avoiding data dependencies whenever possible between consecutive vmac instructions in order to improve performance. If no data dependencies had existed among the three instructions in this example, the instructions would have followed each other through the multiply and add pipelines without delay.

Note that the accumulator scoreboard register is not a visible register.

Example 4–9. Four Double-Precision Floating-Point Accumulators

```
vmac.ssd r4, r7, 0, a0          ; r4 × r7 + 0 → a0
vmac.ssd r5, r8, a0, a0         ; r5 × r8 + a0 → a0
vmac.ssd r6, r9, a0, r10        ; r6 × r9 + a0 → r10
```

Cycle	Floating-Point Multiply			Floating-Point Add			
	Unpack	Multiply	Normalize	Align	Add	Normalize	
1	vmac1	— — —	— — —	— — —	— — —	— — —	
2	vmac2	vmac1	— — —	— — —	— — —	— — —	
3	vmac2	— — —	vmac1	— — —	— — —	— — —	
4	vmac2	— — —	— — —	vmac1	— — —	— — —	
5	vmac3	vmac2	— — —	— — —	vmac1	— — —	
6	vmac3	— — —	vmac2	— — —	— — —	vmac1	→ a0
7	vmac3	— — —	— — —	vmac2	— — —	— — —	
8	— — —	vmac3	— — —	— — —	vmac2	— — —	
9	— — —	— — —	vmac3	— — —	— — —	vmac2	→ a0
10	— — —	— — —	— — —	vmac3	— — —	— — —	
11	— — —	— — —	— — —	— — —	vmac3	— — —	
12	— — —	— — —	— — —	— — —	— — —	vmac3	→ r10

Note: Cycle value assumes no exceptions or any denormal values, and no instruction-cache or data-cache misses, contention, or interrupts.

4.4 Power-Up/Command Word Halt

The MP can be halted for two reasons:

- At power-up reset, $\overline{\text{HREQ}}$ high causes the MP to halt after the reset sequence (see Section 9.4, *Power-Up Halt*).
- The MP can issue a command (cmnd) instruction to itself that has the reset bit, the halt bit, and the MP bit set. The halted MP can be restarted from an external host by a signal on the $\overline{\text{EINT3}}$ (external interrupt 3) pin. However, the MP halts only if it was originally powered-up halted at reset; otherwise, it keeps running (it performs as if it was powered-up running).

Chapter 5

Understanding the MP Memory Organization

This chapter describes how the MP accesses on-chip and external memory, MP parameter RAM, and illegal addresses. The chapter concludes with a discussion of round-robin scheduling of memory accesses by the crossbar arbitration logic.

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

Topics

5.1 Accessing On-Chip Memory

The first 32 megabytes of the MVP's memory address space (addresses below 0x0200 0000) are dedicated to on-chip locations. The MP, PPs, and transfer controller (TC) share identical memory maps, although the access rights of the PPs and TC are more restricted than the MP's.

The MP can directly access all of the following locations with its load and store instructions:

- The MP's parameter RAM
- The PP's data RAMs and parameter RAMs
- The TC's memory-mapped registers
- The VC's (video controller's) memory-mapped registers

The TC can access all of the on-chip data and parameter RAMs but not memory-mapped registers in the TC and VC. Each PP can access not only its own data and parameter RAMs, but also the data and parameter RAMs belonging to the other PPs. On the other hand, a PP cannot directly access the MP's parameter RAM or the memory-mapped registers in the TC and VC.

The MP does not cache its on-chip data accesses. The MP always reads and writes its on-chip RAM and memory-mapped registers directly, bypassing the data cache.

To support testing and emulation, the MP can execute code from on-chip RAM. Instructions read from on-chip memory are cached. However, if you are concerned about code compatibility with future MVP devices, you should avoid executing code from on-chip memory.

5.1.1 Accessing On-Chip Static RAM (Except Data Caches)

The time required to access an on-chip SRAM bank is one clock cycle if there is no contention.

- MP loads can access the following:
  - any PP's data RAMs
  - any PP's parameter RAMs
  - the MP's parameter RAM
  - the MP's data caches†
  - any cache in the emulation mode†
- MP stores can access the following:
  - any PP's data RAMs
  - any PP's parameter RAMs
  - the MP's parameter RAM (only in the MP supervisor mode)
  - The MP's data caches (only in the MP supervisor mode)†
  - any cache in the emulation mode†

† The static RAM storage in the MP's data and instruction caches is mapped into fixed on-chip locations in order to facilitate testing and emulation. Applications software should never access these locations.

Example 5–1 contains several MP assembly code fragments that use load and store instructions to access on-chip memory locations.

Example 5–1. Load and Store Instructions That Access On-Chip RAM and Register Locations

```
        .global   PP2Start           ; PP2's program entry label
        .global   MPPTRFault         ; MP's packet transfer fault routine
        .global   MPIntOvf           ; MP's integer overflow routine

        addu      PP2Start,r0,r9     ; r9 = pointer to PP2's program
        st        0x010021B8(r0),r9  ; write to PP2's task interrupt
                                     ;   vector in PP2's parameter RAM
        . . .

        ld        0x0182000c(r0),r8  ; read TC's FLTSTS register
        bbo.a     MPPTRFault,r8,0    ; FLTSTS bit 0 is the MP packet
                                     ;   transfer request fault bit
        . . .

stMPPram:
        addu      MPIntOvf,r0,r7     ; MP's integer overflow entry
        st        0x010101BC(r0),r7  ; write to integer overflow vector in
                                     ;   MP's parameter RAM
```

Notes: 1) The MP must be in supervisor mode to store into MP parameter RAM (see Section 9.6, *Supervisor and User Modes*, for more information about the supervisor mode).

2) The bbo instruction is a branch if the selected bit is 1.

For preproduction silicon versions 1 and 2, the integer overflow bit is bit 4 of the INTPEN and IE registers and uses MP parameter RAM location 0x0101 0190 (not 0x0101 01BC, as shown in Example 5–1).

5.1.2 Accessing TC and VC On-Chip Registers

Section NO TAG, *Transfer Controller Registers*, in the *MVP Transfer Controller User's Guide*, and Chapter NO TAG, *Frame Timer Registers*, in the *MVP Video Controller User's Guide*, describe the transfer controller and video controller on-chip registers, respectively. These registers are mapped into the MP's memory address space. Example 5–2 shows how the MP accesses these registers with its load and store instructions.

Example 5–2. Accessing TC/VC On-Chip Registers

```
          ld     0x0182000c(r0),r5  ; r5 = TC's FLTSTS register
          bbo.a  PP1bp,r5,17        ; bit 17 = PP1's packet transfer
                                    ;  error
          ...

PP1bp:    or     0x20000,r0,r7      ; r7 = PP1's packet request error bit
          st     0x0182000c(r0),r7  ; writing a 1 to bit 17 turns off
                                    ;  FLTSTS bit 17

          or     0x4000,r0,r8       ; r8 = MP's INTPEN bit mf
          wrcr   INTPEN,r8          ; turn off INTPEN's mf bit

; This will cause the TC to retry PP1's packet request. If the error
; persists, then PP1 must be reset and this task discarded. Once the
; program error has been fixed, the task may then be resubmitted.

VC1sof:   or     VC1SOF,r0,r5       ; r5 = VC1's SOF linked list address
          st     0x010100e8(r0),r5  ; VC1 SOF transfer address in MP PRAM

          ld     0x01820340(r0),r6  ; r6 = VC1's FMEMCTL1 register
          or     0x1000,r6,r6       ; P=1 for VC1 initiated packet
                                    ;  requests
          st     0x01820340(r0),r6  ; store new VC1's FMEMCTL1
          . . .

Xint_On:  rdcr   CONFIG,r7          ; r7 = MP's CONFIG control register
          or     0x08000000,r7,r7   ; CONFIG's X bit (27) is set to 1
          wrcr   CONFIG,r7          ;  which enables externally initiated
                                    ;  packet transfers
```

Note: The program must be in the supervisor mode in order to write to the MP parameter RAM, and to control register numbers < 0x4000.

Note:

As a general rule, when reading or writing to a TC or VC register, you should specify the width of the access to match the width of the target register (16 or 32 bits). The following restrictions apply to load (ld and dld) and store (st and dst) instructions that access the MVP's on-chip memory-mapped registers:

- 32-bit stores are required to all 32-bit TC and VC registers.
- 16-bit stores are permitted only to 16-bit VC registers. Also, you can use 32-bit stores to access a pair of 16-bit VC registers.
- 64-bit loads are not supported from any TC or VC register. Likewise, 8-bit and 64-bit stores to any register are not supported.

5.2 Accessing Control Registers

The MP's control registers are not memory-mapped. They reside in a control-register address space that is separate from the memory address space. Control register numbers range from 0x0000 to 0x4002. For example, the ILRU register number (control register number 0x300) is **not** the same as on-chip RAM location 0x300 (PP0's data RAM bank 0's location 0x300).

Table 2–1, *MP Control Register Numbers*, gives the names and addresses of the accessible registers required for branch, read, swap, or write-control register instructions (brcr, rdcr, swcr, or wrcr, respectively). For a complete description of the control registers, see Chapter 3, *Master Processor Control Registers*.

Example 5–3 shows several control register accesses.

Example 5–3. Read or Write Control Registers

```
        .global   MPPRAdd            ; address of MP's packet transfer

        wrcr      IN0P,r2            ; r2 is input vector's address
        wrcr      IN1P,r4            ; r4 is input matrix's address
        wrcr      OUTP,r6            ; r6 is output vector's address

        rdcr      INTPEN,r9          ; r9 = interrupt pending register
        bbo.a     MPself,r9,25       ; r9's bit 25 = 1 is the MP self
                                     ;   interrupt
        . . .

        rdcr      IE,r8              ; r8 = interrupt enable register
        or        0x100,r8,r8        ; bit 8 is frame 0 timer interrupt
        wrcr      IE,r8              ; enables frame 0 timer interrupt
        . . .

Spin:   rdcr      PKTREQ,r7          ; r7 = packet transfer register
        bbo.a     Spin,r7,1          ; spin if bit 1=1 , for example, if
                                     ;   packet transfer queue is busy
        addu      MPPRAdd,r0,r6      ; MP's packet transfer address
        st        0x10100fc(r0),r6   ; link list pointer for MP packet
                                     ;   transfers
        or        0x01,r0,r7         ; bit 0 = 1 submits packet transfer
                                     ;   request
        wrcr      PKTREQ,r7          ; issue MP's packet transfer request
                                     ;   to the TC
        . . .

MPself:
```

Notes: 1) When control-register numbers are less than 0x4000, the MP must be in supervisor mode for swap (swcr) and write (wrcr) instructions, as shown in Table 2–1, *MP Control Register Numbers*.

2) The bbo instruction causes a branch to occur if the value of the selected bit is 1.

5.3 Accessing PP Data RAMs (Not Cached)

The MP accesses the PPs' on-chip data RAMs through the crossbar. Information in PP data RAMs is accessed directly and is not cached. Any data access to the RAMs by the MP leaves the data-cache tag (DTAG), data LRU (least recently used), dirty bit, and present information in the MP's data cache unchanged.

To support test and diagnostic software, the MVP hardware allows you to execute code from the on-chip data and parameter RAMs. These instruction accesses go through the MP's instruction cache. On-chip code executes somewhat more slowly than code in external-memory code due to the longer time required to update the instruction cache. Application code should avoid executing code from the on-chip RAM in order to avoid incompatibilities with future MVP devices, which may not support code execution from the on-chip RAM.

5.4 Accessing Parameter RAMs (Not Cached)

Each PP accesses its parameter RAM bank using its global or local bus. The on-chip parameter RAMs are **not** cached but are loaded and stored using packet transfers serviced by the transfer controller, or by MP or PP load or store instructions.

The MP accesses the on-chip parameter RAMs (MP or PP) via the crossbar. These RAMs are mapped into the lower addresses of the memory space (see bank addresses in NO TAG, *MVP Memory Map*, in the *MVP System-Level Synopsis*). Any data access to these RAMs by the MP leaves the data cache tag, LRU (least recently used), dirty, and present information in the MP's data cache unchanged.

The MP waits for contention with the TC and the PPs to be resolved by arbitration, as discussed in subsection 5.8.1.

The MP can always read its own parameter RAM but can only write to it when the MP is in the supervisor mode (see Section 9.6, *Supervisor and User Modes*).

The different types of parameter RAMs and their on-chip addresses for a four-PP MVP are shown in NO TAG, *MVP Memory Map* in the *MVP System-Level Synopsis*.

To support test and diagnostic software, the MVP hardware allows you to execute code from the on-chip data and parameter RAMs. These instruction accesses go through the MP's instruction cache. On-chip code executes somewhat more slowly than code in external-memory code due to the longer time required to update the instruction cache. Application code should avoid executing code from the on-chip RAM in order to avoid incompatibilities with future MVP devices, which may not support code execution from the on-chip RAM.

5.4.1 Accessing PP Parameter RAMs

Each PP parameter RAM is shared between the local PP and the TC. The TC uses portions of the RAM to buffer packet transfers requested by the PP. The parameter RAM also contains the interrupt vectors required to service this PP's interrupts and traps. The PP's system stack is a part of this PP's parameter RAM; at power-up reset, PP stack pointer is initially set to 0x010 0#7F0 (unless the pointer is reset by the software) and grows toward lower addresses (# is the local PP number). A section of the parameter RAM is available for general purpose use such as packet transfer address pointers, task parameter passing, or interprocessor command messages. The general-purpose area of this on-chip parameter RAM is not cached but is loaded and stored using packet transfer requests to the transfer controller, or by PP or MP load or store instructions.

The PP's parameter RAM is shown in Figure 5–1 and has the following regions:

- 128 bytes for suspended packet transfer parameters for this PP,
- 96 bytes reserved for future expansion,
- 24 bytes reserved for operating-system-specific software parameters,
- 4 bytes for the load/store (DEA or on-chip) or instruction-cache fault address for this PP,
- 4 bytes for the starting address of this PP's linked-list packet transfer,
- 128-byte buffer for external-to-external packet transfers,
- 128 bytes for 32 interrupt vectors,
- 1524 bytes for general-purpose use (PP stack starts at 0x0100 #07F0 and grows toward lower addresses),
- 12 bytes for stacked state information after reset (see Chapter NO TAG, *Interrupts and Reset*, in the *MVP Parallel Processor User's Guide*).

Note:

Do not execute MP programs from the PP parameter RAMs; future versions of the MVP may not support this option.

Figure 5–1. PP Parameter RAM Contents

Base Address		Size
0×0100 #000	Suspended Packet Transfer Parameters	128 bytes
0×0100 #080	Reserved	96 bytes
0×0100 #0E0	Operating System Parameters	24 bytes
0×0100 #0F8	Cache Fault Address for This PP	4 bytes
0×0100 #0FC	Packet Transfer Linked-List Address	4 bytes
0×0100 #100	Buffer for External-External Transfers	128 bytes
0×0100 #180	Interrupt Vector Transfer Addresses	128 bytes
0×0100 #200	General-Purpose RAM	1524 bytes
0×0100 #7F4	Stacked State Information After Reset	12 bytes

Notes: 1) # is the PP number.

2) The TC is the primary user of parameter RAM addresses in the range 0x0100 #000 to 0x0100 #17F, but the software running on the PP may need to access these parameters.

5.4.2 Accessing MP Parameter RAM

MP parameter RAM is shared between the MP and the TC and is used as a buffer for TC tasks that are dedicated to the MP. The parameter RAM also contains the interrupt vectors required to service MP interrupts and traps.

A section of the MP parameter RAM is available for general purposes, such as passing address pointers, passing task parameters, or sending interprocessor command messages. The general-purpose area of the on-chip MP parameter RAM is not cached but is accessed by the transfer controller during packet transfers, or by MP load and store instructions (the MP must be in supervisor mode to use store, as discussed in Section 9.6, *Supervisor and User Modes*).

The MP parameter RAM is shown in Figure 5–2 and has the following regions:

- 128 bytes for suspended MP packet transfer parameters,
- 96 bytes reserved for future expansion,
- 28 bytes for seven linked-list starting addresses of externally or VC frame timer-initiated packet transfers, as shown in Figure 5–3 (packet transfer numbers 1 through 7),
- 4 bytes for the starting address of the MP's linked-list packet transfer,
- 128 bytes for MP external-to-external transfer buffer,
- 128 bytes for 32 MP interrupt vectors,
- 32 bytes for eight MP trap vectors,
- 128-byte buffer for the externally or VC frame timer-initiated packet request transfer buffer (see CONFIG[X] in Figure 3–7, *MP Configuration Register—CONFIG*), and
- 1376 bytes for general purposes (including additional trap transfer addresses).

Figure 5–2. MP Parameter RAM Contents

Base Address		Size
0×0101 0000	Suspended Packet Transfer Parameters	128 bytes
0×0101 0080	Reserved	96 bytes
0×0101 00E0	7 External Linked-List PT Addresses	28 bytes
0×0101 00FC	Packet Request Linked-List Address	4 bytes
0×0101 0100	Buffer for External-External Transfers	128 bytes
0×0101 0180	Interrupt Vector Transfer Addresses	128 bytes
0×0101 0200	Trap Transfer Addresses	32 bytes
0×0101 0220	Externally Initiated PT Transfer Buffer	128 bytes
0×0101 02A0	General-Purpose RAM	1376 bytes

Notes: 1) PT = Packet transfer

2) The TC uses 0x0101 0000 through 0x0101 017F and optionally 0x0101 0220 through 0x0101 029F; the operating system and the program use the rest.

3) Do **not** use trap 40–71 instructions when externally initiated packet transfers are enabled by CONFIG⟦X⟧ (see Figure 3–7, *MP Configuration Register—CONFIG*) and FMEMCTL⟦P⟧ (see Figure 2–13, *Video Controller Memory Control Register—FMEMCTL 0/1*); these use the 0x0101 0220 through 0x0101 029F buffer in the MP parameter RAM. Traps are explained in Section 9.2, *Managing Exceptions*.

Figure 5–3. Externally Initiated Packet Transfer Linked-List Address Pointers

Address	Externally Initiated Packet Transfer	VC-Initiated Packet Transfer	
0×0101 00E0	External Packet Transfer 7	VC Start of Field 0	FMEMCTL0⟦P⟧ = 1
0×0101 00E4	External Packet Transfer 6	VC SAM Overflow 0	
0×0101 00E8	External Packet Transfer 5	VC Start of Field 1	FMEMCTL1⟦P⟧ = 1
0×0101 00EC	External Packet Transfer 4	VC SAM Overflow 1	
0×0101 00F0	External Packet Transfer 3		
0×0101 00F4	External Packet Transfer 2		
0×0101 00F8	External Packet Transfer 1		

CONFIG⟦X⟧ = 1

Notes: 1) SAM = Serial access memory

2) Externally initiated packet transfers require that CONFIG⟦X⟧ = 1 (see Figure 3–7, *MP Configuration Register—CONFIG*).

3) VC-initiated packet transfers require that FMEMCTL0/1⟦P⟧ = 1 (see Figure 2–13, *Video Controller Memory Control Register—FMEMCTL 0/1*).

Preproduction silicon version 2 and later have the externally initiated packet transfer linked-list pointers (Figure 5–3) and the externally initiated packet transfer buffer (Figure 5–2) implemented; earlier versions do not.

5.5 Accessing MP External Memory

The MP's main off-chip data and instruction areas are within the 4064 megabytes of external memory. Information in this area is cached by the MP's instruction and data caches. Addresses of these caches are shown in NO TAG, *MVP Memory Map*, in the *MVP System-Level Synopsis*.

Note:

Following a reset, the last two words in external memory branch to the first program location (for example, br.a Program_Start).

- 0xFFFF FFF8—annulled branch instruction after reset
- 0xFFFF FFFC—contains the first location of new program as a long immediate address

5.5.1 Direct External Memory Access (DEA)

The MP typically uses its ld and st (load and store) instructions to access data in memory. These accesses are made through the MP's data cache. However, on occasion, the MP may need to use its dld and dst instructions to directly access a word in external memory, bypassing the data cache. These accesses, called DEAs (direct external accesses), are useful for avoiding cache interference when communicating with external processors or peripheral devices.

Any MP access of a memory address below 0x0200 0000 that does not point to a legal on-chip SRAM or register location is converted to a DEA request to be serviced by the TC. The TC enforces all illegal addresses and reports them to the MP as memory faults.

The DEA carries some of the same overhead as a packet transfer request and thus should be used carefully. In particular, using the dld and dst instructions to access several contiguous words in external memory is much slower than accessing them with ld and st.

While accesses to off-chip memory are slower than accesses to on-chip memory, the MP can use the dld/dst (direct load/store) instructions to access off-chip memory as a high-priority request to the transfer controller, as shown in NO TAG, *Transfer Controller's Request Prioritization,* in the *MVP Transfer Controller User's Guide*.

Table 5–1 shows the minimum DEA latency for the TC to access various types of dynamic RAM in external memory. These minimum times assume that the TC is immediately available to service the DEA request. The latency indicates the minimum number of TC cycles that are required to service the DEA request.

Table 5–1. Minimum Transfer Controller DEA Latency for the MP

MP DEA Access Type	Memory Type	Number of Cycles	
		Page Hit	Page Miss
Store	Any type of memory	8	12
Load	1 cycle per column, nonpipelined	11	15
	1 cycle per column, pipelined	12	16
	2 cycles per column	11	16
	3 cycles per column	12	18

The TC latency for a DEA store is less than that for a DEA load:

- For an MP direct load, the MP destination register is scoreboarded until the TC completes the load operation and the data value is written into that register.
- For an MP direct store, the MP source register data and the data address are made available to the TC at the end of the memory access pipeline phase. Then the TC can begin its DEA write operation after the memory access pipeline phase.

When the direct load/store is completed, the TC resumes other operations. If the TC cannot complete the direct load/store operation, a nonmaskable memory fault (mf) interrupt is taken by the MP, and the faulted address is stored in the FLTADR register. The mf interrupt service routine can examine the FLTOP register to determine the type of MP memory fault.

DEAs are generally used only when very little data (usually one or two accesses) is being transferred. In most other cases, it is more efficient to use a packet transfer request to transfer data between on-chip and off-chip memory. Often a packet transfer request can be submitted before the data is actually required. Then, the MP can verify that the packet transfer is done and the data is available without having the pipeline stalled for a register scoreboard delay. Packet transfers are described in more detail in Chapter 7, *Understanding the Packet Transfer Request Protocol*, in this user's guide and in Chapter NO TAG, *Packet Transfers*, in the *MVP Transfer Controller User's Guide.*

You can also move individual 64-byte subblocks between external memory and the data cache. The dcachef and dcachec instructions give MP programs explicit control over data-cache operations.

5.6 Accessing Illegal Addresses

When accessing data in on-chip memory, the MP's two types of load (ld and dld) and store (st and dst) instructions behave identically. The MP performs these accesses itself, without TC intervention. When the MP does not recognize the target address of a load/store instruction as a valid on-chip address, it automatically passes the request to the TC for servicing. The TC is responsible for performing DEAs, cache servicing, and illegal-address detection for the MP. By moving all responsibility for illegal-address detection from the MP to the TC, on- and off-chip illegal addresses can be handled together in a uniform manner.

5.6.1 MP Memory Fault Service Routine Outline

The MP has two types of memory faults:

- Nonmaskable faults, which do not set the INTPEN⟦mf⟧ bit and include:
  - MP instruction cache
  - MP data cache
  - MP load/store (DEA or on-chip) faults
- Maskable faults, which set the INTPEN⟦mf⟧ bit and include:
  - PP instruction cache
  - PP load/store (DEA or on-chip) faults
  - MP packet request
  - PP packet request
  - Externally or VC frame timer-initiated packet request

The nonmaskable interrupt is always taken, even if global interrupts are not enabled (IE⟦ie⟧ = 0) or memory faults are not enabled (IE⟦mf⟧ = 0). Maskable interrupts are posted in INTPEN⟦mf⟧ and an interrupt is taken only if IE⟦ie⟧ = 1 and IE⟦mf⟧ = 1; otherwise, the program must poll INTPEN⟦mf⟧ for service requests.

Note:

If the MP is in the emulator mode, then any legal MVP on-chip address is accessible.

The following text shows the outline of a memory fault service routine.

❑ Test the i, d, and x bits in the FLTOP register to determine whether a nonmaskable memory-fault interrupt has occurred. A value of 1 in one of these bits indicates a nonmaskable fault. After handling the fault, the fault service routine should restore these bits to 0. (The FLTOP register is cleared to 0 at power-up reset.) The fault service routine performs the following steps in any order:

- Test FLTOP⟦i⟧ = 1 for MP instruction-cache fault. If FLTOP⟦i⟧ = 1, then call the **MP–I–Cache–Fault** routine with argument EPC (contains the address of the memory fault).
- Test FLTOP⟦d⟧ = 1 for MP Data–cache fault. If FLTOP⟦d⟧ =1, then call the **MP–D–Cache–Fault** routine with argument FLTADR (contains the address of the memory fault).
- Test FLTOP⟦x⟧ = 1 for MP load/store (DEA or on-chip) memory fault. If FLTOP⟦x⟧ = 1, then call the **MP–Ld–St–DEA–Fault** routine with argument FLTADR (contains the address of the memory fault).

A nonmaskable memory fault may be fatal. If so, call the **Fatal–Exit** routine; otherwise, perform a normal return or else check for maskable memory faults.

❑ If INTPEN⟦mf⟧ = 1, then a maskable PP or TC memory fault interrupt has occurred. The fault service routine performs the following steps (PP tests should include all PPs that are on the MVP chip) in any order:

- Test FLTSTS⟦m⟧ = 1 for an MP packet transfer fault as detected by TC. If FLTSTS⟦m⟧ = 1, then call the **MP–Packet–Request–Fault** routine with argument 0x0101 0000 (address of the MP packet request suspend area).
- Test FLTSTS⟦XPT⟧ for externally or a VC frame timer-initiated packet request fault as detected by the TC. If FLTSTS⟦XPT⟧ is nonzero, then call **XPT–Packet–Request–Fault** routine with argument XPT (XPT indicates which one of the seven XPT faults occurred).
- Test FLTSTS⟦p⟧ = 1 for a PP packet transfer fault as detected by TC. If FLTSTS⟦p⟧ = 1, then call the **PP–Packet–Request–Fault** routine with argument 0x0100 #000 (the address of the PP# packet request suspend area).

- Test FLTSTS[x] for a PP instruction–cache or load/store (DEA or on-chip) memory fault as detected by the TC. If FLTSTS[x] is nonzero, then test the corresponding PP# bit in PPERROR[f] to determine which type of PP memory fault occurred (0 = PP instruction cache, or 1 = PP load/store (DEA or on-chip) in the PPERROR register).
  - If FLTSTS[x] = 1, then test PPERROR[f] = 1 for PP load/store (DEA or on-chip) memory fault. If PPERROR[f] is nonzero, then **PP–Ld–St–DEA–Fault** routine is called with argument 0x0100 #0F8 (contains the address of the PP# load/store (DEA or on-chip) memory fault).
  - If FLTSTS[x] = 1, then test PPERROR[f] = 0 for PP instruction-cache memory fault. If PPERROR[f] is zero, then call **PP–I–Cache–Fault** routine with argument 0x0100 #0F8 (contains the address of the PP# instruction-cache memory fault).

- If recovery is possible, then a normal return is taken. If recovery is not possible (the memory fault is fatal), then call the **Fatal–Exit** routine.

Maskable and nonmaskable faults share the interrupt vector at address 0x0101 01B8 in the MP's parameter RAM. This means that the fault routine pointed to by this vector should be capable of handling both maskable and nonmaskable faults.

The fault routine must not itself cause a nonmaskable fault, which would immediately interrupt the fault routine and make recovery impossible.

A maskable fault that is not cleared or disabled by the fault routine causes the fault interrupt to be taken again upon return to the interrupted program. However, a nonmaskable fault interrupt is not repeated if the fault routine returns without processing the fault.

5.6.2 Nonmaskable Memory Faults

A memory fault is nonmaskable if it occurs as the result of an MP instruction fetch or a data access performed by a load (ld or dld) or store (st or dst) instruction. These are faults that occur when the TC is doing one of the following:

- Servicing the MP's instruction cache or data cache
- A noncached data access requested by the MP

The second item above refers to a load or store instruction that bypasses the MP's data cache. Included in this category are both DEAs and accesses of on-chip memory addresses. An attempt to access an address below 0x0200 0000 that is not recognized as valid by the MP is submitted as an access request to the TC, which reports the fault.

The fault routine checks for nonmaskable faults by testing the FLTOP register's i, d, and x bits, which indicate whether instruction-cache, data-cache, and noncached-data-access faults, respectively, have occurred. Additional information is available from other registers. For example, in the case of an instruction-cache fault, the EPC register contains the address of the instruction that the MP was trying to fetch when the fault occurred. For data-cache and noncached-data-access faults, the additional information is recorded in the FLTTAG, FLTADR, FLTDTH, and FLTDTL registers. Note that these four registers are not modified by instruction-cache faults.

The fault routine should be prepared for the possibility that an instruction-cache fault occurs at the same time as either a data-cache or noncached-data-access fault. However, data-cache and noncached-data-access faults cannot occur simultaneously.

While some systems may choose to treat nonmaskable faults as fatal errors, other systems may attempt recovery from these faults. For example, in a system that supports virtual memory, the handling of nonmaskable faults may entail swapping pages between the MVP's external memory and mass storage and then continuing program execution. In the event of an instruction-cache fault, a return-from-interrupt sequence is sufficient to retry the faulting instruction. However, if a data-cache or DEA fault has occurred, retrying the faulting instruction may not be feasible due to the MP's scoreboarding mechanism, which may have allowed execution to continue past the instruction that actually caused the fault. In this case, recovery may require updating the data cache, registers, and external memory to the values they would have contained if the faulting access had completed normally; the information necessary to make these updates is contained in the FLTOP, FLTTAG, FLTADR, FLTDTH, and FLTDTL registers.

5.6.3 Maskable Memory Faults

The maskable memory faults occur during PP accesses and by TC accesses that are not directly part of the MP's program execution. A maskable fault causes INTPEN[mf] to be set to 1, but the interrupt is taken only if it is enabled by the IE register's mf and ie bits. A maskable fault can occur when the TC is performing one of the following:

- A packet transfer requested by the MP, a PP, or an external device (through the externally initiated packet transfer interface)
- Instruction-cache service or a load/store access requested by a PP

Note that a PP's illegal load/store accesses in the on-chip memory address range are handled similarly to the MP's. In other words, an attempt by a PP load or store instruction to access data at an address below 0x0200 0000 that the PP does not recognize as valid is converted to a request to the TC. The TC then reports the fault.

The information needed to identify the source of a maskable fault is recorded in the TC's FLTSTS register. FLTSTS bit 0 is set to indicate that a fault occurred during a packet transfer requested by the MP. FLTSTS bits 16–19 indicate which, if any, PPs' packet transfer requests have caused faults.

FLTSTS bits 24–27 indicate which PPs' cache-service or load/store requests have caused faults. The information in these bits is further qualified by the four fault bits in the PPERROR register, which indicate whether the fault occurred during cache servicing or a load/store access.

When a fault occurs during an externally requested packet transfer, FLTSTS bits 5–7 record the nonzero number of the externally initiated packet transfer (XPT) that caused the fault. XPT numbers range from 1 to 7. Note that the XPT field in the FLTSTS register is encoded because only one XPT fault can occur at a time. Until an XPT fault is cleared, it prevents any further XPT requests from being serviced. The other FLTSTS bits are not encoded in order to allow more than one of these faults to accumulate before interrupt servicing begins.

The memory-fault flag, INTPEN⟦mf⟧, represents the logical-OR of all of the bits in FLTSTS. As long as FLTSTS is nonzero, INTPEN⟦mf⟧ is driven to 1 on every clock cycle. The fault routine is unable to clear INTPEN⟦mf⟧ until it first clears FLTSTS. If the fault routine returns with INTPEN⟦mf⟧ set because either FLTSTS was not properly cleared or another fault occurred after it was cleared, another fault interrupt is taken as soon as interrupts are enabled.

You clear a FLTSTS bit by writing a 1 to it; writing a 0 has no effect. Also, writing a 1 to a bit that is already 0 has no effect. FLTSTS bits are set only by fault conditions—you cannot change a FLTSTS bit from 0 to 1 by writing to it.

The bits in FLTSTS, when set to nonzero values, act to inhibit the faulting requests to which they correspond. For instance, while FLTSTS bit 0, which indicates an MP packet transfer fault, contains the value 1, the MP's packet transfer request to the TC is blocked. Clearing this bit unblocks the request and causes the TC to resume the transfer. Similarly, FLTSTS bits 16–19 can block packet transfer requests from the PPs to the TC; the corresponding requests are unblocked when these bits are cleared. Also, FLTSTS bits 24–27 can block requests from the PPs for cache service or for TC-assisted load/store accesses, but the requests are unblocked when these bits are cleared. Clearing these bits unstalls the faulting PPs as well.

However, the XPT field in the FLTSTS register operates somewhat differently. As long as the 3-bit XPT-fault code in FLTSTS bits 5–7 remains nonzero, all XPT requests are blocked. Clearing the XPT field causes the TC to begin accepting new XPT requests, but the XPT that was in progress when the fault occurred cannot be resumed and will have to be completely restarted. (Unlike MP and PP packet transfers, XPTs do not resume from the point at which the fault occurred.)

If a faulting request should not be resumed, clear the request at its source before clearing the corresponding FLTSTS bit. Cache-service and DEA requests from a PP can be cleared only by resetting the PP. A packet transfer request from a PP or the MP can be cleared by resetting the processor or by suspending the request (by setting PKTREQ⟦S⟧). XPT requests can be cleared either by resetting the external devices that generate the request signals to the MVP or by disabling the requests from the VC.

When a packet transfer requested by the MP or a PP faults, the TC saves the parameters representing the state of the packet transfer in a buffer at the beginning of the requesting processor's parameter RAM. The linked-list pointer in that processor's parameter RAM is modified to point at the saved parameters. Information such as the faulted address can be determined from the saved parameters.

The fault handler should be prepared for the possibility that a PP may have both a packet transfer request fault and either a cache-service or load/store fault pending at the same time. Cache-service and load/store faults, however, cannot occur simultaneously in the same PP.

When a PP cache-service or DEA fault occurs, the faulted address can be found in the fault address (0x0100 #0F8, where # is the PP number) in the PP's parameter RAM. The fault routine can read the four fault bits in PPERROR to determine whether an instruction-cache service request or data load/store caused the fault (0=cache, 1=load/store). PPERROR is a read-only register and is updated on a clock-by-clock basis to indicate the status of the PPs. When a PP is running normally, that PP's fault bit in PPERROR alternates between 0 and 1 to reflect changes in the type of access being performed. When the PP causes a cache or load/store fault, it stalls and its PPERROR fault bit remains frozen at the type of access that caused the fault.

5.6.4 MP Supervisor Mode

The MP can access most of the MVP's on-chip static RAM in the supervisor mode. Table 5–2 shows which RAM areas the MP is permitted to access in the supervisor mode.

Table 5–2. MP Memory Access in the Supervisory Mode

Memory Address	MP ld, dld, vld	MP st, dst, vst	Comments
PP parameter RAMs	OK	OK	
PP data RAMs	OK	OK	
PP instruction cache	fault	fault	Nonmaskable memory fault; FLTADR = address
MP parameter RAM	OK	OK	
MP instruction cache	fault	fault	Nonmaskable memory fault; FLTADR = address
MP data cache	OK†	OK†	No error, but not recommended
TC memory-mapped registers	OK	OK	
VC memory-mapped registers	OK	OK	
On-chip invalid memory	fault	fault	Nonmaskable memory fault; FLTADR = address
Off-chip valid memory	OK	OK	
Off-chip invalid memory	fault	fault	Nonmaskable memory fault; FLTADR = address

† Not recommended. This entry does not refer to the normal method of accessing memory data through a cache, but rather to direct access of the on-chip RAM that the cache uses for data storage. Direct access of cache RAM is provided to support testing and diagnostics but should not be used by applications code.

5.6.5 MP User Mode

The user mode cannot write to MP parameter RAM (or MP data cache) directly or to control registers < 0x4000. Table 5–3 shows which RAM access the MP is permitted to access in user mode. The only significant difference between Table 5–3 and Table 5–2 is that only supervisor-mode programs can write to the MP's parameter RAM.

Table 5–3. MP Memory Access in the User Mode

Memory Address	MP ld, dld, vld	MP st, dst, vst	Comments
PP parameter RAMs	OK	OK	
PP data RAMs	OK	OK	
PP instruction cache	fault	fault	Nonmaskable memory fault; FLTADR = address
MP parameter RAM	OK	fault‡	Write is nonmaskable memory fault; FLTADR = address
MP instruction cache	fault	fault	Nonmaskable memory fault; FLTADR = address
MP data cache	OK†	fault‡	Write is nonmaskable memory fault; FLTADR = address
TC memory-mapped registers	OK	OK	
VC memory-mapped registers	OK	OK	
On-chip invalid memory	fault	fault	Nonmaskable memory fault; FLTADR = address
Off-chip valid memory	OK	OK	
Off-chip invalid memory	fault	fault	Nonmaskable memory fault; FLTADR = address

† Not recommended. This entry does not refer to the normal method of accessing memory data through a cache, but rather to direct access of the on-chip RAM that the cache uses for data storage. Direct access of cache RAM is provided to support testing and diagnostics but should not be used by applications code.

‡ Only difference from MP supervisor mode

5.6.6 MP Instruction-Cache Sources

To support testing, diagnostics, and emulation software, the MP can execute instructions from on-chip RAM. This feature is not recommended for use by applications code. It may not be supported in future implementations of the MVP architecture.

The MP can access any PP's data or parameter RAMs as instruction-cache source. The TC moves the 16 words from the source address to the MP instruction cache. Table 5–4 shows which RAM areas the MP instruction-cache service can access.

Table 5–4. Source Memory Access for MP Instruction-Cache Service

Memory Address	MP Instruction Cache
PP parameter RAMs	OK
PP data RAMs	OK
PP instruction cache	fault†
MP parameter RAM	fault†
MP instruction cache	fault†
MP data cache	fault†
TC memory-mapped registers	fault†
VC memory-mapped registers	fault†
On-chip invalid memory	fault†
Off-chip valid memory	OK
Off-chip invalid memory	fault†

† Nonmaskable memory fault; FLTOP[[i]] = 1; EPC = address

5.6.7 PP Instruction-Cache Sources

To support testing, diagnostics, and emulation software, the PP can execute instructions from on-chip RAM. This feature is not recommended for use by applications code. Execution of code from on-chip RAM is slower than from external memory. Also, this feature may not be supported in future implementations of the MVP architecture.

The PP can access any PP's data or parameter RAMs as instruction-cache source. The TC moves the 16 instructions from the source address to the PP instruction cache. If this PP instruction-cache fault occurs, the PP is stalled until the fault is fixed; otherwise, the MP issues a reset command (with the cmnd instruction) to the PP to abandon this task. Table 5–5 shows which RAM areas to PP instruction-cache service can access.

Table 5–5. Source Memory Access for PP Instruction-Cache Service

Memory Address	PP Instruction Cache
PP parameter RAMs	OK
PP data RAMs	OK
PP instruction cache	fault†
MP parameter RAM	fault†
MP instruction cache	fault†
MP data cache	fault†
TC memory-mapped registers	fault†
VC memory-mapped registers	fault†
On-chip invalid memory	fault†
Off-chip valid memory	OK
Off-chip invalid memory	fault†

† Maskable memory fault; FLTSTS⟦x⟧ = 1; PPERROR⟦f⟧ = 0; 0x0100 0#0F8 = address; INTPEN⟦mf⟧ = 1; the PP stalls, but no signal is sent to the PP's intflg register.

5.6.8 MP Packet Transfer Requests

The TC can access most (but not all) of the MVP's on-chip static RAM during a packet transfer that was requested by the MP. The TC suspends faulty packet requests and places the address of the suspended parameters in the linked-list location 0x0101 00FC. Information about the error is found in the OPTIONS word in the suspend area in field PTS (see subsection NO TAG, *PT Status (PTS)—Bits 29–30*, in the *MVP Transfer Controller User's Guide*). Recovery is doubtful unless the program can correct the fault and resume the data transmission. Otherwise, the packet request must be discarded before further processing can be continued.

To discard a faulted packet transfer request:

1) Suspend the request by writing a 1 to the PKTREQ⟦S⟧ bit. Wait until PKTREQ⟦Q⟧ = 0.
2) Reset the PKTREQ⟦S⟧, PKTREQ⟦P⟧, FLTSTS⟦m⟧, and INTPEN⟦mf⟧ bits.
3) Do not resume the suspend packet transfer request in 0x0101 00FC.
4) Issue another packet transfer request in the normal manner (described in subsection 7.4.1, *Submitting a Packet Transfer Request*).

Table 5–6 shows which RAM areas the TC can access during an MP-requested packet transfer.

Table 5–6. Memory Access for MP Packet Requests

Src/Dst Memory Address	MP Packet Requests
PP parameter RAMs	OK
PP data RAMs	OK
PP instruction cache	fault†
MP parameter RAM	OK
MP instruction cache	fault†
MP data cache	fault†
TC memory-mapped registers	fault†
VC memory-mapped registers	fault†
On-chip invalid memory	fault†
Off-chip valid memory	OK
Off-chip invalid memory	fault†

† Maskable memory fault; FLTSTS⟦m⟧ = 1; 0x0101 00FC = 0x0101 0000 (suspended packet transfer request); INTPEN⟦mf⟧ = 1. The MP's PKTREQ⟦Q⟧ = 1 on suspended faulted packet transfer requests.

Note:

After an MP program requests a packet transfer, it should monitor the memory-fault interrupt, as well as the packet-complete interrupt. Otherwise, it risks waiting forever for a faulted packet transfer to complete. When an MP packet transfer request faults, neither INTPEN[[pc]] nor PKTREQ[[Q]] gives any indication of the fault: the packet-complete interrupt does not occur and the Q bit remains set. However, the fault does set FLTSTS[[m]] to 1, and this, in turn, sets INTPEN[[mf]] to 1.

For similar reasons, monitoring the bad-packet interrupt prevents the software from hanging in the event of an erroneous parameter value in a packet transfer request.

5.6.9 PP Packet Transfer Requests

The TC can access most (but not all) of the MVP's on-chip static RAM during a packet transfer that was requested by a PP. The TC suspends faulty packet requests and places the address in the linked-list location 0x0100 #0FC. The PP continues running (the PP stalls when there is a PP instruction-cache or DEA fault) and is made aware of a fault; no flag is set in the PP's intflg register to indicate the fault.

The MP (via its memory-fault interrupt routine) examines the PP's faulted packet request parameters (beginning at address 0x0100 #000) and determines a course of action according to the nature of the fault. Two general categories of faults are possible:

- The external memory's address decoding hardware has signaled a memory fault to the MVP. If recovery is possible, the MP arranges with the external system for the faulted address to be made available. Then the MP clears the appropriate x bit in the FLTSTS register, and the packet request is automatically resubmitted (because the PP's Q bit is still set to 1).
- A bug in the software caused a program to attempt to access an on-chip address that is illegal or invalid. The PP can be reset using a cmnd instruction. Debug information is available in the suspended packet request parameters and the current ipe, ipa, and stack pointer are pushed into the PP's parameter RAM. (Refer to subsection NO TAG, *Initial State Following Reset*, in the *MVP Parallel Processor User's Guide*.)

You can adapt the above routine for MP packet request faults and PP cache/DEA faults as appropriate.

Table 5–7 shows which RAM areas the TC can access during a PP-requested packet transfer.

Table 5–7. Memory Access for PP Packet Requests

Src/Dst Memory Address	PP Packet Requests
PP parameter RAMs	OK
PP data RAMs	OK
PP instruction cache	fault†
MP parameter RAM	fault†
MP instruction cache	fault†
MP data cache	fault†
TC memory-mapped registers	fault†
VC memory-mapped registers	fault†
On-chip invalid memory	fault†
Off-chip valid memory	OK
Off-chip invalid memory	fault†

† Maskable memory fault; FLTSTS⟦p⟧ = 1; 0x0100 #0FC = 0x0100 #000, where # = PP number (suspended packet transfer request); INTPEN⟦mf⟧ = 1. The PP's comm⟦Q⟧ = 1 on suspended faulted packet transfer requests.

Note:

When a PP requests a packet transfer that subsequently causes a memory fault, the PP receives no direct notification of the fault. In other words, the PP's intflg and comm registers give no indication that either the fault has occurred or that the packet transfer is no longer active. Instead, the TC notifies the MP of the fault and the MP's fault interrupt routine is responsible for handling the fault on behalf of the PP. If recovery is possible, the MP's fault routine resumes the packet transfer by clearing the PP's packet transfer fault bit in the FLTSTS register. However, if the fault is fatal, the fault routine may need to reset the PP to clear its packet transfer request to the TC.

5.6.10 Memory Error Reporting

During system development, the MP's memory fault mechanism can serve as a useful debugging feature. To track a problem down efficiently, you need as much information about the memory fault as possible. You can create a standard template in which to save as many details as can be determined about the fault. The interrupt service routine should save the following information to help you determine the nature of the fault:

- ❑ Error identification code
- ❑ MP's EPC and EIP registers
- ❑ MP's IE, INTPEN, CONFIG, and PPERROR registers
- ❑ MP's FLTOP, FLTTAG, FLTADR, FLTDTH, and FLTDTL registers
- ❑ TC's FLTSTS register
- ❑ VC's FMEMCTL0/1 registers
- ❑ MP's linked-list pointer for packet transfer requests and the suspended packet transfer request parameters from MP parameter RAM
- ❑ PP's linked-list pointer for packet transfer requests and the suspended packet transfer request parameters from PP parameter RAM
- ❑ PP's saved ipa, ipe, and sp after the PP has been reset
- ❑ Externally or VC frame timer-initiated packet transfer linked-list pointer and associated packet transfer request parameters

Then, as a part of the interrupt service routine, the MP can halt or reset itself by executing a cmnd instruction.

5.7 Big-Endian and Little-Endian Ordering

The MVP can operate in either little-endian or big-endian data format. Endian ordering is determined at power-up reset, which automatically sets the CONFIG⟦E⟧ bit to the correct endian value (from a hardware signal). Bit numbers for internal registers and external data bus follow the little-endian convention, which designates the LSB as bit number 0. Bytes within 64-bit doublewords are numbered beginning from the right in little endian and from the left in big endian.

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

5.8 Crossbar and Transfer Controller Arbitration

The MVP's crossbar network allows the on-chip devices—the MP, PPs, and TC—to access multiple on-chip RAM banks in parallel without interference. Only one device can access any single RAM bank at a time. When two or more devices attempt to access the same RAM bank at the same time, the accesses must be performed serially, delaying some requests.

The arbitration logic associated with the crossbar network makes the decision to grant one of several competing access requests. This logic bases its decision primarily on a fixed set of priorities assigned to the various types of accesses that can be performed.

Memory accesses by the PPs, however, are all assigned the same priority. When two or more requests from PPs are competing for access to the same RAM bank, the crossbar arbitration logic relies on a round-robin scheduling policy to grant the requests in an equitable manner. The round-robin policy ensures that each PP is given a turn accessing the RAM bank and that no PP's request is ignored for a long period of time.

The transfer controller contains its own logic for arbitrating among competing transfer requests from on-chip devices. The TC assigns to each request a priority on the basis of the type of transfer needed. When multiple requests have the same priority, however, the TC grants them on a round-robin basis. Once the TC is prepared to perform the requested transfer, the TC must itself request one or more memory cycles from the crossbar's arbitration logic.

5.8.1 Crossbar Round-Robin Scheduling

Access to the crossbar changes dynamically according to the level of the request that is being serviced. Arbitration of the crossbar requests is shown in NO TAG, *Crossbar Access Priority*, in the *MVP System-Level Synopsis*. The crossbar priority is assigned as follows:

- The TC operates above the MP level when servicing an urgent packet transfer request, a cache service request, or a DEA request, or when flushing its pipeline of pending external cycles. TC pipeline flushing occurs whenever the TC receives an urgent refresh, video controller, or externally initiated request or a soft reset; none of these can begin while external operations are in the pipeline. Pipeline flushing occurs occasionally and locks out the MP for a short period of time.

- The MP has the next level of crossbar priority to service interrupts, schedule operating-system events, and dispatch PP messages and assignments.

- The TC is given priority above the PPs but below the MP for high-priority packet transfers. This gives the TC maximum possible priority without locking out the MP (which could have undesirable operating system implications).

- The PP round robin uses a token to identify the last processor that was granted crossbar access. Note that this is the only round-robin scheduling done by the crossbar's arbitration logic; everything else is driven by fixed priorities. For more information about this token passing, see page NO TAG in the *MVP System-Level Synopsis.*

- The TC priority is below that of the PPs if it is performing a low-priority packet transfer. This prevents the TC from stealing crossbar bandwidth from the PPs when the TC is to be truly low priority.

Whenever the TC receives a higher priority request, it completes or suspends its current operations at the crossbar priority of the new request. This ensures that no blockages occur in the system. For example, a low-priority packet transfer suspension occurs at the high-priority packet transfer level when a high-priority packet transfer request is received.

For more information about the crossbar and its round robin, see Chapter NO TAG, *The Crossbar,* in the *MVP System-Level Synopsis*.

5.8.2 Transfer Controller Round-Robin Scheduling

The transfer controller (TC) handles many different types of requests from other processors and controllers. To ensure optimal system performance, these requests are prioritized by their urgency and importance. Because the TC operates at these different priorities, its own priority on the internal crossbar can vary from cycle to cycle.

The various requests which the TC can receive are prioritized as shown in NO TAG, *Transfer Controller's Request Prioritization,* in the *MVP Transfer Controller User's Guide*. When multiple requests with the same priority are received, the TC arbitration round-robins between them. Note that any processor can have only one active priority for its packet transfers. PPs are restricted to high- and low-priority packet transfers, while the MP can submit packet transfers at any level. The reasoning behind the priorities of NO TAG is as follows:

- **Loss of bus**—An external device, such as a system host, must be able to gain immediate access. The device can yield the bus back to the TC for high-priority requests (such as VC cycles), if required, by monitoring the REQ[1–0] pins.

- **VC request**—VC serial register transfer (SRT) requests receive the second highest priority so that time-critical VRAM transfer cycles can occur without disruption to video display or capture.

- **Urgent refresh**—Because VC and host request cycles occur only intermittently, urgent DRAM refreshes are prioritized below them.

- **Video controller frame timer-initiated packet transfer**—VC frame timer-initiated packet transfers are assumed to be urgent (for example, a start-of-frame data movement is required). One of four different linked-list packet transfers is initiated by the TC when the FMEMCTL⟦P⟧ = 1 for external-to-external transfers.

  This permits the video controller (when FMEMCTL⟦P⟧ = 1) to issue a packet transfer request to the TC. These VC frame timer-initiated packet transfers must wait for urgent refresh, cache requests, or another VC frame timer-initiated packet transfer to finish. VC frame timer-initiated packet transfers cannot be suspended but can fault; moreover, they stop on a fault. However, they are not recoverable or resubmittable.

- **Externally initiated packet transfer**—Externally initiated packet transfers are assumed to be urgent (for example, a data-buffer-full or a display-refresh data transfer). One of seven different linked-list packet transfers is initiated by the TC when the CONFIG⟦X⟧ = 1 for external-to-external transfers.

  This permits a host processor or peripheral device (when CONFIG⟦X⟧ = 1) to issue a packet transfer request to the TC. These externally initiated packet transfers must wait for urgent refresh, cache requests, or another externally initiated packet transfer to finish. Externally initiated packet transfers cannot be suspended but can fault; moreover, they stop on a fault. However, they are not recoverable or resubmittable.

- **High-priority mode MP cache/DEA request**—The next priority level is for MP cache service and DEA cycles that occur when the MP is in a high-priority mode. This allows interrupt service routines and other high-priority MP tasks to be performed quickly to maximize system performance. See subsection 3.6.3, *Enable/Disable High-Priority Mode*, for more information about enabling or disabling high-priority requests.

- **High-priority mode MP urgent packet transfer request**—These can also be essential to MP high-priority tasks. They are prioritized below high-priority MP cache and DEA requests so as not to stall the MP's execution. See subsection 3.6.3 for more information about enabling or disabling high-priority requests.

The next three items are the only round-robin scheduling that the TC arbitration logic performs. Everything else is driven by fixed priorities.

- **Cache, DEA, immediate packet transfers** (round robin)—PP cache service and DEA packet transfers are next in priority. It is important that these be serviced quickly since the processor is idle until the request is serviced. MP cache service, DEA requests, and MP immediate packet transfers are also at this priority if interrupts are enabled (IE⟦ie⟧ = 1), since the MP is not servicing a request from elsewhere in the system.

- **High-priority packet transfers** (round robin)—High-priority (foreground) MP packet transfers imply that the requesting processor is waiting for data to finish transferring or that the TC needs to take priority over the PPs for the crossbar access to optimize external bus bandwidth.

- **Low-priority packet transfers** (round robin)—Low-priority (background) MP packet transfers imply that the processor is not waiting for the data so they are given a very low priority.
- **Refresh**—The lowest priority is given to the trickle DRAM refresh cycles. These cycles are only performed when the external bus is idle and the refresh backlog is nonzero. This helps lower the backlog and reduce the likelihood of a high priority urgent refresh being requested at a later time.

Chapter 6

Cache Management

This chapter describes the master processor's data- and instruction-cache memories. The MP's dual-cache architecture allows it to access instruction and data words in parallel during the same clock cycle. When a cache-miss occurs, the transfer controller (TC) automatically updates the cache from external memory. The MP's data and instruction caches each use 4K bytes of on-chip static RAM for storage. They enable the MP to run at speeds approaching those that would be obtained if the MVP's entire external memory ran at the speed of on-chip RAM.

Chapter NO TAG, *Understanding the MVP Memory Organization*, in the *MVP System-Level Synopsis* provides additional information about instruction and data caches of the MVP.

Topics

6.1 Instruction Cache

The MP's instruction cache has the following features:

- 4096-byte instruction-cache space (four-way set associative):
  - Each set contains four 256-byte blocks
  - Each block contains four 64-byte subblocks
  - Each subblock contains sixteen 32-bit instruction words
  - Each block begins on a 256-byte boundary
  - Each subblock begins on a 64-byte boundary inside the block's address space
- Least recently used (LRU) replacement algorithm

The MP waits for contention with the TC and the PPs to be resolved by arbitration, as discussed in subsection 5.8.1, *Crossbar Round-Robin Scheduling*.

6.2 Data Cache

The MP's data cache has the following features:

- 4096-byte data-cache space (four-way set associative):
  - Each set contains four 256-byte blocks
  - Each block contains four 64-byte subblocks
  - Each subblock contains sixteen 32-bit data words
  - Each block begins on a 256-byte boundary
  - Each subblock begins on a 64-byte boundary inside the block's address space
- Write-back protocol for modified data and cache coherency
- No bus snooping or bus watching
- Least recently used (LRU) replacement algorithm

The MP waits for contention with the TC and the PPs to be resolved by arbitration as discussed in subsection 5.8.1, *Crossbar Round-Robin Scheduling*.

The cache replacement algorithm is described in Section 6.5.

6.3 Cache Management Tasks

There are two cache subsystems on the MP:

- The **instruction-fetch unit** controls the instruction cache and handles all instruction operations, including immediate data fetches.
- The **data-cache controller** manages the data cache and handles all data-fetch accesses (both to the noncached on-chip RAMs and to the cached off-chip memory.)

Both subsystems perform the following management tasks on their respective caches:

- Read data from the cache.
- Request TC to read a cache subblock.
- Keep the present flags and the LRU (least recently used) registers up-to-date.

In addition, the data-cache controller performs the following functions:

- Write data to the cache using write-back protocol.
- Keep dirty flags up-to-date.
- Request TC to write back all dirty subblocks in LRU block.
- Request TC to write back dirty subblock(s) for dcache instructions.

The following sections discuss the MP data cache because it supports all of the operations. The operations supported by the MP instruction cache are a subset of these.

6.4 Cache Architecture

Figure 6–1 illustrates the MP's instruction and data cache organization. Each cache contains two banks (2K bytes each) of on-chip RAM for storing up to 1024 32-bit words (instructions or data).

As Figure 6–1 shows, each cache is divided into four sets (two sets per 2K-byte bank), with four blocks in each set. Each block is further partitioned into four subblocks that each contain sixteen 32-bit words. Each block must be contiguous, beginning on a 256-byte boundary located in external memory or in the on-chip SRAM (see Figure 6–3). Each subblock must also be contiguous, beginning on a 64-byte boundary located within its block.

Cache-miss service by the TC is performed on subblocks. Sixteen 32-bit words are loaded into cache when a cache miss occurs.

Figure 6–2 illustrates the cache architecture, including the tag registers for each block and the LRU (least recently used) stacks.

Figure 6–1. MP Cache Structure—Blocks and Subblocks

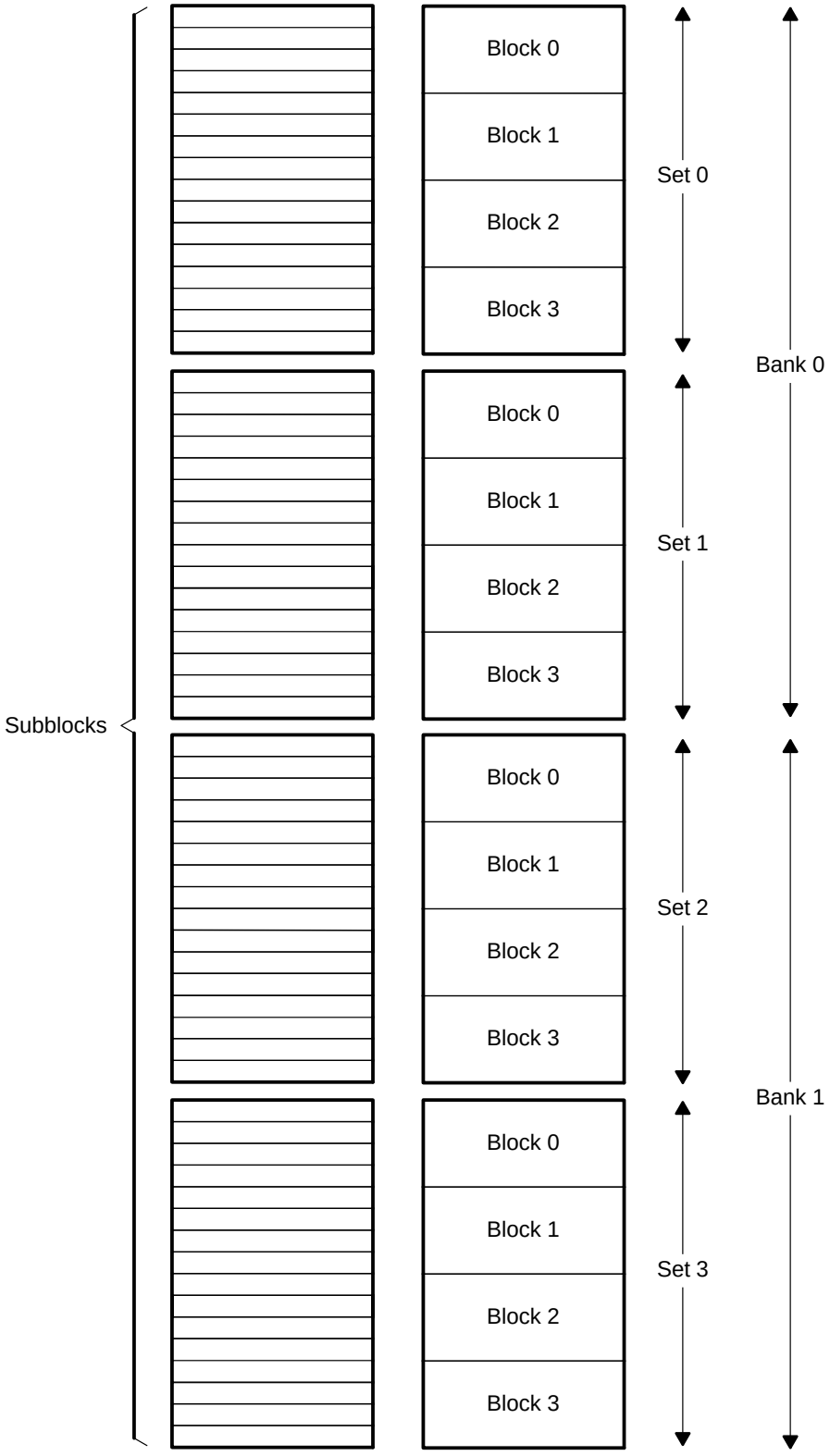
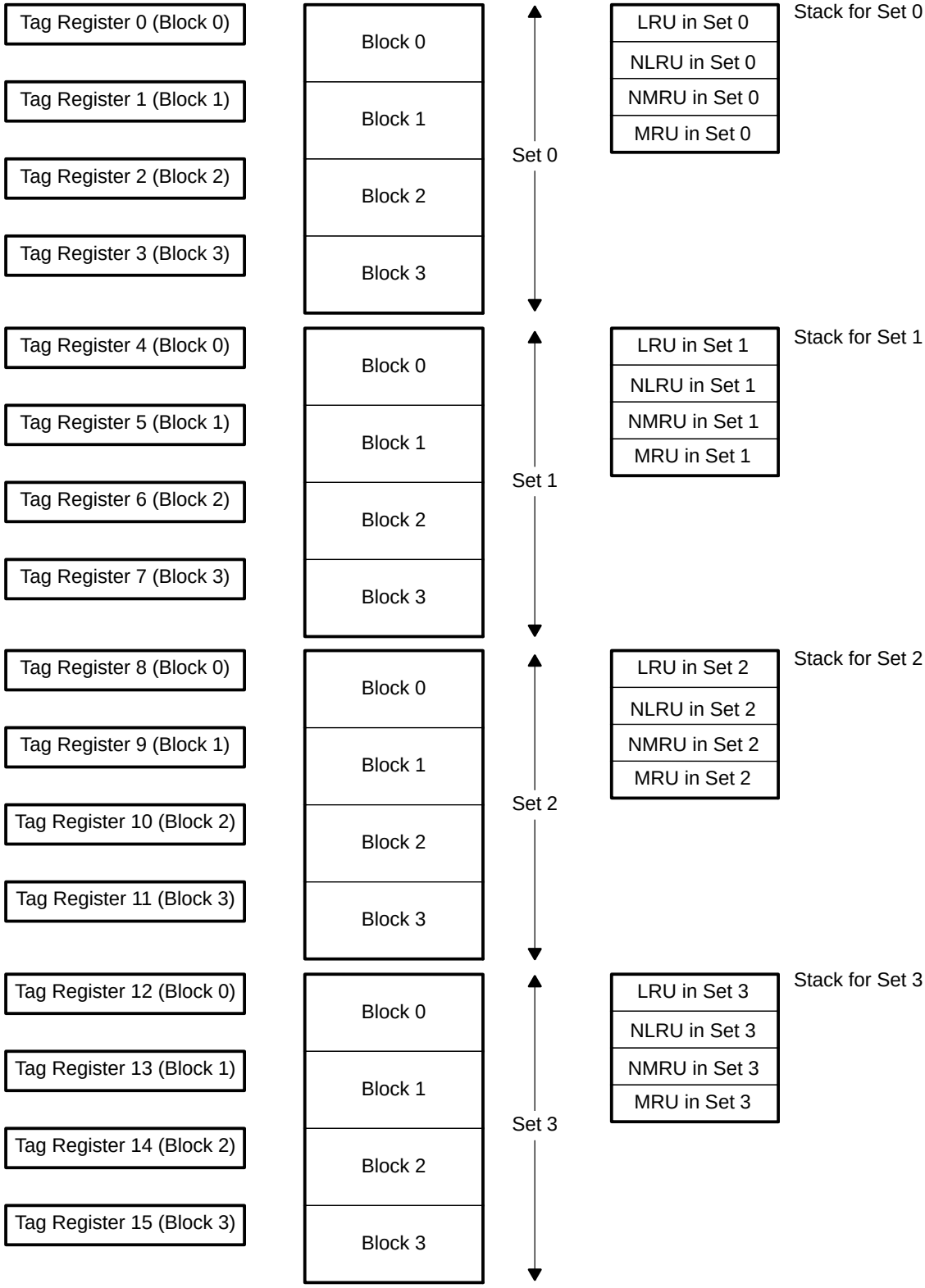


Figure 6–2. MP Cache Structure—Tag Registers and LRU Stacks



Notes: 1) LRU = Least recently used; MRU = Most recently used
NLRU = Next least recently used; NMRU = Next most recently used

2) See Figure 3–5, *MP Cache LRU Register*, for the stack position by sets.

6.5 Cache Replacement Algorithm

When the MP requests a word from external memory, the 22 MSBs of the address are compared with the tag values that are in the corresponding cache (data or instruction) tag registers (see Figure 3–6, *MP Cache Tag Registers*). Address bits 8–7 select the set that contains the four tag registers that are used in the comparison. If the comparison finds a match, a further search determines whether the designated subblock is present (resident in cache); the P bit associated with each subblock indicates the presence of this subblock within a cache block:

- P = 1 indicates that the addressed word is in cache. This is called a **cache hit**.
- P = 0 indicates that the addressed word is not in cache. This is called a **cache miss**.

6.5.1 Cache Hits

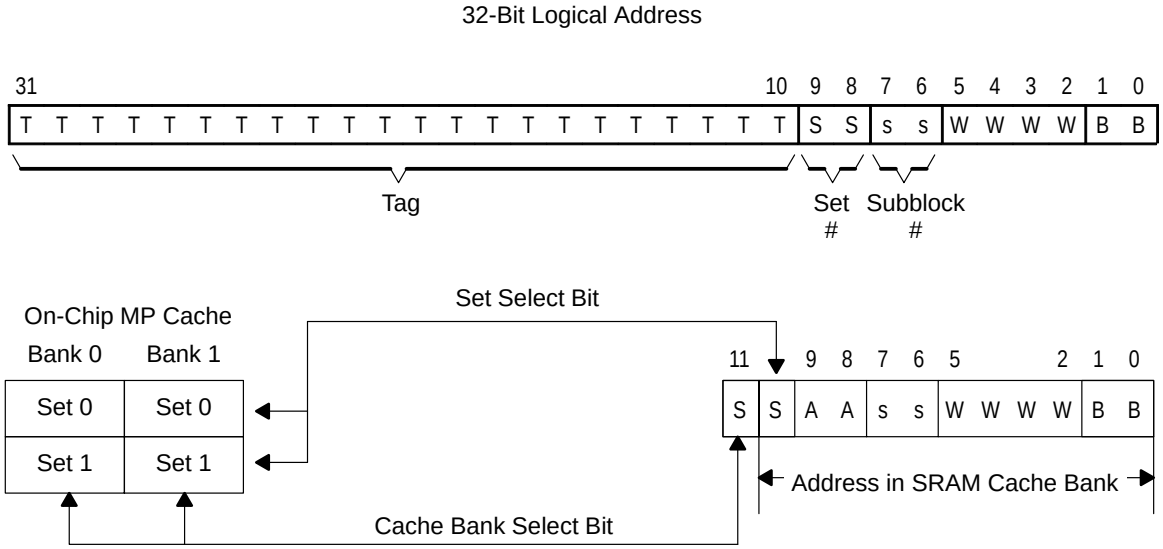
A cache hit indicates that the cache contains the requested value. When a cache hit occurs, the following action is taken:

- A load (ld) or store (st) instruction accesses the value (byte, halfword, word, or doubleword) in cache.
- The block number that contains the cache hit moves to the top of the stack for this set as MRU (most recently used), pushing the other block values toward the bottom of the LRU (least recently used) stack for this set (assuming that this block was not the most recently used block in this set).

When a cache hit occurs, the set and block bits are used with the cache base address (instruction or data) to select the required address in instruction- or data-cache SRAM (see Figure 6–3). The MP uses the following algorithm:

1) Use address 0x0181 8000 if the cache hit is in the instruction cache. Otherwise, use address 0x0181 0000 for a hit in the data cache.
2) Decide which of the two SRAM cache banks to select.
3) Decide which address within the selected bank is needed (for example, which block contains the cache hit).
4) Determine the final SRAM address by adding the base address (the address of the instruction or data cache) and the address in SRAM.
5) Perform the requested load (or store) data-cache access. A store sets the data subblock's cache dirty bit (D) to 1.
6) Move this block number to the top of stack for this set as MRU (if this block was not the MRU block) by pushing down other block values in the LRU stack for this set.

Figure 6–3. MP Cache View of Addresses



Key:

- T Tag bits
- S Set select bits (0–3)
- s Subblock within block select (0-3)
- W Word within subblock select (0–15)
- B Byte within word select (0–3)
- A Block within set select (for example, which tag matched) as 0–3.

Notes: 1) The MP instruction cache begins at 0x0181 8000.
2) The MP data cache begins at 0x0181 0000

6.5.2 Cache Misses

A cache miss indicates that the caches do not contain the requested value that needs to be fetched. There are two types of cache misses:

- **Block miss**—The 22 MSBs of the address are not equal to any of the 22-bit tag register values. In this case, the LRU (least recently used) algorithm determines which block within the set to discard (the set is determined by address bits 8–9). The 32-bit address includes the set and subblock numbers, as shown in Figure 6–3.

  For data cache, if any of the subblock dirty bits (D) are equal to 1, all of dirty subblocks in this block are written back to external memory at their original addresses. The present (P) and dirty bits are then reset. Next, the TC loads the required subblock; the corresponding cache tag value is revised for this block, and the subblock's P bit is set to 1.

  The TC service for the writes (if needed) and reads is arbitrated as a function of the TC round-robin protocol, as shown in NO TAG, *Transfer Controller's Request Prioritization*, in the *MVP Transfer Controller User's Guide*. Instruction-cache service stalls the F (fetch) stage of the pipeline; data-cache service stalls the E (execute) stage of the pipeline only when scoreboarded registers are required or another data access is required before the previous one has completed. After the cache is serviced, the access is performed, and the MP's stalled pipeline resumes operation.

- **Subblock miss**—One of the 22-bit tag register values equals the 22 MSBs of an address, but the present (P) bit representing the cache subblock is not resident. As a result, a cache miss is issued to the TC. Next, the required subblock is loaded by the TC, and this subblock's P bit is set to 1.

  The TC service for the writes (if needed) and reads is arbitrated as a function of the TC round-robin protocol. Instruction-cache service stalls the F (fetch) stage of the pipeline; data-cache service stalls the E (execute) stage of the pipeline only when scoreboarded registers are required or another data access is required before the previous one has completed. After the cache is serviced, the access is performed and the MP stalled pipeline resumes operation.

The minimum time for the TC to service a cache miss is a function of external-memory access time.

- If the TC is not busy, the time to read one cache subblock is approximately 12, 22, and 30 cycles for 1-, 2-, and 3-cycle DRAM, respectively.
- If the TC is busy, the cache service begins when the present $\overline{\text{CAS}}$ cycle has completed.

6.6 Data-Cache Reset

The MP's data and instruction caches are reset during a power-up reset of the MVP device. Software can also reset the caches with a cmnd instruction that specifies the MP as the target of a data-cache reset (DCR) and instruction-cache reset (ICR). The DCR operation and power-up reset both affect the data cache as follows:

- Reset all present (P) bits to 0 (nonresident) in all DTAG registers.
- Reset the data-cache dirty (D) bits to 0 (not modified) in all DTAG registers.
- Reset the DLRU (data-cache least recently used) register.

Note that the DCR operation differs from the cache-flush operation performed either by the transparent cache-block replacement service or by the dcachef instruction. Either of these flush operations writes any dirty subblocks back to memory before clearing the P and D flags. A DCR, on the other hand, clears the P and D flags without first writing the dirty data back to memory; any dirty data contained in the cache is simply discarded.

Note:

If the dirty data in the data cache must be preserved, then you must use a software loop to examine all of the D bits. A command to reset the caches resets only the P and D bits and the LRU register—**no** write-back operations are performed. See the example in Section 11.10, *Clean Data Cache Using the dcachec Instruction*.

6.7 Code-Development Guidelines

Although code alignment is rarely necessary, when developing critical loops for the MP, you should follow these software guidelines:

- **Code alignment.** To minimize cache misses, you should ensure that tight-loop code does not cross subblock or block boundaries. To do so, align sections of code to 64- or 256-byte-aligned addresses, respectively (use the assembler's .align directive). This can reduce cache misses that degrade processor performance.

- **Self-modifying code.** Avoid using self-modifying code; it can cause unpredictable results. When a program modifies its own instructions, only the instruction copy that resides in external memory is altered. Copies of the instructions that occupy cache are not affected, and the internal control logic makes no attempt to detect this situation.

Chapter 7

Understanding the Packet Transfer Request Protocol

The master processor can directly access the MVP's on-chip RAM and memory-mapped registers, but it relies on the transfer controller (TC) to move data on and off chip. The TC's role in performing data transfers for cache servicing and DEA loads and stores is largely transparent to software executing on the MP. However, the MVP architecture provides a mechanism called the **packet transfer request** through which software can explicitly request that the TC transfer one or more blocks of data.

This chapter describes the format, submission, and completion of packet transfer requests, and concludes with an example of using linked packet transfers to output/input a row of image data. For additional information on packet transfers, see Chapter NO TAG, *Packet Transfers*, in the *MVP Transfer Controller User's Guide*.

Topics

7.1 Understanding Packet Transfers

These are the ways in which the MP can access data:

- Typically, the MP accesses data in external memory through the automatic caching provided by the TC and the 4096-byte data cache region found in the on-chip RAMs. The operation of the cache is largely transparent to MP-resident software, as discussed in Section 6.2, *Data Cache*.
- The MP can access data in the MVP's on-chip RAM and memory-mapped registers directly over the MVP's crossbar without using the TC.
- Another method of data access is direct external access (DEA) between MP registers and external memory. However, this method requires 8 to 18 cycles to perform a single load or store from or to external memory. Moreover, this is not efficient when more than several words of data are required.
- The last method the MP uses to access data is for the software to explicitly issue a request for a packet transfer to the TC. This packet transfer typically transfers a block of data from the external memory to the on-chip RAMs or vice versa.
  1) The TC transfers the required data in external memory to the on-chip RAMs in response to a data transfer request (referred to as a **packet transfer request**) that the MP software issues (occasionally on the PP's behalf).
  2) The MP (or PP) processes the data in the on-chip RAMs, which is accessed in a single cycle over the MVP crossbar.
  3) The TC transfers the generated output data from the on-chip RAMs to external memory in response to another packet transfer request.

The rest of this chapter provides a basic overview of the protocol through which software explicitly requests packet transfers and verifies completion of previously issued packet transfer requests.

7.2 Setting Up and Requesting a Packet Transfer

The software that runs on the MP follows these steps to send a packet transfer request to the TC:

Step 1: Load the parameters describing the packet transfer into a 64-byte buffer in on-chip RAM.

Step 2: Load the address of the packet transfer parameters into the linked-list pointer location in the MP's parameter RAM.

Step 3: Issue the packet transfer request to the TC by writing to the MP's PKTREQ register.

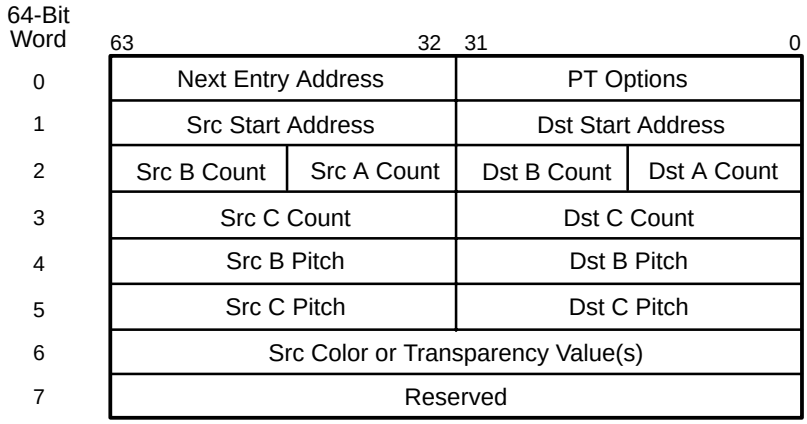
7.2.1 Step 1: Loading the Parameters

Before you issue a packet transfer request, you must set up the packet transfer parameters. The parameters that specify the transfer must be loaded into a 64-byte buffer located in one of the on-chip RAMs (typically in parameter RAM). The buffer must start at a 64-byte-aligned address. The TC responds to a packet transfer request by reading and interpreting the block of parameters specified in the request.

Figure 7–1 shows a parameter block that specifies a dimensioned packet transfer. This type of transfer, which is one of several types that the TC recognizes, can be used to copy a rectangular region of pixels from one location in a frame buffer to another, for example. The individual parameters in the block specify the source and destination addresses; the source and destination transfer modes for the packet transfer; the quantity, location, and structure of data that is transferred; and several other optional features.

Further examples of packet transfer parameter set-ups are shown in subsequent sections.

Figure 7–1. Packet Transfer Parameters—Big Endian



Notes: 1) The parameters **must** start on a 64-byte boundary in on-chip SRAM.

2) Refer to Section NO TAG, *Packet Transfer Parameter Fields*, in the *MVP Transfer Controller User's Guide* for more information about the various types of packet transfer requests that the TC recognizes.

3) Use care when storing packet request parameters in big/little endian, especially the halfword A and B count values. To make your software independent of the big-/little-endian mode, pack these values into a 32-bit word and use a store word instruction.

7.2.2 Step 2: Loading the Address of the Packet Transfer Parameters

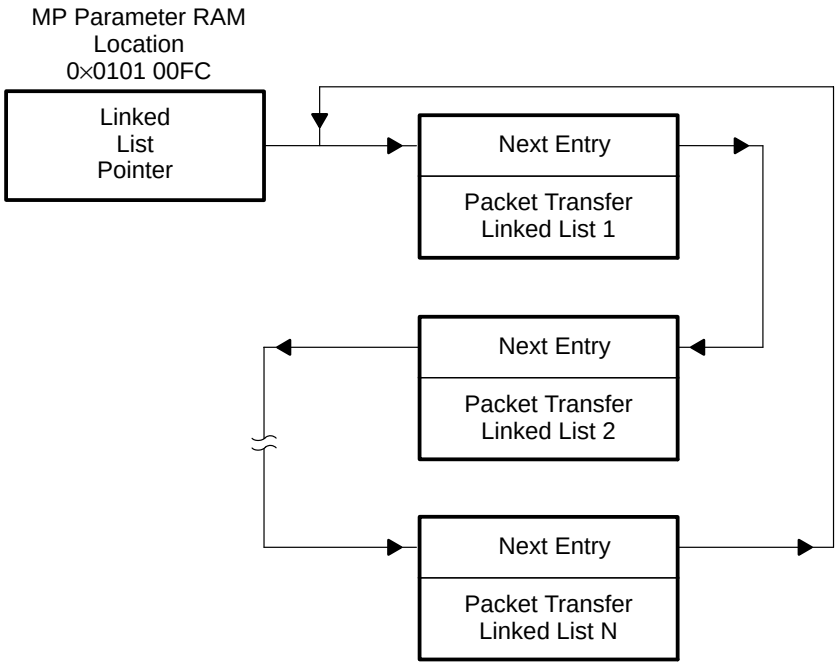
Each processor—the MP or PPs—can submit only one packet transfer request at a time. The processor must wait for the previous request to complete before submitting a new request.

The parameter blocks for multiple packet transfers can be chained together to form a linked list (see Figure 7–2). Each block is linked to its successor through the **next entry address** field shown in Figure 7–1. The processor can submit the list to the TC as a single request. Before making the request, the MP should load the linked-list pointer into address 0x0101 00FC in the MP's parameter RAM. After receiving a request from the MP, the TC retrieves the pointer from this location.

The linked list is terminated by setting the stop bit in the **packet transfer options** field in the last block in the list. When the TC finishes servicing the request, it copies the contents of the last block's **next entry address** field into the linked-list pointer located at address 0x0101 00FC.

Whenever possible, you should ensure that the linked-list pointer that the TC stores at the end of one request is a pointer to the next request that you plan to submit. This eliminates the overhead of having the software explicitly load a new linked-list pointer before each request.

Figure 7–2. Packet Transfer Circular Linked-List Structure



Note: Normally, packet transfer N has option bit 31 set to 1 as a signal to the TC that this is the final packet transfer in the linked list. When the transfer is complete, the TC sets INTPEN[[pc]] = 1 (and issues a **packet-complete interrupt** to the MP if IE[[pc]] = 1 and IE[[ie]] = 1). If options bit 28 = 1, the TC sets INTPEN[[pc]] = 1 when that packet request is complete, even if it is not the last request.

Note:

Do not write to the linked-list pointer location in the MP's parameter RAM while the TC is still processing a previously requested packet transfer (as indicated by PKTREQ[[Q]] = 1). 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.

7.2.3 Step 3: Issuing a Packet Transfer Request to the TC

To issue a packet transfer request to the TC, set the PKTREQ[P] bit to 1 (see Figure 3–1, *MP Packet Request Register—PKTREQ*, for more information about the PKTREQ register). This places the MP'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 Section 7.4.

Example 7–1 shows how an MP program requests a packet transfer from the TC by writing to the PKTREQ register. The program writes a 1 to the P bit to submit the request; it also writes 0s to the I and F bits to specify a background-level priority for the transfer.

Example 7–1. Issuing a Packet Transfer Request: Setting the P Bit (Low Priority)

```
or    1,r0,r5   ; set P (bit 0) to a 1
wrcr  PKTREQ,r5 ; r5 –> PKTREQ
```

Note: The MP must be in supervisor mode to write to control register numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

The MP can submit different priority packet requests, as well as suspend a packet request that is in progress. The PKTREQ bit settings are shown in Table 3–2, *Setting the PKTREQ Bits.*

7.3 Waiting for a Packet Transfer to Complete

There are several situations in which you must ensure that a previous packet transfer has completed before proceeding:

- When entering a tight loop that processes data brought into the on-chip RAMs by the last packet transfer requested
- Before overwriting data in an output buffer that is in the process of being written out to external memory by a packet transfer
- When writing to the linked-list pointer location in the MP's parameter RAM

You can use two methods to verify that a packet transfer has been serviced:

- You can poll the PKTREQ[[Q]] bit or INTPEN[[pc]] bit directly.
- You can use the packet-busy or packet-complete interrupt service routines.

Note:

An error in the parameters specified in a packet transfer request causes a bad-packet interrupt (indicated by INTPEN[[bp]] = 1). Examples of this type of error include a parameter start address that is off-chip or not 64-byte aligned, enabling transparency for an on-chip destination, or specifying fewer source than destination bytes. A bad-packet interrupt is distinct from a memory-fault interrupt (indicated by INTPEN[[mf]] = 1), which can occur if a packet transfer attempts to access an illegal or invalid address. See Section NO TAG, *Packet Transfer Errors*, in the *MVP Transfer Controller User Guide*.

7.3.1 Polling

You can use two polling techniques to determine whether or not a packet transfer is complete:

- Polling the PKTREQ[[Q]] bit to verify the completion of a linked list of packet transfers.
- Polling the INTPEN[[pc]] bit to verify the completion of:
  - The last packet transfer in the linked list. The final packet transfer in the list is designated by setting the stop bit (bit 31) in the PT options field.
  - An individual packet transfer within the linked list (this may not be the final one). This generates a packet-complete interrupt only if the packet transfer's PT options field has its interrupt bit (bit 28) set to 1.

7.3.1.1 Verifying the Completion of a Linked List of Packet Transfers

The most straightforward way to test for completion of a linked list of packet transfers is to poll the queued-packet-transfer bit (PKTREQ[[Q]]). Example 7–2 shows a sample routine that polls the Q bit until it is 0, indicating that no packet transfers are queued for the MP.

When a packet transfer causes a memory fault, the Q bit remains set. The TC suspends the faulting packet transfer but does not set the PKTREQ[[S]] bit. A program that polls the Q bit but does not monitor the memory-fault interrupt runs the risk of waiting forever for a faulted packet transfer to complete. Example 7–2 includes a test of the FLTSTS[[m]] bit to ensure there are no MP packet transfer faults.

Example 7–2. Polling for Completed Packet Transfer

```
Poll:    ld    0x0182000c(r0),r4 ; FLTSTS → r4
         bbo.a MPfault,r4,0      ; m = 1 is MP packet transfer request
                                 ; fault
         rdcr  PKTREQ,r4         ; PKTREQ → r4
         bbo.a Poll,r4,1         ; Test Q bit (PKTREQ register bit 1)
                                 ; Continue in poll loop as long as
                                 ; the packet transfer request is queued
                                 ; (Q = 1)
PTdone:  ....                    ; Packet transfer completion (Q = 0)
         ....
MPfault: ....                    ; MP packet transfer fault (m = 1)
```

Note: There is one delay slot between the submission of a packet transfer by setting the P bit and the time when the Q bit is set. As a result, you should not perform the first poll of the Q bit on the cycle immediately following submission of a packet transfer.

7.3.1.2 Verifying the Completion of a Packet Transfer Within a Linked List

The packet-complete (pc) interrupt flag (bit 26 of INTPEN register, as shown in Figure 3–2, *MP Interrupt Registers—IE and INTPEN*) is set each time a linked list of packet transfers is completed (the final PT options field has its stop bit (bit 31) set to 1). It is also set when an individual packet transfer completes that has the interrupt bit (bit 28) of the PT options field set to 1 (see subsection 7.3.2.2, *Using the Packet-Complete Interrupt Service Routine*). An alternative to polling the PKTREQ⟦Q⟧ bit is to poll the INTPEN⟦pc⟧ bit but is not recommended.

By polling the INTPEN⟦pc⟧ bit versus the PKTREQ⟦Q⟧ bit, you can identify the end of an individual packet transfer in a linked list of packet transfers. The interrupt bit in the PT options field is used to notify the MP software when the TC has finished processing some portion of a linked list of packet transfers. Only then can the software safely access the data buffers used by the previous packet transfers in the list.

For example, when an output packet transfer is linked to an input packet transfer, you may need to ensure that the output packet transfer has completed to avoid overwriting any data before the data has been transferred to external memory. However, the input packet transfer may not be required until the next pass through the processing loop, and as a result, the loop does not need to be completed before continuing.

Note:

By making use of the interrupt bit in the PT options field, you can program a linked list of packet transfers to generate a sequence of packet-completed interrupts as it completes successive stages of processing by the TC. Each time a designated packet transfer in the list completes, the packet-completed interrupt service routine responds appropriately. The routine can distinguish the interrupt generated by completion of the list's final packet transfer from the preceding interrupts by checking the PKTREQ⟦Q⟧ bit, which changes from 1 to 0 only when the TC finishes processing the entire list. You should write the routine to correctly handle the case in which a linked list generates two or more packet-completed interrupts before the first of these interrupts can be serviced.

7.3.2 Using an Interrupt Service Routine

Two interrupt flags provide an efficient alternative to polling in some situations:

- INTPEN[[pb]]—packet transfer queued (packet-busy) interrupt
- INTPEN[[pc]]—packet transfer completed (packet-complete) interrupt

Two additional interrupts indicate when exceptional conditions occur during packet transfers:

- INTPEN[[bp]] (bad packet) indicates an error in the parameters submitted with the packet transfer request.
- INTPEN[[mf]] (memory fault) indicates when the TC attempts to access an illegal address during a packet transfer, or the external memory system signals a memory fault.

This section describes these interrupt service routines:

- Using the packet-busy interrupt service routine to determine whether or not a previous packet transfer is in the queue.
- Using the packet-complete interrupt service routine to determine the completion of a packet transfer.

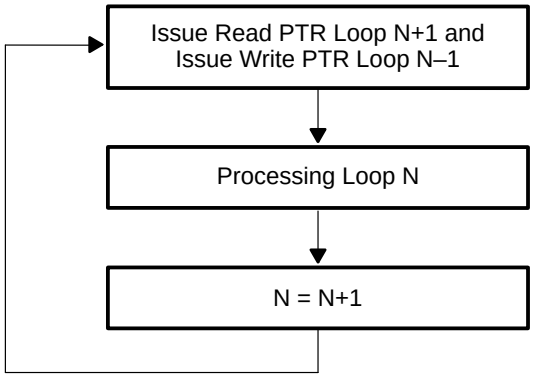
7.3.2.1 Using the Packet-Busy Interrupt Service Routine

The pb (packet transfer queued or packet-busy) flag (bit 27 of INTPEN register) is set to 1 if the software writes a 1 to the PKTREQ[[P]] (packet transfer request) bit when PKTREQ[[Q]] = 1. You can use the packet-busy interrupt as an alternative to polling the Q bit before you issue a packet transfer request.

The packet-busy 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 transfers for PP image data processing is shown in Figure 7–3. The image-processing program transfers the output data from pass N–1 out of the processing loop, then issues a linked list of packet transfers to bring the data for pass N+1 through the processing loop. Next, the program enters pass N of the processing loop. Note that both input and output require double buffering (see Figure 11–4).

For a well-balanced routine, the processing loop requires slightly more time than the TC needs to service the packet transfer requests. In a typical imaging routine, try to balance the load so that most of the time, the previous packet transfer is already complete before you submit the next one in order to avoid the overhead of the packet-busy interrupt service routine. However, because other processors can submit packet transfer requests and because the order of packet transfer service is based on a round robin, it is impossible to be entirely sure that a packet transfer will be complete by the time that it is required.

Figure 7–3. Typical Packet Transfer Request/Processing Loop Flow



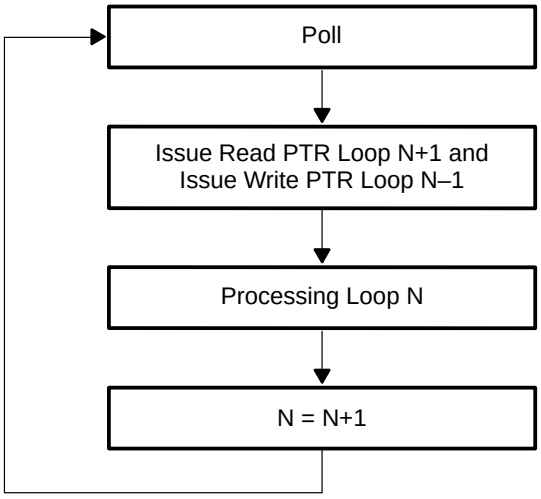
Note: PTR = packet transfer request

This scheme requires that the next entry field from one packet transfer request contain a pointer to the parameters for the next request. When the TC finishes servicing the first request, it copies the next entry field into the linked-list pointer location in the MP's parameter RAM. Until the TC has written this pointer, it ignores any attempt by the program loop in Figure 7–4 to request a new packet transfer. Note that the program loop is unable to safely write to the linked-list pointer location itself unless it first verifies that the TC has finished servicing the previous packet transfer request; having to perform this verification defeats the purpose for using the packet-busy interrupt.

The read and write packet transfers requested at the top of Figure 7–3 are typically contained in a linked list that is submitted to the TC as a single request. Without first verifying that the TC has completed the previous request, the program loop submits the new request by writing a 1 to PKTREQ⟦P⟧. If INTPEN⟦pb⟧ remains 0, this means the TC has accepted the new packet transfer request. If INTPEN⟦pb⟧ changes to 1, however, the attempt to submit the new request failed because the TC had not completed the previously requested packet transfer. Assuming that the packet-busy interrupt is enabled, the service routine for the interrupt is responsible for resubmitting the failed request once the TC has finished servicing the previous request.

While the packet-busy interrupt service routine requires more instructions and generally more cycles than direct polling, the routine is performed only when it is actually required. Direct polling, however, requires one pass through the polling loop, whether or not the packet transfer has already been serviced, as shown in Figure 7–4.

Figure 7–4. Packet Transfer Request/Processing Flow With Polling



Note: PTR = packet transfer request

Example 7–3 illustrates a basic packet-busy (pb) interrupt service routine. The interrupt vector transfer address of the packet transfer request to this routine is located in the MP parameter RAM at address 0x0101 01EC.

Example 7–3. Packet Transfer Busy Interrupt Service Routine

```
; Set the interrupt packet–busy transfer address in MP's parameter RAM
        or    PRbusy,r0,r5      ; address of PRbusy entry.
        st    0x010101EC(r0),r5 ; store in Parameter RAM

; Set the packet–busy interrupt in register IE:
        rdcr  IE,r4             ; IE → r4

        or    0x08000000,r4,r4  ; set bit pb = 1 in IE
        wrcr  IE,r4             ; r4 → IE
        . . .
; Packet–busy interrupt service routine is shown next:

PRbusy:                         ; packet–busy entry
        st.d  –8(r1:m),r4       ; push (r4,r5) onto stack; r1–8 → r1
        ld    0x0182000C(r0),r4 ; FLTSTS → r4

; Optionally enable global interrupts (ie = 1) if polling can be
; interrupted

        rdcr  IE,r4             ; IE → r4
        or    1,r4,r4           ; set bit ie = 1 in IE
        wrcr  IE,r4             ; r4 → IE

; Poll Q bit and m bit to see if TC is still busy with this request

poll:   ld    0x0182000C(r0),r4 ; FLTSTS → r4
        bbo.a MPfault,r4,0      ; MP Packet request fault if m=1
        rdcr  PKTREQ,r4         ; PKTREQ → r4
        bbo.a poll,r4,1         ; spin if Q bit = 1 (packet request is
                                ; queued)

; Test to see if there was a bad–packet error after Q = 0

        rdcr  INTPEN,r5         ; INTPEN → r5
        bbo.a PRerror,r5,28     ; Bad–packet error if bp = 1

; Turn off pb bit in INTPEN

        or    0x08000000,r0,r5  ; bit 28 = 1 → r5
        wrcr  INTPEN,r5         ; turn off pb bit in INTPEN
; Turn on P bit in PKTREQ and (re)issue the packet transfer request

        or    1,r4,r4           ; set P bit = 1
        wrcr  PKTREQ,r4         ; issue packet request to TC

        ld.d  0(r1),r4          ; pop (r4,r5) from stack
        addu  –8,r1,r1          ; restore stack pointer

        brcr  EIP               ; normal return from an interrupt
        brcr  EPC               ; restore PC in delay slot
        . . .
MPfault: . . .                  ; m = 1 is an MP packet request fault
        . . .
PRerror: . . .                  ; bp = 1 is an MP packet request error
```

Note: The MP must be in supervisor mode to write to the MP's parameter RAM or to write to (wrcr instruction) control registers IE, INTPEN, and PKTREQ.

For the packet-busy interrupt service routine, you should use the next-entry address of the final packet transfer in the previous linked list of packet transfers to set the linked-list pointer to the next packet transfer request to be issued. If you write directly to the linked-list pointer without first making sure that there is not a previous packet transfer in the queue, a timing hazard exists, and the linked-list pointer for the previous packet transfer request may be overwritten before it is read by the TC.

The interrupt service routine polls the PKTREQ[[Q]] bit until it is equal to 0. When Q is equal to 0, this indicates that the previous packet transfer is no longer queued. At this point, the new packet transfer request can be resubmitted. Next, the INTPEN[[pb]] is cleared to 0. Finally, the standard return from interrupt operations restores temporary registers, the stack pointer, the instruction pointer (IP), and the PC values, respectively.

7.3.2.2 Using the Packet-Complete Interrupt Service Routine

The TC sets the pc (packet-complete) interrupt flag (bit 26 of INTPEN) when the TC encounters the end of the MP's linked list (PT options bit 31 is set to 1) or completes a packet transfer that instructs the TC to interrupt the MP when the transfer is complete. When the current entry on the linked list has finished, the interrupt bit (bit 28) of the packet transfer options field, if set to 1, causes an interrupt to be sent to the processor that initiated the packet transfer. However, the linked list may contain further entries. This allows the requesting processor to be flagged when a particular point in the linked list has been reached.

7.4 Packet Transfer Handshake Signals

The packet transfer handshake signals between the MP and the TC are contained in the five LSBs of the PKTREQ register (I, F, S, Q, and P), as shown in Figure 3–1, *MP Packet Request Register—PKTREQ*. These bits are cleared to 0 at reset; bits P, S, F, and I are read/write, and Q is read-only.

- P bit: Submit packet transfer request to the TC. If PKTREQ⟦Q⟧ = 0 and PKTREQ⟦S⟧ = 0 (or PKTREQ⟦S⟧ is simultaneously cleared to 0 while the P bit is being set), writing a 1 to the P bit submits a packet request to the TC.
- Q bit: Packet request queue is active. This bit indicates when the TC is busy performing a packet transfer that was requested previously by the MP.
- S bit: Suspend packet request. Software writes a 1 to this bit to suspend a previously requested packet transfer.
- F bit: Foreground priority is selected. You can use this bit along with the I bit, to change the priority of packet transfers dynamically.
- I bit: Immediate (urgent) priority is selected. You can use this bit along with the F bit, to change the priority of packet transfers dynamically.

See Table 3–2, *Setting the PKTREQ Bits*, for more information about setting the S, F, and I bits.

P is not an ordinary read/write bit. P can be written to only if S is 0 or is being simultaneously written with 0; otherwise, P is write-protected.

In order to detect the completion of a packet transfer request, you should install interrupt service routines and enable the IE register's interrupt bits for three cases:

- Packet transfer completed (pc bit). The TC has completed the last packet transfer in a linked list submitted by the MP, or the TC has completed a packet transfer in which the interrupt bit in the PT options field is set to 1.
- Packet transfer queued (pb bit). The TC is busy with an earlier MP packet transfer request, and the new transfer cannot begin.
- Packet transfer error (bp bit). The TC cannot finish servicing this new MP packet transfer request, because a parameter error has been detected in the packet transfer list.

In addition to monitoring these interrupts in the INTPEN register, you should monitor the INTPEN[[mf]] interrupt to determine if a memory fault occurs while servicing a packet transfer request.

If an interrupt condition occurs, the bit corresponding to that interrupt is set in the INTPEN register (for example, packet-busy sets the INTPEN[[pb]] bit to 1); however, an interrupt is taken only if the following conditions are satisfied:

- The interrupt is enabled:
  - The bit representing the interrupt in the IE register is set to 1.
  - The global interrupt-enable bit is set (IE[[ie]] = 1).
- No interrupt of higher priority is both pending and enabled.

Figure 7–5 shows the handshake signals between the PKTREQ register and the corresponding interrupt enable bits that can be activated in the INTPEN register. The protocol for manipulating the packet transfer handshake bits in the PKTREQ register is shown in Figure 7–6.

Figure 7–5. Packet Transfer Handshake Signals

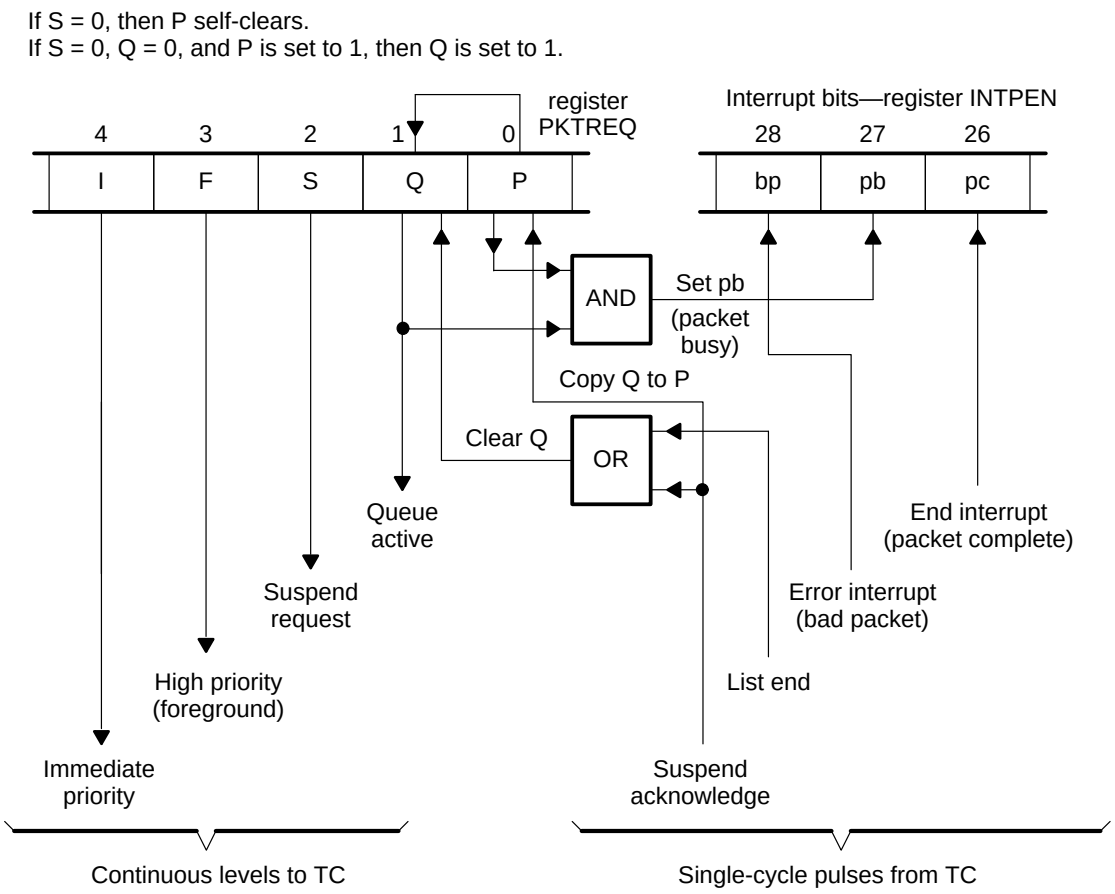
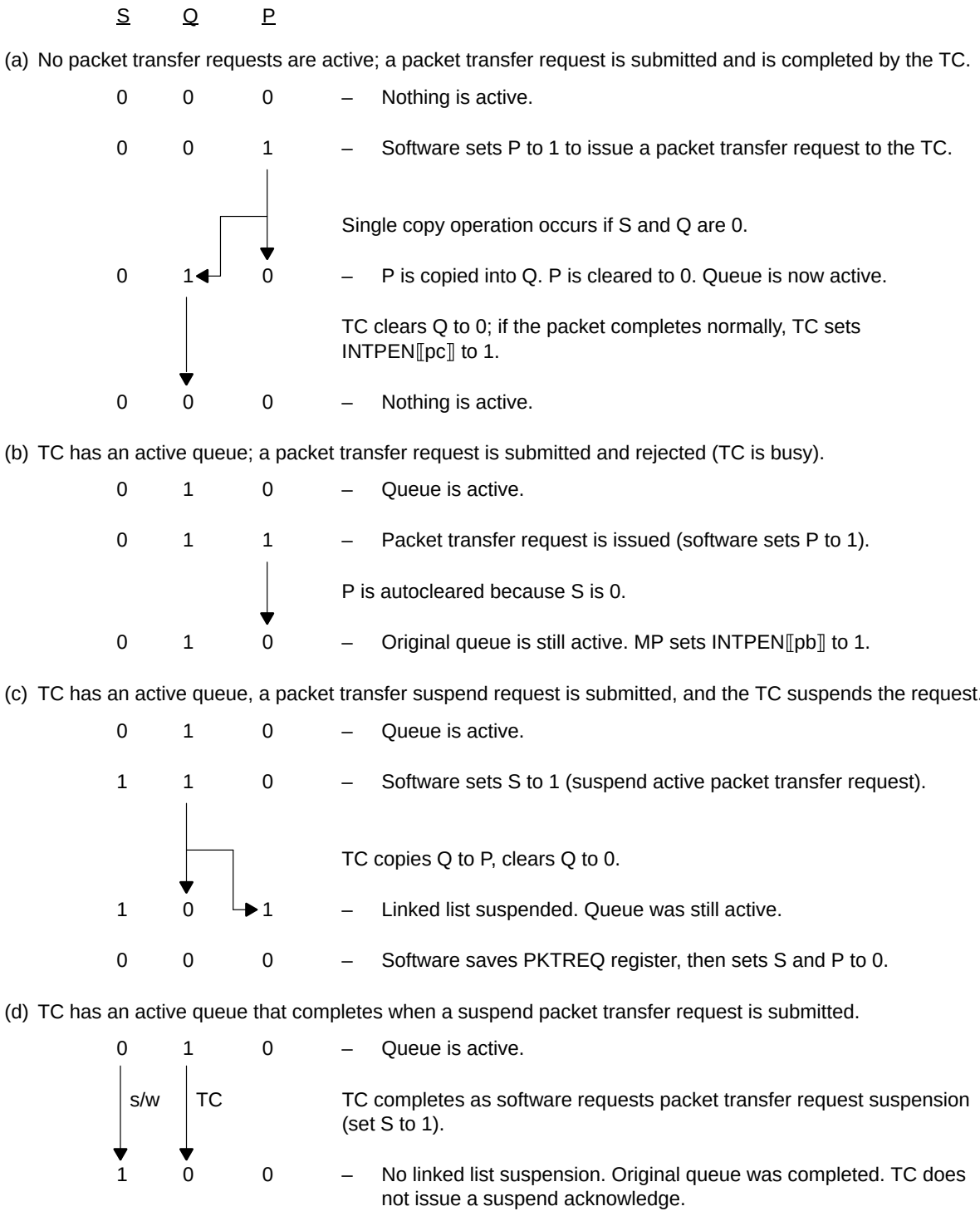


Figure 7–6. Packet Transfer Request Protocol



The packet transfer request protocol shown in Figure 7–6 is further described in the subsections listed in Table 7–1. In order to comprehend the packet transfer request operation, you must understand the behavior of the P (submit packet request to TC) and Q (packet request queue is active) bits of the PKTREQ register.

- **P bit.** The software can write a 0 or 1 to the P bit only if the PKTREQ⟦S⟧ (suspend packet request) bit is already 0 or if the software writes 0 to the S bit simultaneously.
  - If S = 0 (no suspension requested) at the time that a 1 is written to P, the Q bit is set to 1 (to make the packet request active) in the next cycle and the P bit is cleared simultaneously to 0.
  - If S = 1 (suspend packet request is active), writes to P have no effect. The P bit is maintained at its current value except during the cycle in which the TC sends the suspend-acknowledge signal. In this cycle, P is set to the current value of the Q bit.
- **Q bit.** The software can never write to the Q bit directly (Q is read-only).

  If S = 0, Q = 0, and P = 1, then Q is set to 1 on the next cycle (at the same time that P is cleared to 0). When Q = 1, a packet request is in the TC queue awaiting processing or that processing is underway.

  The TC clears the Q bit to 0 when a linked list is completed normally, when a linked list is terminated with an error, or when the suspend-acknowledge signal is sent. When Q = 0, the MP has no active entry in the TC's packet transfer request queue.

Table 7–1. Packet Transfer Handshake Scenarios

Packet Transfer Handshake Scenarios	See Subsection
Submitting a Packet Transfer Request	7.4.1
Suspending a Packet Transfer Request	7.4.2
Resuming a Suspended Packet Transfer Request	7.4.3
Detecting the Completion of a Packet Transfer Request	7.4.4
Changing the Priority of a Packet Transfer Request	7.4.5

7.4.1 Submitting a Packet Transfer Request

When the MP software submits a packet transfer request (sets the P bit to 1), one of two actions can result:

- If the MP has no other packet transfer requests active in the TC's queue (Q = 0), the MP can submit the request successfully.
- If the MP already has a packet transfer request active in the TC's queue (Q = 1), the MP cannot submit the new request.

7.4.1.1 Submitting a Request With No Requests Queued

Before you can submit a packet transfer request, the following conditions must be true:

- The linked-list pointer in 0x0101 00FC is set to the beginning of the packet request's parameter list.
- PKTREQ⟦S⟧ = 0
- PKTREQ⟦Q⟧ = 0

Figure 7–6 (a) shows a typical case in which no packet transfer requests are active and a packet transfer is requested. The software writes a 1 to the PKTREQ⟦P⟧ bit. One cycle later, the Q bit is set to 1 and the P bit is cleared to 0. The new request is now queued and the TC services the linked list. When the TC completes this packet transfer, it clears the PKTREQ⟦Q⟧ bit to 0 and signals either:

- Packet transfer completed, which sets the INTPEN⟦pc⟧ bit (this causes a packet-completion interrupt if IE⟦pc⟧ = 1 and interrupts are enabled).
- Packet error, which sets bit the INTPEN⟦bp⟧ bit (this causes a bad-packet interrupt if IE⟦bp⟧ = 1 and interrupts are enabled).

The PKTREQ⟦Q⟧ bit can still be 1 for the following reasons:

- MP's packet request is queued or is active.
- MP's packet request had a memory fault (FLTSTS⟦m⟧ = INTPEN⟦mf⟧ = 1).
- The program has issued a suspend (PKTREQ⟦S⟧ = 1), and the suspend action is not yet complete.

7.4.1.2 Submitting a Request While Another Request is Queued

While the TC is actively servicing a previous packet transfer request (PKTREQ⟦Q⟧ = 1) and that request is not suspended (PKTREQ⟦S⟧ = 0), the TC cannot accept another request. Figure 7–6 (b) demonstrates the case in which the software attempts to submit another packet transfer request to the TC. The logic associated with the PKTREQ register rejects the new packet transfer request (clears the PKTREQ⟦P⟧ bit to 0) and also sets the INTPEN⟦pb⟧ bit to 1 (this causes a packet-busy interrupt if IE⟦pb⟧ = 1 and interrupts are enabled).

Even if the TC clears the Q bit at about the same time the software writes a 1 to the P bit, the logic associated with the PKTREQ register still performs correctly:

- If the TC clears Q first, the new request is accepted.
- If the software sets P first, the logic associated with the PKTREQ register rejects the new request.

These are the only two possibilities. The response is never ambiguous, regardless of the timing of the request.

7.4.2 Suspending a Packet Transfer Request

While a packet transfer request is being serviced, the request may need to be suspended. There are five suspension cases:

- ❑ The suspension of the current request (normal case)
- ❑ An attempt to suspend a completed or nonexistent request
- ❑ The automatic suspension and resumption of a request
- ❑ The automatic suspension of a request that caused an error
- ❑ The automatic suspension of a request that caused a memory fault

7.4.2.1 Suspension of the Current Request (Normal Case)

Figure 7–6 (c) illustrates the case in which the MP software suspends a previously requested packet transfer that has not yet completed by setting the PKTREQ⟦S⟧ bit to 1. The TC suspends the active packet transfer by saving packet transfer information in the suspended packet transfer area (parameter RAM locations 0x0010 1000 to 0x0101 007F, as shown in Figure 5–2, *MP Parameter RAM Contents*), and copies PKTREQ⟦Q⟧ to PKTREQ⟦P⟧ while clearing the Q bit to 0. Then, the software must save the PKTREQ register so that another packet transfer request can be issued.

The software polls the Q bit until it becomes a 0 to ensure that the list is suspended. If P = 1 when the list suspends (when Q is cleared to 0), this indicates that the linked list had not yet been completed at the time of suspension. The linked-list pointer is pointing to the packet request to be performed if the linked list is to be resumed from where it left off.

Note that there is only one save area for suspended packet transfers initiated by the MP; do **not** try to suspend more than one MP packet transfer at a time.

Note:

Once the MP software has set the S bit to 1 to suspend a packet transfer request, it should wait until the Q bit is 0 before clearing the S bit to 0. If the software fails to observe this protocol, TC operation is not defined.

7.4.2.2 Attempt to Suspend a Completed or Nonexistent Request

When attempting to suspend a previously requested packet transfer, the MP software typically has no way to guarantee that the TC does not complete the transfer before it can be suspended. Figure 7–6 (d) shows what happens when a packet transfer request reaches completion just before the software attempts to suspend the request. This is similar to Figure 7–6 (c) (normal suspension, as described in subsection 7.4.2.1); however, the linked-list information is not saved in the suspended packet transfer area in the MP's parameter RAM, because the original request has completed. After verifying that Q = 0, the software must check the value of the P bit to distinguish between the cases shown in (c) and (d) of Figure 7–6.

7.4.2.3 Automatic Suspension and Resumption of a Request

During the processing of a linked list, the TC may suspend the service of the linked list temporarily. This occurs when the TC switches to servicing a higher priority linked list from another processor or when a time-out occurs. When automatic suspension occurs, the TC suspends the packet parameters to location 0x0101 0000 of the MP's parameter RAM and updates the linked-list pointer (in 0x0101 00FC) to point at that location. This suspension is transparent to the software; PKTREQ register's S, Q, and P bits are unaffected. The TC resumes the packet service automatically when that packet becomes the highest priority pending request again.

The TC correctly handles the case in which the MP software directs the TC to suspend a previously requested packet transfer that has already been suspended automatically by the TC. The automatic suspension remains transparent to the software, and the software suspension occurs normally.

7.4.2.4 Automatic Suspension of a Request That Caused an Error

If the TC detects a bad-packet error during a packet transfer requested by the MP, the TC stops processing the linked list. The linked-list pointer points to the parameters for the packet transfer that caused the bad-packet interrupt. The TC saves the parameters in the suspended packet transfer area (address 0x0101 0000) of the MP's parameter RAM, unless the original pointer was illegal. Then, the TC clears the PKTREQ[[Q]] bit to 0. If a bad-packet error caused the suspension, the TC also sets the INTPEN[[bp]] bit to 1.

7.4.2.5 Automatic Suspension of a Request That Caused a Memory Fault

If the TC detects a memory fault during a packet transfer requested by the MP, the TC suspends processing of the request. The TC saves the parameters for the faulted packet transfer in the suspension buffer (address 0x0101 0000) at the beginning of the MP's parameter RAM, unless the source of the fault was the original linked-list pointer submitted to the TC with the request. The TC updates the linked-list pointer to point to the saved parameters so that the faulted packet transfer is ready to be resumed, if necessary.

Unlike a suspension that occurs as a result of a bad-packet error, a memory-fault suspension has no effect on the PKTREQ[[Q]] bit. In other words, the Q bit remains 1 (and the S and P bits remain 0). Although the Q bit remains set, the FLTSTS[[m]] bit, which is set to 1 by the memory fault, inhibits the MP's packet transfer request to the TC until the memory-fault software clears FLTSTS[[m]]. Clearing the m bit to 0 causes the packet transfer to resume from the point at which it was suspended.

If the memory-fault software decides to abort the faulted packet transfer request, it follows these steps:

- Manipulate the PKTREQ register's S, Q, and P bits to clear the MP's packet transfer request to the TC. First, set S to 1 and wait until Q becomes 0. Then simultaneously write 0s to S and P.
- Reset the TC's internal memory-fault signal by clearing FLTSTS[[m]] to 0. (Don't do this until the Q bit is 0, or the TC will resume the faulted request.)
- Assuming that the entire FLTSTS register is now 0, reset the memory-fault interrupt bit (INTPEN[[mf]]) by writing a 1 to it.
- If necessary, preserve the parameters in the suspension buffer by copying them to another area of memory.

At this point, the MP is ready to request a new packet transfer.

7.4.3 Resuming a Suspended Packet Transfer Request

When a previously requested packet transfer is suspended under software control or by a packet error or memory fault, the software can resume the suspended request at a later time. At the time of the suspension, the TC saves the state of the packet transfer so that it can be resumed from the point at which it was suspended.

MP software interfaces to the TC through the S, Q, and P bits in the PKTREQ register. As described in subsection 7.4.2.1, the software requests the suspension by setting the S bit to 1. (This is the only way the S bit can be set to 1.) After completing the suspension, the TC sends acknowledgment by copying the Q bit to the P bit and clearing the Q bit to 0. At that point, the software can read the P bit to determine the state of the suspended packet transfer:

- If P=1, the TC had not completed servicing the packet transfer request at the time it was suspended. In this case, the software can resume servicing of the request by writing a 0 to the S bit and writing a 1 to the P bit. Both bits can be modified simultaneously within the same write cycle.
- If P=0, the TC had already completed servicing the packet transfer request at the time the software attempted to suspend it. The software can clear the suspension request by writing a 0 to the S bit. The software can request a new packet transfer by writing 1 to the P bit either in the same write cycle in which S is cleared or at a later time.

When the TC detects a packet error, signals from the TC clear the Q bit to 0 and set the INTPEN⟦bp⟧ bit to 1. By writing a 1 to the PKTREQ⟦P⟧ bit, the software resumes the suspended packet transfer from the point at which it was suspended. The software should also clear INTPEN⟦bp⟧ to avoid generating additional interrupts.

When the TC detects a memory fault, signals from the TC set the FLTSTS⟦m⟧ bit to 1 and set the INTPEN⟦mf⟧ bit to 1. The PKTREQ register is not affected by the memory fault; the Q bit remains set to 1. By clearing the FLTSTS⟦m⟧ bit to 0, the software resumes the suspended packet transfer from the point at which it was suspended. The software should also clear INTPEN⟦mf⟧ to avoid generating additional interrupts.

7.4.4 Detecting the Completion of a Packet Transfer Request

When a linked list of packet transfers has been submitted and is in the TC's queue, you can detect the normal completion of the list by monitoring the INTPEN[pc] (packet complete) interrupt, as described in subsection 7.3.2. For a packet transfer request to complete normally, assume PKTREQ[S] = 0 as long as PKTREQ[Q] = 1.

When the TC finishes the linked list, it updates the linked-list pointer in 0x0101 00FC to the **next entry** field of the final packet. Next, the TC clears the PKTREQ[Q] bit to 0 and sets the INTPEN[pc] bit to 1 to indicate packet completion.

Be sure to monitor the INTPEN[bp] and FLTSTS[m] bits for possible packet transfer request error/fault conditions. The Q bit is still 1 on packet request faults.

7.4.5 Changing the Priority of a Packet Transfer Request

The I (immediate) and F (foreground) bits of the PKTREQ register determine the priority of the queued (or active) linked list in the TC's priority logic; you can modify the these bits dynamically to change the priority of a submitted packet transfer request. The priorities are ordered as follows:

immediate priority > foreground priority > background priority

You can have only one active request in the TC's queue at one time. The linked-list pointer belongs to that queue and cannot point to any other request. The TC updates the linked-list pointer as each packet request on the linked list is completed or when suspension occurs.

You can change the I and F bit codings at any time because the TC monitors the priorities of all requests continuously. If the priority of a queued linked list is raised above the priority level of a currently active list, the TC suspends the active list and begins the higher priority list. Conversely, if the priority of an active linked list is lowered below the priority level of another queued packet request, the TC suspends the list and begins the other packet request.

The I and F bits are independent of PKTREQ's P, Q, and S bits. If the software writes a 1 to the P bit at the same time as it changes the F and I bits, a normal submission (described in subsection 7.4.1.1) or packet-busy submission (described in subsection 7.4.1.2) occurs. Similarly, if the software changes the I and F bits when it sets the S bit to 1, a normal suspension or unnecessary suspension (described in subsections 7.4.2.1 and 7.4.2.2, respectively) occurs, and so on.

7.5 Understanding the Classes of Packet Transfers

There are two broad classes of packet transfer source and/or destination transfer modes. Examples include:

- **Dimensioned transfers** are used to transfer tables, linear arrays, and rectangular screen regions from source to destination positions. The rectangular screen move is an example of dimensioned transfer that is used in imaging algorithms.
- **Guided transfers** are used to gather random data into linear arrays or write linear data out to random locations. The random addresses are found in a **guide table**. One example of using guided transfers is writing pixels that represent a line to random addresses. This permits the line to have any orientation on the screen.

The TC provides the MP (or PP) program with the ability to request a 2- or 3-dimensional data transfer in parallel with program execution. This allows movement of data from a source address to a destination address. Dimensioned transfers (for example, one row of pixel data) include sequential data moves in the first dimension, while guided transfers allow address tables for non-sequential address selection (as required, for example, to implement Bresenham's line algorithm).

The *MVP Transfer Controller User's Guide* fully describes these classes of packet transfers in Section NO TAG, *Guided Transfers*, and Section NO TAG, *Dimensioned Transfers*.

Examples of these types of transfers are shown in Appendix D, *Examples of MP Packet Transfers*, of this user's guide.

7.6 Externally Initiated Packet Transfer Requests: XPT1–XPT7

Devices external to an MVP can request that the MVP perform packet transfers. The parameter tables for the transfers must be set up in the on-chip SRAM prior to a request. Also the linked-list pointers must be stored in the MP's Parameter RAM to service these requests. Up to seven externally initiated packet requests, XPT1–XPT7 are recognized by the MVP. The linked-list pointers for these externally initiated packet transfers are stored in MP parameter RAM addresses 0x0101 00E0 to 0x0101 00F8 as shown in Figure 5–3, *Externally Initiated Packet Transfer Linked-List Address Pointers*.

In order for the externally initiated packet transfers to be enabled, CONFIG[X] must be set to 1 after the linked lists and parameter lists have been initialized. For all seven externally initiated packet transfers only, ensure that FMEMCTL0/1[P] is 0 so that no frame timer events will cause an automatic VC packet request. As one can see from Figure 5–3, XPT4–XPT7 share the same linked-list addresses as the four VC frame timer packet requests.

Notes:

1) If CONFIG[X] = 1, then traps 40–71 are unavailable as this area is used for an external transfer buffer area (see Figure 5–2, *MP Parameter RAM Contents*).

2) If CONFIG[X] = 0, then the external pin signals XPT[2:0] are ignored.

Externally initiated packet transfers are not available on preproduction silicon version 1.

7.7 Video Controller-Initiated Packet Transfer Requests

Both of the two video controller (VC) frame timers (0–1) have the option of issuing four packet transfer requests similar to externally initiated packet transfers. These include start-of-field (sof) and serial-access-memory overflow (SAM) events for each of the two sets of frame timer registers. As stated in Section 7.6, the linked-list pointers and the parameter lists must be initialized before this option is enabled. As before, CONFIG[X] = 1 if any one of the seven externally initiated initiated packet requests are to be serviced.

For the four VC frame timer events, FMEMCTL0/1[P] must be set to 1 and the correct options stored into FMEMCTL0/1[EMS] field that trigger these packet transfers.

Notes:

1) XPT4–XPT7 and video controller SAM-overflow-1 to video controller start-of-field-0 **must not** be enabled simultaneously. They must be used in a mutually exclusive manner; otherwise, XPT7 could issue a VC start-of-field-0 packet transfer or vice versa, depending on the linked-list pointer stored in 0x0101 00E0, etc. See Figure 5–3, *Externally Initiated Packet Transfer Linked-List Address Pointers*. For example, only XPT1–3 and all four of the VC frame timer-initiated packet requests satisfy this constraint.

2) If CONFIG[X] = 1 or FMEMCTL0/1[P] = 1, then traps 40–71 are unavailable as this area is used for an external transfer buffer area (see Figure 5–2, *MP Parameter RAM Contents*).

VC frame timer packet transfers are not available on preproduction silicon version 1.

Chapter 8

Floating-Point and Vector Operations

This chapter discusses the floating-point unit and how it handles floating-point and vector operations. For more information about floating-point numbers and their formats, see Appendix A, *Understanding the Floating-Point Numbering System*.

Topics

8.1 The Floating-Point Unit

The MP's floating-point unit supports the IEEE-754 standard. Hardware support consists of a full double-precision floating-point add unit and a double-precision floating-point multiply unit with a single-precision core. The floating-point hardware is pipelined, and the floating-point multiply unit includes microcode to perform double-precision multiplies and single- and double-precision divides and square roots. In addition, you can start a double-precision floating-point add or a single-precision floating-point multiply in each clock cycle.

Note:

Integer multiply is performed by the floating-point multiply unit.

When an instruction is dispatched, floating-point operands are read from the same registers (r0–r31) that are used for integer instructions. Execution always takes more than one cycle. The result is stored back into the register file when the instruction completes. A register scoreboard (the SB register, as discussed in subsection 4.1.2, *Instruction Execute (E)*, and subsection 2.2.3, *Register Scoreboarding*) maintains correct access sequences. The floating-point unit has a dedicated write port to the register file to allow floating-point operations in parallel with memory-interface operations.

The floating-point unit is divided into two independent units:

- The **floating-point multiply unit** performs all multiplies (integer and floating-point), divides, and square roots.
- The **floating-point add unit** handles additions, subtractions, compares, and conversions.

The two units are independent in the sense that if the floating-point multiply unit is busy, a floating-point add operation can begin.

The floating-point multiply unit uses the floating-point add unit for special cases (for example, handling denormals, as discussed in subsection 8.3.7, *Denormals in the Multiplier*).

8.1.1 Floating-Point Execution

Figure 4–1, *General Floating-Point Unit Flow*, shows the dual pipelines for the floating-point unit.

1) During the unpack stage of a floating-point instruction, all data necessary to begin the floating-point operation arrives (source operands, opcode, precisions, destination address).

2) The two source operands are read from the registers (r0–r31).

3) These operands are then unpacked into sign, exponent, and mantissa fields, and special cases are detected.

4) Input exceptions are detected in this cycle. Regardless of whether the input exception interrupts are enabled or disabled, the input exception is piped through the floating-point unit and is signaled on the same cycle as a single-precision output exception.

   The other special cases involving SNaN (signaling not-a-number), QNaN (quiet not-a-number), infinity, denormal, and zero are also detected, and this information (not visible to the user) follows the data through the floating-point unit's pipeline stages.

5) Output exceptions are detected in the final normalize stage.

When the floating-point unit result is determined, some of the individual information about this floating-point operation is recorded in the FPST register. See subsection 3.3.3, *Floating-Point Status Register: FPST*, for more information about how the bits in the FPST register are set.

8.1.2 Vector Instructions

Vector instructions use many of the floating-point unit's instructions and can also provide a load or store in the same instruction to improve program efficiency. These vector load/store operations have dedicated address registers—IN0P, IN1P, and OUTP—with automatic address post-incrementing. There are also four double-precision accumulators (a0–a3) that maintain values from multiply and add vmac-type instructions and free some of the general-purpose registers (r0–r31).

The vector instructions are:

- Vector floating-point multiply and add to accumulator (vmac) (see Example 4–7)
- Vector floating-point multiply and subtract from accumulator (vmsc)
- Vector subtract accumulator from source (vmsub)
- Vector multiply (vmpy)
- Vector add (vadd) or subtract (vsub)
- Vector conversions (vrnd)
- Parallel vector load (vld) or vector store (vst) with other vector operations

These instructions are described in their operations in Chapter 10, *The MP Assembly Language Instruction Set.*

8.1.3 Floating-Point Status Register: FPST

The FPST register contains all of the exception and interrupt information held by the floating-point unit. Figure 3–3, *MP Floating-Point Status Register—FPST*, shows the structure of the FPST register. Each floating-point instruction writes the i, z, o, u, and x bits to the FPST register at its conclusion. Also, the accumulated FPST bits (ax, au, ao, az, and ai) are updated at this time; these bits are the logical-OR of each of the individual bits (x, u, o, z, and i, respectively) since FPST was last reset (for example, new bit ax = old bit ax OR bit x).

Note:

The only indication of an integer multiply overflow using the floating-point multiply (fmpy) instruction is FPST⟦mo⟧ = 1.

Every instruction that is sent to the floating-point unit writes to FPST once and only once. A floating-point instruction is considered complete when its status is written to FPST. When a floating-point instruction is complete, the floating-point scoreboard bit for that instruction is cleared.

8.1.4 Reset

When a power-up or software (cmnd instruction) MP reset is issued, the floating-point unit performs the following:

- The floating-point unit aborts all operations in its pipeline.
- The floating-point unit state machine is set to a known value, indicating that no instructions are in the floating-point unit pipeline.
- The floating-point unit is set to empty, and the floating-point add unit and floating-point multiply unit are set to idle.
- All bits in the FPST register that are not shown as read-only in Figure 3–3, *MP Floating-Point Status Register—FPST*, are cleared to 0, except the fm bit; the fm bit is set to 1 upon reset.

8.1.5 Latencies

Table 8–2 shows estimated latencies (the normal time to generate results) for the various floating-point operations and precisions. The earliest time that subsequent floating-point operations from the same class (see Table 8–1) can normally begin is tabulated in the “start next” column. If mixed-precision inputs are used, the output format determines the cycle count.

- All denormalized operands for the fmpy, fdiv, and vmpy instructions (when not in the fast mode) are piped to the floating-point add unit. For more information, see subsection 8.3.7.
- For the fmpy, fdiv, and vmpy instructions, some normalized operands are piped to the floating-point add unit. For more information, see subsection 8.3.7.
- For mixed-precision adds and subtracts such as fadd.sdd and fadd.dsd, when the single-precision input operand is denormal, the latency and start-next value increase by three in order to normalize the denormal input.
- fmpy, fdiv, vmpy, fsqrt instructions with predictable results caused by special input operands (such as infinity operands) will take three clock cycles to complete.
- If a QNaN (quiet not-a-number) is one of the operands in a fmpy, fdiv, vmpy, or fsqrt instruction, the following table specifies the number of clock cycles.

Instruction	Single Precision	Double Precision
vmpy, fmpy	3 cycles	6 cycles
fdiv, fsqrt	4 cycles	7 cycles

The latencies provided in Table 8–2 are the minimum possible. The latency counts increase when the floating-point multiply unit uses the floating-point add unit pipeline to fix input or output denormals, when the floating-point add unit is needed by another instruction, or when floating-point multiply unit and floating-point add unit produce results on the same clock cycle.

Table 8–1. Classes of Floating-Point and Vector Operations

Class	Instructions	
Floating-point add	fadd	floating-point add
	fcmp	floating-point compare
	frndx	floating-point conversion
	fsub	floating-point subtract
	vadd	vector floating-point add
	vrnd	vector round
	vsub	vector floating-point subtract
Floating-point multiply	fmpy	integer or floating-point multiply
	fdiv	floating-point divide
	fsqrt	floating-point square root
	vmpy	vector floating-point multiply
vmac-type	vmac	vector floating-point multiply and add to accumulator
	vmsc	vector floating-point multiply and subtract from accumulator
	vmsub	vector subtract accumulator from source

Table 8–2. Latencies of Floating-Point Operations

Vector or Floating-Point Operations	Single		Double		Integer	
	Latency	Start Next	Latency	Start Next	Latency	Start Next
fadd, fcmp, fsub, vadd, vsub	4	1	4	1		
frndx, vrnd	4	1	4	1	4	1
fmpy:						
❑ No input denorm	3	1	6	4	3	1
❑ 1 output denorm	6	1	9	4		
❑ 1 input denorm	6	4	9	7		
❑ 2 input denorm	7	5	10	8		
❑ 1 input/1 output denorm	9	4	12	7		
fdiv:						
❑ No input denorm	8	6	22	20		
❑ 1 output denorm	11	6	25	20		
❑ 1 input denorm	12	10	26	24		
❑ 2 input denorm	12	10	26	24		
❑ 1 input/1 output denorm	15	10	29	24		
fsqrt:						
❑ No input denorm	11	9	28	26		
❑ 1 input denorm	15	13	32	30		
vmpy:						
❑ No input denorm	3	1	6	4		
❑ 1 output denorm	6	1	9	4		
❑ 1 input denorm	6	4	9	7		
❑ 2 input denorm	7	5	10	8		
❑ 1 input/1 output denorm	9	4	12	7		
vmac:						
❑ No input denorm	6	1				
❑ 1 input denorm	9	4				
❑ 2 input denorm	10	5				
vmsc:						
❑ No input denorm	6	1				
❑ 1 input denorm	9	4				
❑ 2 input denorm	10	5				
vmsub:						
❑ No input denorm	6	1				
❑ 1 input denorm	9	4				

8.2 The Floating-Point Add Unit

The floating-point add unit is a fully implemented double-precision pipeline. This means that you can start a double-precision operation on every cycle, and the double-precision instruction has the same latency as a single-precision instruction. The floating-point add unit has the following stages:

- Compare the exponents of the two numbers and shift the smaller number right to align the binary points.
- Add/subtract the two numbers.
- Normalize and round the output value.

8.2.1 Comparing Exponents and Shifting

In order to add two mantissas to each other, the exponents must be equal. Therefore, the first step in adding or subtracting two binary numbers is to shift the smaller source right to align the mantissa binary points. The net amount to shift the smaller source (which has the smaller exponent) right is the absolute difference between exponent1 and exponent2.

8.2.2 Adding/Subtracting Two Numbers

Once the mantissa binary points are aligned, the two source operands (source1 and source2) are fed into a standard adder/subtracter. Special hardware predicts the location of the leading 1 in the mantissa, which is used in the normalize step.

8.2.3 Normalizing an Output Value

When the add/subtract is complete, the most significant mantissa bit may not be a 1 (for example, if one mantissa is subtracted from the other, this bit could become a 0). The definition of a normalized number requires the most significant mantissa bit (before packing) to be a 1. The floating-point add unit normalizes the mantissa, which means that the number is shifted left until the most significant mantissa bit is a 1. The number of shifts required to normalize the mantissa was generated previously.

- If the mantissa is shifted left, the exponent must be decremented to reflect the new mantissa (for example, the shift count is subtracted from the exponent). This is satisfactory if the exponent does not go below the minimum exponent value (a biased value of 1).
- If the exponent goes below its minimum value, the number is denormalized.

The left shift is allowed only until the exponent is 1 or the shift has completed. The number coming out of the left shifter is normalized (as much as possible) and is ready to be rounded into the destination precision.

The last block in the normalize and rounding stage takes all of the status information generated by all the blocks in the pipeline stages and generates the output status information required by the IEEE standard. Input status is generated by the unpackers. The FPST register has four status flags defined for the floating-point add unit; these flags include invalid, overflow, underflow, and inexact and are explained in Section 8.4, *Floating-Point Unit Exceptions*, and in subsection 3.3.3.1, *Current Floating-Point Status.*

8.2.4 Rounding an Output Value

The IEEE standard requires that all operations be performed as if using infinite precision and unbounded range, then rounding to the destination format. The right shifter can shift 1s off the end of the mantissa. To conform to the IEEE standard, the right shifter saves three extra bits at the least significant end of the mantissa. The first and second bits are normal, extra bits of precision. The third bit is the logical OR of all of the bits shifted off the end of the mantissa during the right-shift operation.

The logic that performs rounding of results uses the three extended bits of precision (first, second, and third), along with the mantissa's LSB, the sign, and the rounding mode. The rounding modes are defined by the IEEE standard. You can choose from four rounding modes, shown in Table 8–3, by setting the drm field in the FPST register.

The round-to-nearest and the round-to-zero (truncate) modes are the most common rounding modes.

- The round-to-zero mode is simple; the input mantissa is truncated to the destination precision.
- The round-to-nearest mode rounds to nearest, or to even, if exactly halfway. In the round-to-nearest mode, the mantissa is incremented if the following equation is true:

  round_up = 1st AND (2nd OR 3rd OR LSB)

The directed rounding modes (both positive and negative infinity) are normally used to control round-off error in a predictable manner.

Table 8–3. Floating-Point Rounding Modes—drm Field in the FPST Register

drm Field	Rounding Mode
0	N = round toward nearest (default at reset)
1	Z = round toward zero (truncate)
2	P = round toward positive infinity
3	N = round toward negative infinity

8.3 The Floating-Point Multiply Unit

The floating-point multiply unit is double-precision with a single-precision core. Multiple cycles are required to perform single- and double-precision multiplies and both single- and double-precision divides and square roots, but a new single-precision multiply can be introduced into the pipeline on each new clock. The floating-point multiply unit handles numbers with normalized mantissas directly but needs the assistance of the floating-point add unit to handle denormalized numbers. As a result, denormalized inputs or outputs impose an execution time penalty. The mechanism for handling denormal numbers is discussed in subsection 8.3.7.

Since the floating-point multiply unit has a number of longer instructions, the unit frequently cannot start a new instruction on every cycle. However, the following instructions can start on every clock: fmpy.iii, fmpy.uuu, fmpy.sss, vmac.sss, vmac.ssd, vmsc.sss, vmsc.ssd, vmpy.ss, and vmsub.ss (assuming that there are no exceptions or denormals). Table 8–2 lists the floating-point latencies.

The floating-point multiply unit has the following stages after the execution unit's unpack stage:

- Perform 32-bit integer or single-precision floating-point multiplication
- Normalize/round the output value

This section also covers the following operations:

- Double-precision multiply
- Divide and square root
- Integer multiply
- Other IEEE-754 operations

Also covered in this section is a discussion about denormals in the multiplier (subsection 8.3.7).

8.3.1 Performing Single-Precision Floating-Point Multiplication

You can perform single-precision floating-point multiplication by adding the two exponents to get the output exponent, and by multiplying the two mantissas. At this point, the correct exponent and a nearly normalized mantissa are available.

Multiplication uses a modified Booth algorithm to provide a 32-bit product in one cycle.

The input mantissas' range is $1 \leq$ mantissa < 2, so the resulting multiply's range is $1 \leq$ output < 4. Therefore, the floating-point multiply unit normalization stage is only a one-bit shift, compared to the multibit shift in the floating-point add unit.

8.3.2 Normalizing and Rounding the Output Value

The next stage of the floating-point multiply unit normalizes the mantissa and rounds the output value. During the normalize and rounding stage, the mantissa can be greater than 2. If it is, the mantissa is shifted right by one bit, and the exponent is incremented.

The rounding stage is similar to that of the floating-point add unit but requires only two extra bits of precision: the first bit and the second bit. During any multiply, the results have more bits than the inputs. A single-precision floating-point multiply is 24×24, resulting in 48 bits. The lower 23 bits are ORed together to form the second bit. The upper 24 bits plus the second bit are kept as the mantissa. The rounding is like that of the the floating-point add unit.

The normalize stage also generates the output status. The status bits in the FPST register for the floating-point multiply unit are explained in Section 8.4 and include invalid, overflow, integer multiply overflow, underflow, inexact, and divide-by-zero.

8.3.3 Double-Precision Multiply

The floating-point multiply unit has a 32 × 32 array for use in integer × integer calculations, which makes double-precision multiply an iterative process. To generate a double-precision result, the double-precision inputs are divided into upper and lower halves. Then multiplies are performed and summed as:

result = (upper1 × upper2) + (upper1 × lower2) + (lower1 × upper2) + (lower1 × lower2)

Even though the least significant half of the result is discarded during rounding, all four multiples must be performed in order to ensure proper IEEE rounding. The exponent path and the normalize stage of the floating-point multiply unit are full double-precision widths, so no iterations are needed through these sections.

8.3.4 Divide and Square Root

The MP uses the following methods for performing divides and square roots:

- The convergence method uses a series of multiplications to approximate a divide.
- The square root function is based on the same type of logic as the divide, but has a different set of equations.

Both the divide and square root functions require the mantissa of floating-point operand to be between 1 and 2 in order to converge with enough precision to round properly. If a number is denormal, it is not always within these bounds. As in the multiply instruction, the denormal operands must be wrapped to be operated on. The divide can result in a wrapped number that must be denormalized in the floating-point add unit. However, the square root can never generate a denormal output.

For more information about wrapped numbers, see Section A.6, *Wrapped Numbers*.

8.3.5 Integer Multiply

For the signed and unsigned integer multiply instructions, only the 32 LSBs of the result are output from the floating-point multiply unit. For a description of integer multiply when overflow occurs, see subsection C.5.7, *Floating-Point Multiply Overflow of Integers*.

Integer multiply overflow is signaled by FPST⟦mo⟧ = 1.

8.3.6 Other IEEE-754 Operations

The IEEE-754 standard also defines a remainder operation, conversion between floating-point and decimal strings, and conversion of floating-point to integral value. All these operations must be performed by library routines.

Note that decimal strings are intended as human-readable forms; the C library routines fprintf() and fscanf() comply with the IEEE-754 standard, provided that they have the correct behavior.

8.3.7 Denormals in the Multiplier

If inputs to the floating-point multiply unit are denormal numbers, some of the assumptions made about input operands are no longer true. Mainly, the mantissa is no longer between the values 1 and 2. Since the biased exponent is already at its minimum value (biased value of 1), when the mantissa is normalized by the ALU, it first becomes zero and then negative. This is referred to as a wrapped number. To avoid making the Booth adder array much larger to handle these denormal numbers, denormal numbers are normalized. With the exponent wrapped and the mantissa normalized, the multiply can take place normally, as long as the exponent hardware handles the wrapped number's exponent as a negative number.

If the result of a floating-point multiply operation is a wrapped number, then it is unwrapped and rounded into a proper denormalized number. The unwrap operation is a right shift of the mantissa until the exponent has reached a biased value of one.

All floating-point multiply denormal/wrap handling is done in the floating-point add unit to save having extra shifters in the floating-point multiply unit just for denormal/wrapped numbers. The hardware sends the denormal/wrapped numbers into the input of the floating-point add unit to be processed.

Floating-point multiply unit instructions that have input denormal or produce output denormals take extra clock cycles to complete. Refer to Table 8–2 for the latency values.

Some wrapped results that the floating-point multiply unit produces are so small in value that it is not necessary to send them to the floating-point add unit in order to produce the correct IEEE rounded result:

- For single-precision results, if the resultant's exponent is less than the unbiased value of –150, then the floating-point multiply unit produces the correctly rounded IEEE result without sending the result to the floating-point add unit.
- For double-precision results, if the resultant's exponent is less than the unbiased value of –1075, then the floating-point multiply unit produces the correctly rounded IEEE result without sending the result to the floating-point add unit.

Wrapped results from the floating-point multiply unit are sent to the floating-point add unit to be denormalized as stated above. In addition, some normalized results from the floating-point multiply unit are sent to the floating-point add unit also.

For vmpy and fmpy instructions, if the sum of the input operand's exponents is equal to an unbiased –127 for single-precision results (–1023 for double-precision), then the result operand is sent through the floating-point add unit, even if the resultant exponent is not wrapped. –127 and –1023 are the first wrapped exponent values, as shown in Example A–13, *Wrapped Numbers in Single-Precision Format*, and Example A–14, *Wrapped Numbers in Double-Precision Format*.

The decision to send floating-point multiply unit results to the floating-point add unit is based on the intermediate exponent and not the final exponent; therefore, some floating-point multiply results that are sent to the floating-point add unit fall in the normal range of numbers. Although the sum of the input exponents may fall in the wrapped range, the mantissa multiply can result in a mantissa ≥ 2. In this case, the floating-point multiply unit does a one-bit shift right to normalize the mantissa. The intermediate exponent is incremented by one (to compensate for the shift) in order to get the final exponent. Since the intermediate exponent is the first wrapped exponent, the increment makes it the smallest normal exponent.

Wrapped results that are sent to the floating-point add unit are denormalized and rounded. Normal results that are sent to the floating-point add unit are only rounded.

If denormal inputs to the multiplier are detected at the same time that a wrapped number is generated in the multiplier, then the wrapped output from the floating-point multiply unit is first fixed by the floating-point add unit before the denormal inputs of the floating-point multiply unit are fixed by the floating-point add unit.

The floating-point unit signals fpmpy_busy while the floating-point multiply unit is waiting on the input denormals to be normalized; therefore, no new floating-point multiply instruction can be sent to it.

If the floating-point add unit or floating-point multiply unit is busy, this causes the MP to stall if a floating-point unit instruction is fetched that needs the corresponding busy floating-point unit.

For more information about the effect of denormals on the floating-point unit pipeline, see Appendix B, *Pipeline Implications of Floating-Point Denormals*.

8.4 Floating-Point Unit Exceptions

The IEEE exceptions are divided into two classes: input exceptions and output exceptions. For a complete definition of the IEEE exceptions and the results that should be returned for each exception, refer to the IEEE-754 standard.

See subsection 3.3.3.1, *Current Floating-Point Status*, for more information about the floating-point exception bits in the FPST register.

8.4.1 Input Exceptions

Input exceptions can be detected in the early part of a floating-point unit operation. Examples of input exceptions are the invalid flag and the divide-by-zero flag.

- An **invalid** exception is set for:
  - Operations on a SNaN (signaling not-a-number)
  - Magnitude subtraction of positive infinities: $(+\infty)-(+\infty)$ or $(+\infty)+(-\infty)$ or $(-\infty)-(-\infty)$
  - Conversions of binary floating-point numbers to signed or unsigned integers when infinity, overflow, or NaN (not-a-number) precludes a faithful representation in that format
  - Conversion of any nonzero negative number to an unsigned integer
  - $0\times\infty$ or $0\div 0$ or $\infty\div\infty$
  - Square root of an operand that is less than zero, except that $\sqrt{}-0 = -0$
- A **divide-by-zero** exception is set when the divisor is zero, and the dividend is a finite nonzero number.

8.4.2 Output Exceptions

Output exceptions cannot be detected until the final result is known. Examples of output exceptions are overflow, underflow, inexact, integer-multiply overflow, and invalid due to overflow during conversions to integer.

- An **overflow** exception is set when the maximum exponent has been exceeded.
- An **inexact** exception is set when any loss of precision has occurred. This is simply the logical OR of the three extra bits in the floating-point add unit rounder or the two extra bits in the floating-point multiply unit rounder.
- An incomplete definition of the **underflow** exception is when tininess occurs (that is, when a tiny value results). Refer to the IEEE-754 standard for a complete definition of underflow.
- An **integer-multiply overflow** exception is set during floating-point multiply (fmpy) instructions when a signed or unsigned integer is the destination precision and the answer exceeds 32 bits.
- An **invalid exception** is set due to overflow during conversions to integer.

8.4.3 Traps and Interrupts Associated With Exceptions

There is a corresponding interrupt enable bit for each of the floating-point unit exceptions, except integer-multiply overflow (FPST[[mo]]). For a given exception, the behavior of the floating-point unit and fetch pipeline is dependent upon whether or not the interrupt enable bit is set. The various behaviors are explained in Appendix C, *Floating-Point Unit Exceptions*.

- **It is illegal to issue a floating-point unit instruction with interrupts masked (IE[[ie]] = 0).** If this happens, a floating-point error trap 36 will occur, even though interrupts are disabled. This is a nonrecoverable error.

- **It is illegal to issue a floating-point unit instruction when FPST[[fs]] equals 1** (floating-point unit pipeline is stalled). If this happens, a floating-point error trap 36 will occur. This is a nonrecoverable error. The floating-point unit will set FPST[[fs]] to 1 only when a floating-point exception interrupt has been taken. While servicing the interrupt, an attempt to execute a floating-point instruction (with the floating-point pipeline stalled) is treated as an error.

  If a floating-point exception interrupt handling routine has not been taken, the floating-point unit never sets the fs bit; however, you can set it by using the swcr/wrcr (swap/write control register) command. If you set the fs bit by using the swcr/wrcr instruction and floating-point unit instructions are issued, the conditions for this error are met.

Although the invalid exception is usually treated as an input exception, invalid operations due to overflow during conversions to integers are treated as output exceptions. Refer to subsection C.5.7, *Floating-Point Multiply Overflow of Integers*.

For more information about floating-point exceptions and their results, see Appendix C, *Floating-Point Unit Exceptions.*

Note:

Integer multiplies require the floating-point multiply unit and can only be performed if IE[[ie]] = 1 and FPST[[fs]] = 0.

8.5 Floating-Point Unit Modes of Operation

You can choose to execute floating-point instructions with two additional options:

- **Fast mode**, in which all denormals are treated as zero for both input operands and the output result (see subsection 8.5.1).
- **Sequential mode**, in which the floating-point unit executes a single floating-point instruction to completion before starting another floating-point instruction (see subsection 8.5.2).

8.5.1 Fast Mode Versus IEEE Mode for Denormal Numbers

Since handling denormals is time consuming, the fast mode option allows a denormal to be treated as a value of zero. This applies to both denormal operations and denormal results. All floating-point instructions can operate in the fast mode.

- When the unit is operating in the IEEE (non-fast) mode, there are **three** magnitudes of numbers that are not special (numbers that **are** special include NaNs and infinity numbers); these numbers are normal numbers, denormal numbers, and smaller-than-denormal numbers.
- In the fast mode, there are **two** magnitudes of numbers that are not special: normal numbers and smaller-than-normal numbers.

You can select the fast mode by using the FPST⟦fm⟧ bit (see subsection 3.3.3, *Floating-Point Status Register: FPST*, for more information about the fm bit):

- If fm = 0, then the floating-point unit operates in the fast mode and forces denormals to zero.
- If fm = 1, then the floating-point unit operates in the IEEE mode and returns IEEE denormals (fm = 1 at reset).

If the floating-point unit operates in the fast mode, then all denormal input operands are treated as zero (even if the input operand is single-precision and the result operand is double-precision), and the operands are subject to the same invalid and divide-by-zero exception conditions as a zero is subject to. This is true, independently of the values for the floating-point underflow or inexact interrupt enables.

While the floating-point unit is in the fast mode, output denormals are handled differently if either the floating-point underflow interrupt or the floating-point inexact interrupt is enabled. These differences are explained in subsequent paragraphs.

If neither the floating-point underflow interrupt nor the floating-point inexact interrupt is enabled, then all floating-point unit results that have magnitudes less than the smallest normalized number after rounding are changed to a zero. The sign remains unchanged. Those numbers that are flushed to zero have magnitudes that are smaller than the smallest representable number in this mode (that is, the smallest normalized number); therefore, the floating-point status inexact and underflow are signaled.

When the floating-point underflow interrupt is enabled, all results that are denormal after rounding will signal an underflow. Therefore, all denormal results, including some results that are normalized after rounding, cause an underflow interrupt. Enabling interrupt handling routines in the fast mode defeats the purpose of the fast mode because the saving of a few clock cycles is outweighed by the lengthy interrupt handling routine. Nonetheless, floating-point interrupt enables are **not** ignored during the fast mode. While in the fast mode, the floating-point unit delivers different results and exceptions if the floating-point underflow and/or inexact interrupts are enabled.

If the floating-point unit is in the fast mode and the floating-point underflow interrupt is enabled (or the underflow and the inexact traps are both enabled), for instructions that are unaffected by denormal inputs being treated as zero, the floating-point unit will return the same result and exceptions to the underflow trap handling routine as in the IEEE mode. This means that numbers with magnitudes that are smaller than the smallest normalized number will **not** be flushed to zero. This is done because all of the numbers that would have been flushed to zero would have been flagged underflow, regardless of their exactness, causing the underflow trap handling routine to be executed. To provide the most useful information to the trap handler, the results are **not** flushed to zero.

If the floating-point unit is in the fast mode and the floating-point inexact interrupt is enabled (without the underflow interrupt being enabled), for instructions that are unaffected by denormal inputs being treated as zero, the floating-point unit returns the same result and the same exceptions to the inexact interrupt handling routine as in the IEEE mode, with one difference. All denormal results (after rounding) signal inexact, regardless of their exactness. If numbers that have a magnitude smaller than the smallest normal number would have been flushed to zero, they would have become inexact (regardless of their original exactness), causing the inexact interrupt handler to be executed.

To provide the most useful information to the interrupt handler, the results are **not** flushed to zero. Inexact is signaled, even for exact denormal results, because if that number had been flushed to a zero, it would have been inexact. Also, if an exact denormal result is not flushed to zero and the inexact flag is not set, the floating-point unit returns a denormal result in the fast mode, which is contrary to the purpose of the fast mode.

Handling denormal results in this manner allows you to determine through the inexact interrupt enable when smaller than normal results would have been flushed to zero. If this is unimportant, then you can disable the inexact interrupt. If it is important, enable the inexact interrupt, and the inexact interrupt handling routine can check to see if the floating-point unit is in the fast mode and if the result is denormal. In this way, the inexact interrupt handling routine can take care of IEEE inexact interrupt handling and denormal interrupt handling for the fast mode.

8.5.2 Floating-Point Sequential Mode Versus Pipelined Mode

A potential difficulty in debugging the exception mechanisms described in Appendix B, *Pipeline Implications of Floating-Point Denormals*, and Appendix C, *Floating-Point Unit Exceptions*, is that it is hard to relate an output exception back to the instruction that caused it. This problem does not exist in the sequential mode. This mode is controlled by the FPST⟦sm⟧ bit (see subsection 3.3.3, *Floating-Point Status Register: FPST*, for more information about the sm bit):

- If sm = 0, then the floating-point unit operates in the pipelined (nonsequential) mode (sm = 0 at reset).
- If sm = 1, then the floating-point unit operates in the sequential mode.

The sequential mode causes floating-point unit instructions to stall the MP's execution unit until they complete, allowing only one instruction into the floating-point unit at a time. Its main use is in debugging because it allows you to precisely determine the instruction that causes a problem.

Also, in order for the invalid and divide-by-zero interrupt handlers to access input operands that cause exceptions, the floating-point unit must be in sequential mode to ensure that the address of the exception instruction is in register EIP when the interrupt is taken.

The sequential mode affects the floating-point register write enable. When an instruction with an input exception completes with its corresponding interrupt enabled, the write to the register is inhibited in the sequential mode. This prevents the result of an input exception instruction from destroying the contents of a register that contains the input operand that caused the exception. The interrupt handlers may want access to the input operand. In the sequential mode, the register write is unaffected by conversions from floating-point format to integer when overflow occurs (causing invalid) because this is classified as an output exception. The sequential mode never affects the FPST write enable.

If the floating-point unit is in sequential mode, then the integer unit stalls (in execute) every time a floating-point instruction is sent to the floating-point unit. It remains stalled until the floating-point instruction completes or signals an exception.

You should **not** code parallel loads for floating-point unit vector instructions while in the sequential mode because the load to the register file could destroy the contents of a register containing an input operand that is needed in the interrupt handling routine. The parallel load is allowed, but if the load to the register file destroys the contents of a register that is needed in the interrupt handling routine, the interrupt handler may be unable to determine the value of the operand that caused the exception.

8.6 Floating-Point Unit Interrupt Handlers

The recommended procedure to follow for floating-point exception interrupt handling routines is shown in subsection 11.11.1, *Recommended Floating-Point Exception Interrupt Procedure*.

A typical scenario for a floating-point exception interrupt handling routine is as follows: IE⟦ie⟧ is 1; otherwise, a floating-point unit instruction cannot start. A floating-point exception, with its corresponding interrupt enable set, is detected by the floating-point unit. FPST⟦fs⟧ is set to 1, and the floating-point unit stalls. The MP's FEA pipeline may be stalled for some reason (for example, a data-cache miss occurring at the time that a floating-point exception interrupt handling routine is detected by the floating-point unit). Therefore, the floating-point exception interrupt handling routine may not begin execution immediately.

The MP's interrupt logic copies the PC to the EPC and the IP to the EIP. The return value of the ie bit (1 for floating-point operations and/or interrupts globally enabled) is copied into the LSB of the EPC at the same time that the PC is copied. The MP's execution unit branches to the appropriate floating-point exception interrupt handling routine. The routine clears the current interrupt out of the INTPEN register so that, on the return from the routine, the same interrupt is not taken again. The floating-point exception interrupt handling routine saves the context of the interrupted program into memory. This context-save saves a value of 1 for FPST⟦fs⟧, a 0 for IE⟦ie⟧, and a 1 for the return value of ie in the LSB of EPC. The main body of the floating-point exception interrupt handling routine is then executed.

Since more than one floating-point exception (with its corresponding interrupt enabled) can be generated by a single floating-point instruction, the interrupt routine must choose whether to simply clear (and ignore) any additional exceptions, or to attempt to handle the remaining exceptions on an individual basis. If any other floating-point exception interrupt handling routine is to be executed, the current routine should not clear the FPST⟦fs⟧ bit but should restore the context of the interrupt program. Then the floating-point exception interrupt handling routine should execute the interrupt return sequence (that is, branch to EIP and then branch to EPC), whether or not the unit is in the sequential mode (see the interrupt handling routine on page MP:11-35).

After the body of the floating-point exception interrupt handling routine has been executed, if you do not want to execute any other floating-point exception interrupt handling routine to handle the other enabled exceptions for the current instruction, be sure that the current routine being executed clears all of the other pending floating-point interrupts from INTPEN. Next, the context of the interrupted program should be restored. Then use the swcr/wrcr instruction to clear the FPST⟦fs⟧ bit.

- If in the sequential mode, branch EPC only. Do **not** branch EIP; use a nop instruction in the delay slot instead.
- If not in the sequential mode, branch EIP and then branch EPC.

If a floating-point exception interrupt-handling routine enables other interrupts by setting IE⟦ie⟧ to 1, then neither the floating-point exception interrupt-handling routine nor any interrupt that is serviced should change the value of FPST⟦fs⟧ to 0. After a floating-point exception interrupt-handling routine is initiated, the fact that fs = 0 and ie = 1 signals to the execution unit and to the floating-point unit that the routine has completed.

- If this condition arises before the completion of a floating-point exception interrupt-handling routine, then the floating-point unit unstalls prematurely.
- If the floating-point unit unstalls before the real completion of the floating-point exception interrupt-handling routine, then unpredictable results will occur in the floating-point unit and execution unit. However, this condition will **not** cause the floating-point error trap 36. It is left up to you to ensure that after a floating-point exception interrupt-handling routine has begun, the conditions that signal the completion of that routine (ie = 1 and fs = 0) do not occur before the actual completion.

If the MP is **not** executing a floating-point exception interrupt handling routine, interrupts and non-floating-point unit exceptions wait for the floating-point pipeline to empty before they are serviced. The FEA pipeline stalls when the exception is recognized and remains stalled until the floating-point unit signals that it is empty or that a floating-point exception has occurred. Any floating-point exceptions are then handled before other exceptions, unless the floating-point exception interrupt handling routine enables other interrupts.

If the MP **is** executing a floating-point exception interrupt handling routine that has enabled other interrupts and an enabled interrupt is signaled, the MP services the interrupt at that time, even if the floating-point unit is not empty.

Note:

Floating-point interrupt service routines should not use floating-point instructions or enable other interrupt service routines that use floating-point instructions.

8.7 Floating-Point Unit Busy Signals

Two signals indicate whether the floating-point unit is busy and cannot accept a new floating-point unit instruction:

- **fpadd_busy** indicates that the floating-point add unit is busy.
- **fpmpy_busy** indicates that the floating-point multiply unit is busy.

fpadd_busy or fpmpy_busy causes the execution unit to stall if a floating-point unit instruction is fetched that needs the corresponding busy floating-point unit. Floating-point multiply unit instructions with input denormals require the floating-point add unit; therefore, fpadd_busy would stall the MP's execution unit.

vmac-type instructions require the floating-point add unit. In order to ensure that the floating-point add unit is free when needed, a vmac-type instruction makes the fpadd_busy signal active at the beginning of its execution. fpadd_busy remains active until the vmac-type instruction has finished using the align stage (stage 0) of the floating-point add pipeline.

The fpadd_busy and fpmpy_busy signals are set because of a floating-point unit exception interrupt one cycle after the exception is signaled. The signals remain set for the entire floating-point exception interrupt handling routine and for one cycle after the routine has completed.

Note:

These are internal signals; they are not user-visible.

8.8 Changing Control Registers

If interrupts are enabled and a floating-point exception interrupt handling routine is not being executed, the swcr (swap control register) and wrcr (write control register) instructions wait for the floating-point unit to complete all previously initiated floating-point operations before allowing modification of the target control register. This interlock prevents any change in the values that the IE[[ie]] bit, the individual floating-point exception bits in the IE register, and the mode bits in the FPST register had when a floating-point instruction was initiated. It also guarantees that all floating-point results have been written to their destination registers before the instruction that follows the swcr or wrcr instruction begins to execute.

If a floating-point exception interrupt handling routine is initiated while the FEA pipeline is stalled and waiting for the floating-point pipeline to empty so that it can execute an swcr/wrcr instruction, that swcr/wrcr instruction is not executed until that interrupt handling routine is complete and the floating-point unit is empty.

Control registers can be modified when interrupts are enabled and a floating-point exception interrupt handling routine is executing because that routine must be able to modify the FPST register in order to clear the fs bit.

Control registers can be modified when interrupts are disabled. However, floating-point unit instructions can be initiated only while interrupts are enabled (as stated in Section 8.4). Therefore, if a floating-point exception interrupt handling routine is executing or has been interrupted, the floating-point unit can be “not empty” only when interrupts are disabled. It is assumed that the handlers can detect whether or not they need to change the floating-point exception enable bits in the IE register or any bits in the FPST register during a floating-point exception interrupt handling routine (or an interrupted routine); doing so would not prevent the case in which a floating-point unit instruction is initiated with one set of enables or modes and then completed with another.

An swcr or wrcr instruction executed by a routine that interrupts a floating-point exception interrupt handling routine follows the same rules (as far as the swcr/wrcr waiting/not waiting for the floating-point unit to be empty) as if the floating-point exception interrupt handling routine is actually being executed.

If a floating-point exception interrupt handling routine enables other interrupts by setting IE[[ie]] to 1, then neither that routine nor any additional interrupt routines that execute before the first routine completes should change the value of FPST[[fs]] to 0. The floating-point pipeline stalls on the condition fs = 1 or ie = 0. If the interrupt service routine both clears fs and sets ie, the pipeline immediately unstalls.

- If this condition arises before the completion of a floating-point exception interrupt-handling routine, then the floating-point unit will unstall prematurely.
- If the floating-point unit unstalls before the real completion of the floating-point exception handling routine, then unpredictable results may occur in the floating-point unit and execution unit. However, this condition will **not** cause the floating-point error trap 36. It is left up to you to ensure that after a floating-point exception interrupt-handling routine has begun, the conditions that signal the completion of that routine (ie = 1 and fs = 0) do not occur before the actual completion of the floating-point exception interrupt-handling routine. Refer to Section 11.11.1, *Recommended Floating-Point Exception Interrupt Procedure*, for a description of a typical floating-point exception interrupt-handling routine.

Chapter 9

Interrupts, Traps, and Reset

This chapter explains the MP interrupts and traps that affect instruction execution. Also covered in this chapter are resets and operating modes.

Topics

9.1 Exceptions—Interrupts and Traps

The MP has two ways of responding to exceptional conditions: interrupts and traps. Both interrupts and traps cause the normal instruction flow to be interrupted and divert control to a routine that services the exception.

- Interrupts are exceptions caused either by conditions outside the instruction stream or by previously executed instructions.

- In contrast, a trap occurs when the instruction that is being executed by the MP causes an exception.

While these distinctions are useful for understanding how interrupts and traps are likely to be used, the dividing line between interrupts and traps may be more difficult to define in practice.

Traps and interrupts are closely related. They differ primarily in the mechanisms they use to return from an exception service routine.

- A return-from-interrupt restarts the instruction whose execution was interrupted. This is the instruction whose address was saved in the EIP register when the interrupt was taken.

- A return-from-trap, however, resumes execution at the instruction that follows the instruction that caused the trap. The resumed instruction's address is saved in the EPC register when the exception occurred.

If a trap service routine is mistakenly terminated with an interrupt-style return, the instruction that caused the trap is executed again, causing another trap. On the other hand, incorrectly terminating an interrupt service routine with a trap-style return causes the interrupted instruction to be skipped entirely.

Whether a particular exception is treated as a trap or an interrupt may depend on circumstances. For example, if the MP attempts to write data to an illegal on-chip memory address, the access causes a memory-fault exception. The memory-fault service routine may treat this as a fatal error, in which case control is never returned to the interrupted program. The same service routine, however, may choose to simply ignore an attempted read of an illegal on-chip location; in this case, the service routine uses a trap-style return to avoid executing the faulting instruction a second time.

In a system supporting virtual memory, the memory-fault routine may treat a faulting access of the MVP's external memory as a demand to swap in a new memory page from disk. Once the new page has been downloaded, the memory-fault routine performs an interrupt-style return in order to retry the instruction that caused the fault.

Another example is a floating-point exception, which can be treated as a trap or an interrupt, depending on circumstances. When the floating-point unit is operating in sequential mode, a floating-point exception should, in general, be treated as a trap. This is because the floating-point instruction that caused the exception is pointed to by the EIP saved when the exception occurs. When the floating-point unit is operating in pipelined mode, however, the instruction pointed to by the EIP is a successor to the floating-point instruction that caused the exception. In this case, an interrupt-style return is needed. For more information about handling floating-point exceptions, see Section 11.11, *Floating-Point Exception Interrupt Handler Routines.*

9.2 Managing Exceptions

When an exception occurs, it preempts the execute stage of the FEA pipeline. If the MP is **not** executing a floating-point interrupt service routine, interrupts and nonfloating-point unit exceptions wait for the floating-point unit to be empty before they are serviced. The FEA pipeline stalls when the exception is recognized and remains stalled until the floating-point unit signals **empty** or a floating-point exception. Any floating-point exception is then handled before other exceptions, unless the floating-point interrupt service routine enables other interrupts. This prevents the delayed writes of the floating-point unit from modifying exception handler registers rather than those of the context in which the floating-point operation was issued.

If the MP is executing a floating-point interrupt service routine and that routine has enabled other interrupts and an enabled interrupt is signaled, the MP services the interrupt at that time, even if the floating-point unit is not empty. Floating-point exceptions are serviced before other exceptions because they have a higher priority.

Floating-point exception service routines may require access to the original operands. The integrity of the operands in the source registers is preserved if the floating-point unit is run in the sequential mode (FPST⟦sm⟧ = 1, as shown in Figure 3–3, *MP Floating-Point Status Register—FPST*). This helps you in debugging your program because the floating-point exceptions are **not** written into the destination register.

In addition, allowing the floating-point pipeline to become empty is useful because some exception handlers need to perform integer multiply or modulo operations, both of which require the floating-point unit. This assumes that the floating-point units are idle and the IE⟦ie⟧ bit is set.

When an exception occurs, the state of the interrupted program is saved by copying the PC (program counter) and the IP (instruction pointer) into the corresponding exception versions of these registers (EPC and EIP, respectively). The state of the interrupt enable bit (IE⟦ie⟧) is saved in bit 0 of the EPC register. The state of the user mode bit (um) is saved in bit 1 of the EPC register (see Figure 9–1).

Interrupts are then disabled by clearing the IE⟦ie⟧ bit to 0, and the user mode bit is cleared in order to enter the supervisor mode (user mode bit is 0). The exception (or trap) number *N* is used to index into the vector transfer address table, and that value is loaded into the PC as address(0x0101 0180 + 4*N) $\rightarrow$ PC.

When a higher-priority trap or interrupt occurs while another trap/interrupt is being serviced, you must save the values in the EIP and EPC registers (on the stack) and re-enable interrupts if this is required. If EIP and EPC are not saved, any trap instructions used in the exception handler destroy the return context.

Note:

Do not use trap 32–35 instructions in your program. These instructions are reserved for TI emulation.

9.2.1 Interrupt and Trap Vector Addresses

The MP's interrupts and traps (and their vector addresses) are shown in Table 9–1 and Table 9–2, respectively. The IE register is graphically shown in Figure 3–2, *MP Interrupt Registers—IE and INTPEN*.

The integer overflow bit (io) is bit 4 in the IE and INTPEN registers and uses location 0x0101 0190 in preproduction silicon versions 1 and 2.

Table 9–1. Maskable Interrupt Priorities and Vector Addresses

	IE Bit	Name	Vector Address	Maskable Interrupts—Highest Priority Is 0
Highest Priority	0	ie	0x0101 0180	Nonmaskable global interrupt enable
	2	fi	0x0101 0188	Floating-point invalid interrupt
	3	fz	0x0101 018C	Floating-point divide by zero interrupt
	5	fo	0x0101 0194	Floating-point overflow interrupt
	6	fu	0x0101 0198	Floating-point underflow interrupt
	7	fx	0x0101 019C	Floating-point inexact interrupt
	8	f0	0x0101 01A0	VC frame timer 0 interrupt when VCOUNT0 = VFTINT0
	9	f1	0x0101 01A4	VC frame timer 1 interrupt when VCOUNT1 = VFTINT1
	10	ti	0x0101 01A8	MP timer interrupt when TCOUNT decrements to 0
	11	x1	0x0101 01AC	External interrupt 1
	12	x2	0x0101 01B0	External interrupt 2
	14	mf	0x0101 01B8	Memory fault ❑ Nonmaskable: MP instruction-/data-cache faults; MP load/store (DEA or on-chip) faults ❑ Maskable: PP instruction-cache faults; PP load/store (DEA or on-chip) faults; all MP/PP packet request address faults
	15	io	0x0101 01BC	Integer overflow
	16	p0	0x0101 01C0	PP0 message interrupt
	17	p1	0x0101 01C4	PP1 message interrupt
	18	p2	0x0101 01C8	PP2 message interrupt
	19	p3	0x0101 01CC	PP3 message interrupt
	20	p4	0x0101 01D0	Reserved for PP4 message interrupt
	21	p5	0x0101 01D4	Reserved for PP5 message interrupt
	22	p6	0x0101 01D8	Reserved for PP6 message interrupt
	23	p7	0x0101 01DC	Reserved for PP7 message interrupt
	25	mi	0x0101 01E4	MP message interrupt
	26	pc	0x0101 01E8	Packet transfer complete
	27	pb	0x0101 01EC	Packet transfer busy
	28	bp	0x0101 01F0	Bad packet transfer (error)
Lowest Priority	29	x3	0x0101 01F4	External interrupt 3
	30	x4	0x0101 01F8	External interrupt 4
	31	pe	0x0101 01FC	PP error (illegal instruction)

Notes: 1) IE bit numbers < 32 that are not listed are reserved.

2) For a definition of floating-point exceptions, see Section 8.4, *Floating-Point Unit Exceptions*.

3) A nonmaskable interrupt (mf) will always be taken, regardless of IE⟦ie⟧ and IE⟦mf⟧ values.

4) A maskable memory-fault interrupt service depends on the values of IE⟦ie⟧ and IE⟦mf⟧.

Table 9–2. Nonmaskable Trap Priorities and Vector Addresses

Number	Name	Vector Address	Traps (All Traps Are Nonmaskable)
32	e1	0x0101 0200	TI emulator trap 1 (reserved for TI emulation software)
33	e2	0x0101 0204	TI emulator trap 2 (reserved for TI emulation software)
34	e3	0x0101 0208	TI emulator trap 3 (reserved for TI emulation software)
35	e4	0x0101 020C	TI emulator trap 4 (reserved for TI emulation software)
36	fe	0x0101 0210	Floating-point error trap when a floating-point instruction is issued with IE⟦ie⟧ = 0 or FPST⟦fs⟧ = 1
37		0x0101 0214	Reserved for future MVP products
38	er	0x0101 0218	Error illegal MP instruction (for example, all 0s or 1s)
39		0x0101 021C	Reserved for future MVP products
40 to 71		0x0101 0220 to 0x0101 029C	Reserved for externally or VC frame timer-initiated packet transfer buffer (see bits CONFIG⟦X⟧ and FMEMCTL0/1⟦P⟧)
72		0x0101 02A0	System or user defined
–		– – –	
415		0x0101 07FC	System or user defined

Notes: 1) The illegal instruction trap (trap 38) is nonmaskable and loads PC as memory(0x0101 0180+38 × 4) → PC, independent of the value of IE⟦ie⟧.

2) For a definition of floating-point exceptions, see Section 8.4, *Floating-Point Unit Exceptions*.

The address of a vector is calculated by shifting the interrupt number left by two bits and adding the result to the vector base address 0x0101 0180. This sum forms a 32-bit aligned address of the interrupt vector location.

The on-chip parameter RAM is accessed directly, bypassing cache. Fetching the interrupt vector transfer address does not disturb the data cache for addresses < 0x0101 0800.

Typically, at least one trap is used to call the operating system; trap 0 is a good choice. You cannot use the vector at 0x0101 0180 for an interrupt, because it maps to the global interrupt-enable bit (IE⟦ie⟧).

See also:

- PPERROR register in Figure 3–4, *MP Register Showing PP Errors—PPERROR (Read Only)*
- FPST register in Figure 3–3, *MP Floating-Point Status Register—FPST*
- FLTSTS register in Figure 2–9, *Transfer Controller Fault Status Register—FLTSTS*

9.2.2 Trap Mechanism

When a trap is recognized, the PC update that would normally occur in that cycle is suppressed, and the instruction fetched in that cycle discarded. The unincremented PC is copied into the EPC (exception program counter). The address of the instruction that caused the trap (contained in the IP) is stored in the EIP (exception instruction pointer).

Table 9–2 lists the traps and their vector transfer addresses. Sample trap instructions are shown in Example 9–1.

Example 9–1. Trap Sample Code

```
bcnd.a  Not0,r9,ne0.w  ; is r9 = 0?
trap    38             ; service illegal instruction
                       ; PC = Memory(0x01010218)
xnor    r9,r9,r9       ; service routine returns to this xnor

                       ; assume 72 = 0x010102A0 is a user trap,
trap    72             ; call trap 72 service routine
                       ;   PC = Memory(0x010102A0)
addu    r4,r5,r6       ; service routine returns to this addu
. . .

; Return from a trap (skips trap instruction at EIP)

brcr    EPC            ; Resume operation at EPC
nop                    ; Branch delay slot instruction
. . .

; Return from an interrupt

brcr    EIP            ; Resume operation at EIP
brcr    EPC            ; Branch delay slot instruction
```

9.2.3 Interrupt Mechanism

The mechanism for internal or external interrupts is very similar to that for traps. Interrupts are recognized by the execute stage of the MP. Instead of executing the prefetched instruction, the execute unit performs the operations that initiate the interrupt-vector read. The EIP register holds the address of the instruction that was prefetched, which is typically the first instruction executed upon return from an interrupt. The fetch stage is inhibited in the same manner as for traps.

If an instruction contains a 32-bit long-immediate value, then interrupts are recognized only in the fetch long-immediate cycle. At this point, the IP still points to the instruction itself, and the PC points to the 32-bit long-immediate data.

Once an instruction with a long immediate has obtained its data, interrupts are **not** recognized for the actual cycle during which it executes. This is because the PC register is incremented to fetch the next instruction, and there are circumstances (such as branch delay slots) in which saving PC and IP into EPC and EIP would not permit the instruction sequence to resume correctly (see Section 9.3, *Returning From Interrupts and Traps*).

A similar problem occurs if an instruction is not executed because it is in the annulled branch delay slot. In these situations, normal interrupts are not accepted; certain nonmaskable, MP memory fault interrupts must be taken, and a special return sequence may be required (see Section 9.3).

To ensure that an interrupt is not taken when you are modifying the IE register, consider Example 9–2, which enables the floating-point divide-by-zero interrupt (fz). The swcr (swap control register) instruction simultaneously writes 0 to the IE register and saves the previous contents of IE. This disables interrupts and prevents interrupt service routines from modifying the IE register until IE has been updated by the wrcr (write control register) instruction. This assumes that no enabled interrupts are posted for the swcr instruction.

Note that if the swcr instruction is replaced by an rdcr (read control register) instruction, an interrupt could modify the IE register after it was read by the rdcr instruction, but before it was updated the by swcr instruction.

Example 9–2. Enable Floating-Point Divide-by-Zero Interrupt

```
swcr  IE,r0,r5     ; disable all interrupts; set r5 = original IE

or    0x8,r5,r5    ; sets fz bit to 1, assumes ie = 1

wrcr  IE,r5        ; restores new IE with fz interrupt enabled

                   ; any enabled interrupts that are now posted in INTPEN
                   ;  will be taken before the next instruction

addu  0x123,r0,r6 ; interrupt service routines return to this addu
                   ;  instruction
```

Note: This example assumes that no traps or enabled interrupts are posted upon entry.

9.2.4 Nonmaskable Interrupts

Nonmaskable interrupts are always enabled, regardless of the contents of the IE register. Nonmaskable interrupts include floating-point errors, illegal instruction errors, and certain memory faults caused by MP data or instruction accesses. Nonmaskable interrupts are similar to maskable interrupts in that they save the IP and PC of the interrupted program in the EIP and EPC registers.

A nonmaskable interrupt occurring during the servicing of a maskable interrupt is likely to be fatal. This is because the EIP and EPC values saved when the nonmaskable interrupt occurs may overwrite the EIP and EPC values saved earlier by the maskable interrupt, destroying the information necessary to return from the maskable interrupt. For this reason, use caution when writing routines that service maskable interrupts to avoid generating nonmaskable interrupts.

Note that the four emulator traps are nonmaskable but can safely interrupt the service routines for nonemulator interrupts because they do not disturb the EIP and EPC registers. An emulator trap saves the IP and PC values in a separate pair of registers, MIP and MPC.

9.2.5 Synchronizing Interrupts

The signals input on the MVP's four external interrupt pins are, in general, asynchronous to the MVP's internal processor clock. The MP synchronizes these signals to its clock before attempting to process them. This is done by passing each asynchronous signal through a synchronizer latch and applying positive feedback. The output of the synchronizer latch is not sampled until sufficient time has passed to allow any potentially metastable signal level to resolve itself to a valid logic level.

The MVP's four external interrupt pins are fed to the MP. The four external interrupts arrive at the MP on the asynchronous logic. The MP logic synchronizes the signals to its clock by using standard synchronizers.

Notes:

1) The MVP's $\overline{\text{EINT3}}$ pin is multiplexed between the external-interrupt-3 signal and the signal that unhalts the MP.

2) x1, x2, and x3 are edge-triggered interrupts. x4 is a level-sensitive interrupt.

9.3 Returning From Interrupts and Traps

Returning from interrupts and traps is accomplished via the branch control register (brcr) instruction. Bit FLTOP⟦R⟧ (as shown in Figure 3–8, *Memory Fault Operation Register—FLTOP*) affects the interrupt return sequence. This section covers the following cases:

- The normal case (FLTOP⟦R⟧ = 0)
- The alternate case (FLTOP⟦R⟧ = 1)

Note that a memory fault can cause FLTOP⟦R⟧ to be set to 1.

9.3.1 Traps and Normal Interrupt Return (R = 0)

When FLTOP⟦R⟧ = 0, the return sequence uses the brcr instruction to copy a control register (typically the EPC or EIP) to the PC—affecting a branch. At the same time, the LSB of the control register is copied to IE⟦ie⟧; the control register's bit 1 restores the MP to the original user or supervisor mode:

- During a **brcr EIP**, ie is 0—inhibiting interrupts unless the bit is modified by the exception handler.
- During a **brcr EPC**, the value of EPC's LSB is the saved state of the ie bit at the time of the exception; EPC's bit 1 is the original user or supervisor mode value.

To refetch and execute the instruction that EIP points to (as required for interrupts or to retry a faulting instruction), this sequence is used:

```
brcr EIP
brcr EPC
```

To skip over a trap instruction at EIP or if the MP is in the sequential mode for floating-point exceptions, this sequence is used:

```
brcr EPC
nop
```

Here, the nop fills the normal delay slot of the brcr.

9.3.2 Interrupt Return (R = 1)

To refetch and execute the instruction after a long-immediate or branch-delay slot when FLTOP⟦R⟧ = 1, the following sequence restores the original interrupt state and user/supervisor mode:

```
rdcr  EIP,r6    ; r6 = EIP
rdcr  EPC,r7    ; r7 = EPC
addu  4,r6,r6   ; r6 = EIP + 4
sl.im 0,2,r7,r6 ; r6 = ( [EIP+4] & ~3 ) OR ( EPC & 3 )
wrcr  EPC,r6    ; new EPC = r6  MP must be in the
                ;               supervisor mode to
                ;                wrcr.
brcr  EIP
brcr  EPC
```

The case R = 1 can occur only when a memory fault has occurred during execution of an instruction in one of the following sets of circumstances:

- An instruction with a long immediate
- A branch delay slot instruction following a branch instruction

9.4 Power-Up Halt

The power-up halt option is controlled by the state of the $\overline{\text{HREQ}}$ pin on the rising (going inactive) edge of $\overline{\text{RESET}}$:

- If $\overline{\text{HREQ}}$ is **low** (active), the MP comes up running and begins the reset sequence.
- If $\overline{\text{HREQ}}$ is **high**, the MP is halted, waiting for a signal on the $\overline{\text{EINT3}}$ pin to begin the MP's reset sequence.

To accommodate a system in which the MP begins executing immediately upon power-up, there is a zero hold time on $\overline{\text{HREQ}}$ low after $\overline{\text{RESET}}$ is no longer low. This allows you to enable the power-up running option by strapping $\overline{\text{HREQ}}$ to $\overline{\text{RESET}}$.

9.5 Reset

Since the MP's interrupt vectors are stored in its on-chip parameter RAM, the reset mechanism is distinct from interrupts. Since the on-chip RAM is mapped to low addresses, a system with external ROM and RAM should usually keep external RAM areas contiguous; any ROM in the system is likely to reside at high addresses. In order to minimize the constraints on the arrangement of memory, the reset mechanism makes use of the very end of the address space.

Upon reset, the PC is loaded with the byte address 0xFFFF FFF8 (in other words, two 32-bit words below the top of memory). This allows these two words to contain an annulled branch or jump instruction with a 32-bit immediate value. As a result, a reset can vector to any location in the address space. The branch (or jump) should be annulled; otherwise, the instruction at address 0x0000 0000 (in the on-chip PP0 data RAM) is executed in the branch's delay slot.

Note:

To support test, diagnostic, and emulation software, the MVP permits the execution of instructions from the on-chip memory. The MP issues an instruction-cache fill request to the TC, and the request is serviced in a manner similar to an internal-to-internal packet transfer. However, applications software should avoid executing instructions from the on-chip memory because future versions of the MVP may not support this mode of operation.

The reset mechanism described above avoids the use of the data cache and prevents any complications arising from byte ordering associated with big- versus little-endian configurations. Following reset, the TC fetches the subblock containing the instruction at location 0xFFFF FFF8 into the MP's instruction cache.

The state of the machine immediately following power-up reset is as follows:

- ❑ Clear the interrupt enable (IE) register (bit ie = 0 disables all interrupts).
- ❑ Clear the interrupt pending (INTPEN) register (no outstanding interrupts).
- ❑ Reset all cache tag, present (P), and dirty (D) bits.
- ❑ Reset all cache LRU (least recently used) registers to a known state.

- ❑ Clear the CONFIG register, except the endian (E) bit and 16 LSBs.
- ❑ Enable the supervisor mode of the MP operation (user mode bit um = 0; see Section 9.6).
- ❑ Set FPST⟦fm⟧ (the fast mode bit) to 1 to return IEEE denormals and clear the other read/write bits to 0.
- ❑ Clear all pipelines (execution, floating-point unit, memory, TC, VC, PPs).
- ❑ Reset the transfer controller and video controller.
- ❑ Reset the parallel processors (performed by a reset sequence).
- ❑ The PPERROR register is cleared to 0, except that all PPs are halted as indicated by their H bits.
- ❑ The FLTSTS register is cleared to 0.
- ❑ The contents of registers r1–r31 and accumulators a0–a3 are undefined.
- ❑ The contents of the remaining control registers are undefined.
- ❑ The contents of the memory-mapped registers are undefined.
- ❑ The master processor will either:
  - Run—power-up run begins the MP reset sequence—or
  - Halt—power-up halt waits until the $\overline{\text{EINT3}}$ (external interrupt 3) pin is signaled, then begins the MP reset sequence.

9.5.1 Soft Reset Halt

The MP can also be halted as the result of a soft reset if the previous reset was a power-up halt (see Section 9.4). This allows a system with a host (which expects power-up halt) to fully emulate a hard reset. The MP halts when it issues a command word (with the cmnd instruction) to itself with reset (R), halt (h), and MP (M) bits set in the command word and when the power-up halt signal is enabled. (See Section NO TAG, *Interprocessor Communications*, in the *MVP System-Level Synopsis* for more information about the command word.)

There is a one-cycle delay before the MP can begin the halt sequence:

```
                    ; reset, halt, and MP bits = 1
cmnd  0xC0000100    ; issue command instruction
br.a  $             ; spin until MP halts
```

Note: If the contents of data cache must be preserved before the reset command is issued, make sure the data cache is cleaned first (see Section 11.10, *Clean Data Cache Using the dcachec Instruction*).

9.6 Supervisor and User Modes

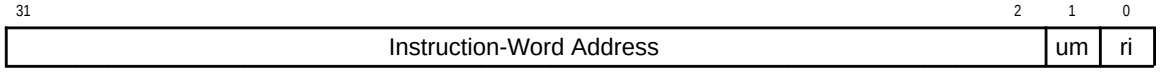
A basic protection mechanism built into the MP provides system software with the ability to regain control from a faulty program. This is accomplished with a user/supervisor mode bit (shown in Figure 9–1) that is loaded by entering a value in bit 1 of the EPC register before executing a brcr EPC instruction. The um bit is loaded into the fetch stage of the pipeline, along with the PC register value. The um bit remains latched there until it is reset by an exception.

- **Supervisor mode** (um = 0)—the software has access to the entire device:
  - MP parameter RAM can be read or written.
  - The MP can branch, read, write, or swap all MP control registers.

  When the MP enters an interrupt service routine, the MP is always in the supervisor mode.

- **User mode** (um = 1)—the software is denied write access to the MP's parameter RAM and all control registers below 0x4000 (see Table 2–1, *MP Control Register Numbers*). This protects the interrupt vectors and their enable bits from interference. In the user mode:
  - MP parameter RAM can be read.
  - MP parameter RAM **cannot** be written; a nonmaskable memory fault (mf) will be generated if a write is attempted.
  - The MP can read all MP control registers.
  - The MP can write to any MP control register $\geq$ 0x4000.
  - The MP cannot branch, write, or swap any MP control register < 0x4000; if attempted, this instruction is executed as a nop instruction, and there is no error signalled to the program.

Figure 9–1. User Mode and Supervisor Mode Designation in the EPC Register



Notes: 1) Instruction-word address is in the top 30 bits of the register.

2) Bit um indicates the user mode (1) or supervisor mode (0).

3) Bit ri is reserved for saving the interrupt enable bit ie when an exception occurs.

4) In the emulation mode, register MPC also has this same format.

The system is placed in supervisor mode upon reset or whenever an exception occurs. In supervisor mode, the software has access to the whole device. User mode is entered when bit 1 (0x0002) is set in the address to be executed in a control register (typically EPC) and then a brcr instruction is executed.

Note:

Programs executing in user mode should not attempt to defeat or circumvent the user/supervisor-mode protection mechanisms. In particular, the result of using the following technique to switch from user mode to supervisor mode is undefined:

Writing a 0 to bit number 1 of a control register that can be written in user mode (IN0P, IN1P, or OUTP) and executing a brcr instruction with that register.

Similar to the PC register, the user mode bit is not directly visible to software (other than what is recorded in EPC[1] after an interrupt or trap). While in the debugger, you can determine if the user mode is enabled by executing a **wrcr FLTTAG,r0** instruction in your program and examining control register FLTTAG to see if its value was changed (supervisor mode) or unchanged (user mode). This assumes that the FLTTAG register was nonzero initially. Note that FLTTAG is an arbitrary choice for a control register and is used here for illustration only.

9.7 Master Processor Interrupt Latency

Many factors influence interrupt latency time. Latency is defined as the time from the activation of an interrupt request to the execution of the first instruction in that interrupt's service routine. The following factors contribute to interrupt latency:

- Crossbar activity
- Transfer controller activities (cache service and packet requests)
- DRAM refresh requests
- Video controller serial register transfer (SRT) requests
- Externally or VC frame timer-initiated packet requests
- Critical nonmaskable interrupts such as MP memory faults
- Number of enabled interrupts that are awaiting service and their relative priorities

9.7.1 Interrupt Latency Estimates

Table 9–3 lists the minimum and maximum bounds for interrupt latency (in clock cycles). The table entries are defined as follows:

- The values given for **minimum interrupt latency** assume that the interrupt service routine is resident in instruction cache.
- The **load interrupt service routine latency** occurs when the interrupt service routine is not resident in instruction cache and requires a cache-miss service to load the initial interrupt service routine instructions before service can begin.
- The **maximum interrupt latency** is the worst-case estimate for the interrupt latency. This includes urgent DRAM refresh, video controller SRT requests, MP instruction cache in progress, MP data-cache block write-back request, MP instruction cache for interrupt service routine, and emptying of the floating-point unit pipeline.

Table 9–3. Master Processor Interrupt Latency Estimates (in Clock Cycles)

	DRAM		
Latency	**1-Cycle**	**2-Cycle**	**3-Cycle**
Minimum latency estimate	8	8	8
Load interrupt service routine minimum estimate	29	37	45
Maximum latency estimate	170	226	282

9.7.2 Latency Assumptions

The following assumptions are presented for the maximum latency estimates shown in Table 9–3.

- The interrupt remains enabled from the point at which the interrupt is requested to the point at which the interrupt is taken.
- There is no crossbar contention (see NO TAG, *Crossbar Priority*, in the *MVP Transfer Controller User's Guide*).
- There is no bus master request from an external host.
- The VC frame timer 0 may require SRT data strobes.
- The VC frame timer 1 may require SRT data strobes.
- An urgent DRAM refresh may be required.
- Externally initiated packet transfers from external hosts, external devices, or VC frame timer-initiated packet requests are considered to be inactive for this analysis. If any of them are active, they could increase the latency because of their high priority (see NO TAG, *Transfer Controller's Request Prioritization*, in the *MVP Transfer Controller User's Guide*).
- MP instruction-cache service may be in progress for the program that will be interrupted.
- MP data-cache service may be required to write back all modified subblocks within a block so that a new subblock replacement can be loaded.
- MP instruction-cache service for the interrupt service routine may be required.
- Enabled floating-point exceptions cause the FEA pipeline to stall until the floating-point unit is empty. Floating-point exceptions can occur in this process which could require higher-priority enabled interrupt service routines; the assumption here is that these additional exceptions do not occur.
- MP or PP packet requests are considered to be at a lower priority and thus will wait in the background and not affect the MP cache service and other urgent TC events (DRAM refresh or VC SRT strobes).
- PP instruction-cache service is at a lower priority than MP cache service and should not affect the TC activity for MP cache requests.
- The external memory speed affects the timing (DRAM with 1, 2, or 3 cycles per column).

9.8 Examples of Interrupt Timing

This section provides some sample programs showing interrupts being activated in the program flow. In these examples, it is assumed that no instruction-cache misses or interrupt service routines are taken, the floating-point unit is in the pipeline mode, and all interrupts are enabled.

9.8.1 Integer Overflow and Resetting IE Register Simultaneously

This example shows the order of execution when an enabled integer overflow occurs at the same time the program is setting register IE to 0 to disable all interrupts.

```
                          ; assume IE = –1, INTPEN = 0
   addu  0x80000000,r0,r5  ; r5 = 0x80000000
   add   r5,r5,r6          ; r6 overflows; sets INTPEN[io] = 1

; integer overflow interrupt is serviced before the next instruction is
;  executed

   swcr  IE,r0,r4          ; executed after the io interrupt service
                           ;   routine returns, new IE = 0, r4 = old IE
```

9.8.2 MP Command Instruction Issues MP Self Interrupt Message

The MP can use a cmnd instruction to issue a message to itself. This is signaled as the MP self interrupt. Note the 1-clock-cycle delay before the mi (MP message interrupt) interrupt is signaled.

```
                          ; assume IE = –1, INTPEN = 0
   addu  0x2100,r0,r5      ; r5 = Message to MP
   cmnd  r5                ; issue command; sets INTPEN[mi] = 1
   subu  r6,r7,r8          ; delay slot of cmnd instruction

; MP message interrupt is serviced before the next instruction is
;  executed

   xor   r6,r7,r4          ; executed after the mi interrupt service
                           ;  routine returns, r4 = r6 XOR r7
```

9.8.3 Integer Overflow and Trap Simultaneously

This example shows the order of execution when an enabled integer overflow occurs at the same time the program issues a trap instruction.

```
                          ; assume IE = –1, INTPEN = 0
   addu  0x80000000,r0,r5  ; r5 = 0x80000000
   add   r5,r5,r6          ; r6 overflows; sets INTPEN[io] = 1

; integer overflow interrupt is serviced before the next instruction is
;  executed

   trap  73                ; executed after the io interrupt service
                           ;  routine returns, then call trap 73 service
                           ;  routine

; trap 73 is serviced before the next instruction is executed

   swcr  SYSSTK,r1,r1      ; executed after the trap 73 interrupt service
                           ;  routine returns, new SYSSTK = original r1,
                           ;  and new r1 = original SYSSTK
```

9.8.4 Integer Overflow is Set in a Branch Delay Slot

This example shows the order of execution when an enabled integer overflow occurs as the result of the program executing a branch delay slot instruction causing the overflow.

```
                                ; assume IE = –1, INTPEN = 0
         addu  0x80000000,r0,r5  ; r5 = 0x80000000
         br    PathC             ; unconditional branch
         add   r5,r5,r6          ; r6 overflows; sets INTPEN[io] = 1 in
                                 ;  the branch delay slot
. . .

; integer overflow interrupt is serviced before the first instruction at
;  PathC is executed

PathC:   wrcr  OUTP,r9           ; executed after the io interrupt
                                 ;  service routine returns, OUTP = r9
```

9.8.5 Integer Overflow and Annulled Branch Simultaneously

This example shows the order of execution when an enabled integer overflow occurs at the same time the program issues an annulled branch instruction.

```
                          ; assume IE = –1, INTPEN = 0
   addu  0x80000000,r0,r5  ; r5 = 0x80000000
   add   r5,r5,r6          ; r6 overflows; sets INTPEN[io] = 1

; integer overflow interrupt is serviced before the next instruction is
;  executed

   br.a  PathC             ; executed after the io interrupt service
                           ;  routine returns, program now branches to
                           ;  PathC
. . .
PathC: . . .
```

9.8.6 Floating-Point Divide-by-Zero Interrupt

This example shows the order of execution when an enabled floating-point divide-by-zero input exception occurs.

```
                          ; assume IE = –1, INTPEN = 0
   addu     0,r0,r5       ; r5 = 0
   fdiv.sss 3.75,r5,r4    ; r4 = 3.75/0 sets INTPEN[fz] = 1

; Floating-point divide-by-zero interrupt is serviced before the next
;  instruction is executed. Note that r4 is scoreboarded.

   fadd.sss 4.56,r4,r6    ; executed after the fz interrupt service
                          ;  routine returns, r6 = 4.56 + r4

   fdiv.sss 3.75,r5,r4    ; r4 = 3.75/0 sets INTPEN[fz] = 1
   addu     0x11,r0,r11
   addu     0x12,r0,12

; Floating-point divide-by-zero interrupt is serviced before the next
;  instruction is executed.

   addu     0x13,r0,r13   ; executed after the fz interrupt service
                          ; routine returns
```

9.8.7 Floating-Point Overflow Interrupt

This example shows the order of execution when an enabled floating-point overflow output exception occurs (square 10^{20}).

```
                              ; assume IE = –1, INTPEN = 0
   addu     0x60ad78ec,r0,r5  ; r5 = 1.0 x e+20
   fmpy.sss r5,r5,r4          ; r4 overflows, sets INTPEN[fo] = 1 and
                              ;  INTPEN[fx] is also set
   addu     0x11,r0,r11       ; 1st instruction after fmpy
   addu     0x12,r0,r12       ; 2nd instruction after fmpy

; Floating-point overflow and inexact interrupts are serviced before the
;  next instruction is executed

   fadd.sss 4.56,r5,r6        ; executed after the fo/fx interrupt
                              ;  service routines return, r6 = 4.56 + r5
```

9.8.8 Floating-Point Underflow Interrupt

This example shows the order of execution when an enabled floating-point underflow output exception occurs (square 10^{-20}).

```
                              ; assume IE = –1, INTPEN = 0
   addu     0x1e3ce508,r0,r5  ; r5 = 1.0 x e–20
   fmpy.sss r5,r5,r4          ; r4 underflows, sets INTPEN[fu] = 1 and
                              ;  INTPEN[fx] is also set
   addu     0x11,r0,r11       ; 1st instruction after fmpy
   addu     0x12,r0,r12       ; 2nd instruction after fmpy

; Floating-point underflow and inexact interrupts are serviced before the
;  next instruction is executed

   fadd.sss 4.56,r5,r6        ; executed after the fu/fx interrupt
                              ; service routines return, r6 = 4.56 + r5
```

9.8.9 Enable All Interrupts and Clear INTPEN Registers

This example demonstrates how to clear all pending interrupts and enable all interrupts.

```
                        ; assume IE = –1, INTPEN = 0
   wrcr  IE,r0          ; disables all interrupts
   addu  –1,r0,r2       ; r2 = –1
   wrcr  INTPEN,r2      ; turn off all INTPEN 1 bits
   wrcr  IE,r2          ; IE = –1, enables all interrupts
```

9.8.10 Disable All Interrupts

This example demonstrates how to disable all interrupts. See subsection 9.8.1 to see what happens when INTPEN is nonzero.

```
                        ; assume IE = –1, INTPEN=0
   addu  0x80000000,r0,r5  ; r5= 0x80000000
   wrcr  IE,r0             ; disables all interrupts

; This writes all zeros into register IE and disables all interrupts
; before the next instruction

   add   r5,r5,r6          ; sets INTPEN[io] = 1 but does not take an io
                           ;  interrupt
```

Note: If there are any enabled interrupts that are also posted in the INTPEN register, the interrupt service routine for those interrupts are performed before the **wrcr IE,r0** instruction is executed in this example.

9.9 Polling Bit Values

This section provides some sample programs that show how polling bit values can be used.

9.9.1 MP Determines When PP0 Has Halted

This example shows the order of execution when the MP issues a reset command (with the cmnd instruction) to PP0. The MP waits for the PP0 reset sequence to complete, which is indicated when PP0's halt bit is set in the PPERROR register. Note that halting a PP does **not** set INTPEN[pe] to 1.

```
                              ; assume IE = –1, INTPEN = 0,
                              ;  PPERROR = 0
        addu  0x80000001,r0,r5  ; r5 = 0x80000001, reset PP0
        cmnd  r5                ; issue cmnd word to PP0
        addu  0,r0,r6           ; set a counter in r6 = 0

; Allow at least 2 cycles between the MP's cmnd instruction and the first
; attempt to poll PPERROR's halt to bit to see when PP0 has halted.
;
; Poll PPERROR halt bit for a value of 1 which indicates PP0 is halted

Loop2:  rdcr  PPERROR,r7        ; r7 = PPERROR register
        bbz   Loop2,r7,16       ; test PPERROR PP0 halt bit = 1,
        addu  1,r6,r6           ; increment r6 counter
```

9.9.2 MP Waits for PP0 to Halt Itself

This example shows the order of execution when the MP issues an unhalt command (with the cmnd instruction) to PP0. The MP waits for the PP0 program to issue a halt command (using the cmnd = instruction) to itself. The PP0 halt is indicated when PP0's halt bit is set in the PPERROR register. Note that halting a PP does **not** set INTPEN[pe] to 1.

```
         .global  PP0_TASK
                                  ; assume IE = –1, INTPEN = 0,
                                  ;  PPERROR = 0
         addu     PP0_TASK,r0,r4  ; PP0's task program
         st       0x010001b8(r0),r4 ; store PP0's task interrupt vector
         addu     0x20004001,r0,r5  ; r5 = 0x20004001, unhalt PP0 with a
                                  ;  task
         cmnd     r5              ; issue cmnd word to PP0
         addu     0,r0,r6         ; set a counter in r6 = 0

; Poll PPERROR halt bit for a value of 1 which indicates PP0 is halted

Loop3:   rdcr     PPERROR,r7      ; r7 = PPERROR register
         bbz      Loop3,r7,16     ; test PPERROR PP0 halt bit = 1,
         addu     1,r6,r6         ; increment r6 counter
```

Portions of the PP program are shown next:

```
         .global  PP0_TASK

PP0_TASK . . .
         d6 = 0x40000001         ; halt = 1, PP0 = 1 in command word
         cmnd = d6               ; issue the halt PP0 command
         . . .
```

9.9.3 MP Waits for a Message From PP0

This example shows the order of execution when the MP issues an unhalt command (with the cmnd instruction) to PP0 and waits for PP0 to send a message to MP. A message to the MP from PP0 is posted when INTPEN⟦p0⟧ = 1. The software can poll INTPEN, or a PP0-message-to-MP interrupt service routine can be serviced when IE⟦ie⟧ = IE⟦p0⟧ = 1. The following example shows the polling approach.

```
         .global  PP0_TASK
                                ; assume IE = –1, INTPEN = 0,
                                ;  PPERROR = 0
         addu  PP0_TASK,r0,r4   ; PP0's task program
         st    0x010001b8(r0),r4 ; store PP0's task interrupt vector
         addu  0x20004001,r0,r5  ; r5 = 0x20004001, unhalt PP0 with a
                                ;  task
         cmnd  r5               ; issue "cmnd" word to PP0
         addu  0,r0,r6          ; set a counter in r6 = 0

; Poll INTPEN PP0 message bit = 1 which indicates PP0 has sent message to
;  MP

Loop5:   rdcr  INTPEN,r7        ; r7 = INTPEN register
         bbz   Loop5,r7,16      ; 0 = spin until PP0 sends message to MP
         addu  1,r6,r6          ; increment r6 counter

PP0MSG:  ld    PP0Box(r0),r9    ; PP0 message to MP has arrived
         cmp   r9,r8,r7         ; program continues after PP0 message
```

Portions of the PP program are shown next:

```
         .global  PP0_TASK

PP0_TASK . . .
         d7 = 0x2100              ; message to MP
         cmnd = d7                ; issue the PP0 message to the MP
         . . .
```

9.10 Floating-Point Operations During Interrupt Servicing

A service routine for a nonfloating-point interrupt may need to perform floating-point operations. The general rule is that the routine can do so, as long as it comprehends the effect of the interrupt environment on the execution of floating-point instructions.

Note that an unwary C programmer may use the floating-point unit without realizing it. For instance, if an interrupt routine accesses an array element, the C compiler may generate an integer multiply instruction, which is performed by the floating-point unit, as a means of calculating an index into the array.

Floating-point operations are enabled when both IE[[ie]] = 1 (the global interrupt enable bit is set) and FPST[[fs]] = 0 (the floating-point stall bit is not set). The machine interprets an attempt to execute a floating-point instruction while the floating-point pipeline is stalled (that is, while either ie = 0 or fs = 1) as a floating-point error and generates a nonmaskable interrupt. This error is likely to be fatal if it occurs during servicing of another interrupt.

In general, the best method for clearing the fs bit is to make this the responsibility of the routines that service floating-point interrupts. Any time a nonfloating-point interrupt routine begins executing, it can always assume that fs = 0 without having to verify this fact.

Interrupts generated by floating-point exceptions have higher priorities than interrupts from other sources. If a floating-point interrupt becomes pending at the same time as a nonfloating-point interrupt, the floating-point interrupt is taken first. This prioritization guarantees that the floating-point interrupt routine always has the opportunity to clear the fs bit before the service routine for the nonfloating-point interrupt has a chance to run.

When the MP takes an interrupt, the MP automatically clears the ie bit, which disables interrupts globally and also stalls the floating-point unit. Assuming that the fs bit is already 0, an interrupt routine can enable floating-point operations by setting the IE register to 0x0000 0001. This both sets the ie bit and individually disables each of the MP's maskable interrupts.

An interrupt routine that sets IE = 0x0000 0001 in order to enable the floating-point pipeline also disables all floating-point exceptions. When writing an interrupt service routine, you should carefully consider the effects of errors in the floating-point calculations performed by that routine.

Chapter 10

The MP Assembly Language Instruction Set

This chapter presents the MP instruction formats and describes the MP instruction set.

Topics

10.1 Instruction Formats

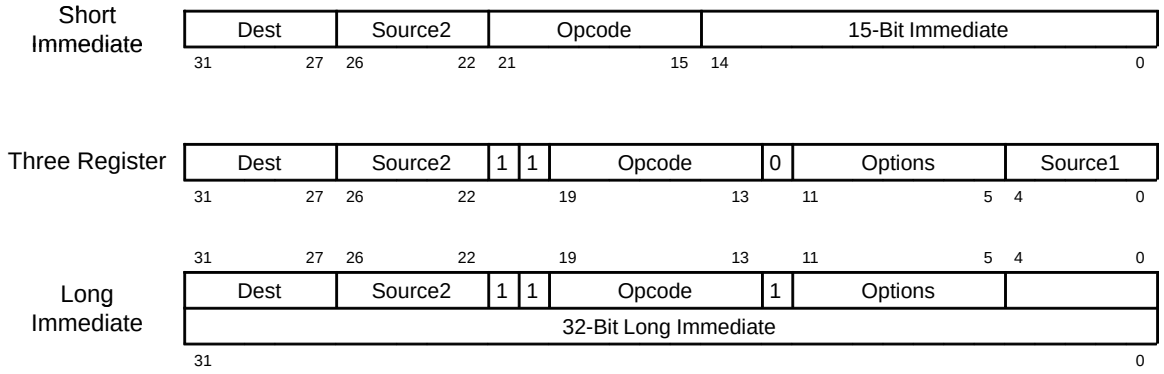
The MP has three basic instruction formats (shown in Figure 10–1):

- **Short immediate.** This format allows you to specify a 15-bit immediate operand, one source register, and a destination register. The 15-bit immediate value is considered to be signed for some instructions and unsigned for others.
- **Three register.** This format allows you to specify two source registers and one destination register. The remaining bits (labeled *options*) are either ignored or specified to the instruction.
- **Long immediate.** This format allows you to specify all of the parts of the three-register form, plus a 32-bit immediate constant.

If the same operation can be specified in all three formats (for example, the add instruction), the 7-bit opcode portion of the instruction is the same in all three formats. However, because the three-register and long-immediate formats can decode extra options, these formats have a binary 11 prefix in the opcode field.

Table 10–8 to Table 10–10 summarize the instruction set encoding. Table 10–7 provides a key to abbreviations used in these tables.

Figure 10–1. Instruction Formats



10.2 Arithmetic, Logical, and Compare Instructions

The master processor supports a full range of arithmetic, logical, and compare instructions. These instructions are divided into the following categories:

- Integer add and subtract instructions
- Logical instructions
- Compare instructions

10.2.1 Integer Add and Subtract Instructions

Integer addition and subtraction are performed by the integer ALU, and these instructions can be signed or unsigned operations. The add and subtract instructions are:

Operation	Instruction	See page
Signed add	add	MP: 10-49
Unsigned add	addu	MP: 10-50
Signed subtract	sub	MP: 10-151
Unsigned subtract	subu	MP: 10-152

Note:

The unsigned addu and subu instructions do not cause an integer-overflow signal; however, the signed add and sub instructions can cause an integer-overflow signal.

10.2.2 Logical Instructions

The integer ALU performs the standard Boolean operations, which include bitwise-OR (**or**), bitwise-AND (**and**), bitwise exclusive-NOR (**xnor**), and bitwise exclusive-OR (**xor**). The **and** and **or** instructions permit each source operand (source1 or source2) to be either inverted or used in its true form. The Boolean (logical) instructions are:

Operation	Instruction	See page
Bitwise-AND	❑ and	MP: 10-51
	❑ and.ff	MP: 10-52
	❑ and.ft	MP: 10-53
	❑ and.tf	MP: 10-54
Bitwise-OR	❑ or	MP: 10-122
	❑ or.ff	MP: 10-123
	❑ or.ft	MP: 10-124
	❑ or.tf	MP: 10-125
Bitwise exclusive-NOR	xnor	MP: 10-181
Bitwise exclusive-OR	xor	MP: 10-182

10.2.3 Compare Instructions

Comparisons are performed explicitly by using the **cmp** instruction for integer compares and the **fcmp** instruction for floating-point compares. The results of the compare are stored in the destination register. A typical program can test for a specific condition by using the branch bit one (bbo) instruction or the branch bit zero (bbz) instruction following a compare (see subsection 10.4.5, *Branching Conditionally: Branch on Bit*). The two types of compares are:

Operation	Instruction	See page
Integer compare	cmp	MP:10-72
Floating-point compare	fcmp	MP:10-93

Note:

The integer-overflow exception (INTPEN[[io]]) **cannot** occur if the integer compare (cmp) instruction overflows.

10.3 Floating-Point and Vector Instructions

The floating-point unit is capable of performing IEEE-754 floating-point operations in 32-bit single-precision and 64-bit double-precision floating point. Conversion between different formats is also supported. In addition, the floating-point unit provides vector floating-point operations, with the option to perform a parallel load or store data (vld or vst, respectively) to improve program efficiency.

If the *dest* field is encoded in vector instructions as 0 (= r0) and the destination precision is encoded as 1 (double-precision floating point), the answer is written to the accumulator that is encoded by the *a* (accumulator) bits. Otherwise, the answer is written to r0 (or discarded). This is true for the vmac, vmsc, vmsub, and vrnd instructions with floating-point input. This allows you to examine the opcode fields generated by the MP assembler (mpasm).

Notes:

1) Interrupts must be enabled for any floating-point or vector operations (IE⟦ie⟧ = 1).

2) Floating-point exceptions are discussed in Section 8.4, *Floating-Point Unit Exceptions*.

3) The floating-point multiply unit also performs integer multiplication (see the fmpy instruction on page MP:10-97).

4) Double-precision floating-point registers must be specified as even register numbers. For additional information about the use of r0, see subsection 10.9.2.

5) Register r1 cannot be a single-precision floating-point destination register.

You can select the type of rounding on each floating-point operation by setting the default rounding mode field (drm) in the floating-point status (FPST) register, as shown in Table 10–1.

Table 10–1. Floating-Point Default Rounding Modes

drm	Rounding Method for Floating-Point Numbers	FPST Bit 18	FPST Bit 17
0	Rounds toward nearest (default at reset)	0	0
1	Rounds toward zero	0	1
2	Rounds toward positive infinity	1	0
3	Rounds toward negative infinity	1	1

10.3.1 Floating-Point Arithmetic Instructions

The standard arithmetic operations are provided for floating-point operands:

Operation	Instruction	See page
Floating-point add	fadd	MP: 10-91
Floating-point subtract	fsub	MP: 10-105
Integer or floating-point multiply	fmpy	MP: 10-97
Floating-point divide	fdiv	MP: 10-96
Floating-point square root	fsqrt	MP: 10-104
Floating-point compare	fcmp	MP: 10-93

10.3.2 Floating-Point Conversion Instructions

The conversion between various formats is supported by the floating-point add unit:

Operation	Instruction	See page
Round toward **nearest** number	frndn	MP: 10-101
Round toward **zero** and truncate the mantissa	frndz	MP: 10-103
Round toward **positive infinity**	frndp	MP: 10-102
Round toward **minus infinity**	frndm	MP: 10-99

10.3.3 Vector Floating-Point Arithmetic Instructions

The vector instructions allow a load or store to occur in parallel with a floating-point operation. The main feature of some of the vector instructions (as far as the floating-point unit is concerned) is that the floating-point add unit pipeline is added to the end of the floating-point multiply unit pipeline. Note that vmac, vmsc, vmsub, vrnd.sd, and vrnd.dd are the only instructions that can have an accumulator destination.

The vmac-type instructions can both write to an accumulator and have a parallel load or store coded in them. Note that an accumulator destination is mandatory when a parallel load or store is used, regardless of the register location coded in the opcode (single-precision accumulator destinations are **not** legal). When the result of a vector instruction is written to an accumulator, r0 is written to the FPST register as the destination register.

The available vector arithmetic instructions are:

Operation	Instruction	See page
Vector floating-point add	vadd	MP: 10-157
Vector floating-point subtract	vsub	MP: 10-177
Vector floating-point multiply	vmpy	MP: 10-164
Vector floating-point round	vrnd	MP: 10-171, MP: 10-173

Note:

Vector divide and vector square root are not supported.

See Section 10.7, *Memory-Access Instructions* for information concerning vector load or vector store (vld or vst, respectively) of 32-bit words or 64-bit doublewords.

10.3.4 Vector Floating-Point Multiply and Add/Subtract Instructions

Because you can pipeline products directly from the floating-point multiply unit into the floating-point add unit, it is much easier to perform convolutions, dot products, and FFT butterflies. The operations supported are:

Operation	Instruction	See page
Vector floating-point multiply and add to accumulator	vmac	MP:10-161
Vector floating-point multiply and subtract from accumulator	vmsc	MP:10-166
Vector floating-point subtract accumulator from source	vmsub	MP:10-169

When using these operations, you must remember the following:

- The two source operands to the floating-point multiply unit (source1, source2) must be single-precision floating-point values.
- The source1 × source2 product that the floating-point multiply unit produces is always a double-precision floating-point value and becomes one of the inputs to the floating-point add unit (see Figure 4–1, *General Floating-Point Unit Flow*).
- The other input to the floating-point add unit is either a zero or a double-precision floating-point value from one of the four double-precision floating-point accumulators.
- The output sum (or difference) from the floating-point add unit is initially a double-precision floating-point value. It remains a double-precision floating-point number if the destination is an accumulator (for example, **vmac.ssd** *source1, source2, acc, acc*). However, the double-precision floating-point result can be written to an even register (other than r0); this practice is still IEEE-compatible.

  If the destination format is single-precision, then the dest value must be a register (**vmac.sss** *source1, source2, acc, dest*).
- For vmac.sss instructions, if the single-precision result is denormal before rounding, the result is flushed to 0.
- For vmac.sss instructions, round-to-zero (truncate) is forced, independent of the drm value in the FPST register.

The vmac-type instructions (vmac, vmsc, and vmsub) also allow you to include parallel load or store operations. For more information concerning vector load or vector store (vld or vst, respectively) of 32-bit words or 64-bit doublewords, see Section 10.7, *Memory-Access Instructions*.

The IEEE-754 standard does not specify what a multiply-and-add precision should be. The following product is used in the multiply phase of a vmac.ssd or vmac.sss instruction-type and complies with the standard:

single precision	×	**single precision**	→	**vmac double-precision product**

Likewise, the following sum used in the add phase of the vmac.ssd instruction-type is also in compliance:

accumulated double-precision old sum	+	**vmac.ssd double-precision product**	→	**double-precision sum**

The following sum, used in the add phase of the vmac.sss-type instruction, in and of itself, is not in compliance:

accumulated double-precision old sum	+	**vmac.sss double-precision product**	→	**single-precision sum**

As a result, vmac.sss does not comply with the IEEE-754 standard. In order to produce results that comply to the IEEE-754 standard, use vmac.ssd to write to a register, and then use frnd*.ds to round/convert from double precision to single precision (* is n, z, p, or m for the rounding mode of nearest, zero, plus infinity, or minus infinity, respectively.)

10.3.5 Vector Floating-Point Conversion Instructions

The floating-point add unit also supports format conversions with an optional vector load or store instruction by using default rounding mode (drm) bits in the FPST register. This makes it possible to round toward:

- Nearest—round toward nearest number
- Zero—truncates mantissa in rounding towards zero
- $+\infty$—rounds toward positive infinity
- $-\infty$—rounds toward minus infinity

Information concerning vector load or vector store (vld or vst, respectively) of 32-bit words or 64-bit doublewords is found in Section 10.7, *Memory-Access Instructions*.

For a complete definition of rounding, consult the IEEE-754 standard.

Operation	Instruction	See page
Vector round with floating-point input	vrnd.sx, vrnd.dx	MP: 10-171
Vector round with integer input	vrnd.ix, vrnd.ux	MP: 10-173

10.3.6 Constraints Associated With Using the Accumulators

Refer to subsection 8.3.7, *Denormals in the Multiplier*, for an explanation of how to use denormal numbers as inputs to the three vmac-type instructions (vmac, vmsc, and vmsub).

Note that the values stored in the accumulators are **not** altered because of a hardware or software reset. After a reset has occurred, be sure to use an input accumulator value of zero the first time an accumulator is used with a vmac-type instruction; otherwise, unpredictable results may occur. See Example 10–1.

Example 10–1. Set/Use Zero Accumulator Values

```
vrnd.sd  r0,a1       ; sets a1 = 0
vmac.ssd r2,r3,0,a1  ; uses input sum = 0 in
                     ; r2×r3 + 0 → a1
```

When you use a vmac-type instruction with an accumulator destination, follow these guidelines:

- **Input operands must be single-precision for vmac, vmsc, and vmsub instructions.** This prevents the generation of output exceptions in the floating-point multiply unit during the multiply portion of the instruction.

  The multiply that is performed during the execution of these instructions is an fmpy.ssd. When two single-precision numbers are multiplied together with a double-precision result, the product never overflows or underflows, and the result is always exact.

  - **For accumulator destinations**, the operation in the add portion of these instructions is always an fadd.ddd or an fsub.ddd. One input operand to this add/subtract is the product of the fmpy.ssd operation. The other accumulator input operand is the result of a similar add/subtract. The result of the add/subtract in the add portion of these vector operations should not overflow or underflow. This helps eliminate input invalid exceptions to the floating-point add unit during these instructions.
  - **For register destinations**, the fadd.dds and fsub.dds instructions are allowed, along with the fadd.ddd and fsub.ddd instructions in the floating-point add portion of a vmac-type of instruction (vmac, vmsc, vmsub). Overflow and underflow can occur with fadd.dds and fsub.dds. When the destination precision is single-precision, the result should have a register file destination other than r0 or r1; for double precision other than r0, see subsection 10.9.2.

    Although fadd.dds and fsub.dds are not a valid IEEE-754 mix of precisions, they are allowed in the floating-point add portion of vmac-type instructions. The definitive list of allowed precision combinations is provided in the description of each instruction. The fadd.dds or fsub.dds part of the vmac.sss instruction can result in a denormal result. For more details about this occurrence, see Section C.2, *Vector Operation Exceptions*.

- **For the vmac, vmsc, and vmsub instructions, the input operands should not be ± ∞.** The use of ± ∞ is allowed for these instructions, but an invalid exception may occur at the input of the floating-point add unit. The use of ± ∞ is allowed for these instructions; however:

  - If ∞ is an input operand to the floating-point multiply unit, ∞ can be generated for the input to the floating-point add unit.
  - If ∞ is an input to the floating-point add unit, then ∞ can be stored in one of the accumulators.
  - If ∞ is in an accumulator, then an invalid exception can be generated at the input of the floating-point add unit during vmac-type instructions (vmac, vmsc, and vmsub). The only invalid operation that can occur at the floating-point add unit during vmac-type instructions is the magnitude subtraction of infinities.
  - If ∞ is an input operand to one of these instructions, and an invalid operation occurs at the floating-point add unit, the exception is signaled when the result of the operation is written to an accumulator or the register file, and the invalid interrupt is taken at this time if the invalid interrupt is enabled. A TI QNaN (quiet not-a-number) is the result. The result is written to an accumulator for accumulator destinations. Otherwise, the result is written to the register file.

    If you are executing these instructions in the sequential mode with input interrupts enabled, the register write is affected; see subsection 8.5.2, *Floating-Point Sequential Mode Versus Pipelined Mode*.

  For vmac-type instructions, an invalid operation can occur at the input of the floating-point multiply unit at the same time that an invalid operation occurs at the input of the floating-point add unit. However, since invalid operations are piped to the output of the appropriate arithmetic unit (floating-point multiply or floating-point add unit, as described in Section 8.4, *Floating-Point Unit Exceptions*), the two invalid operations will **not** be signaled at the same time.

  Also, if an invalid operation occurs at the input to the floating-point add unit during a vmac-type instruction and an invalid interrupt is taken as a result, the interrupt handler does not have access to the input operands that caused the invalid operation, because one of the inputs is from the accumulator. However, since only ∞ can cause an invalid operation in this

situation, the interrupt handler could deduce the operand values.

- **The vrnd.dd instruction with an accumulator destination is intended to be used only for restoring the state of an accumulator.** Therefore, if you use the vrnd.dd instruction with an accumulator destination that did not originate from an accumulator, you **should** store either a double-precision zero or a number within the double-precision range:

  $\pm(2^{-253} < \text{number} < 2^{+253})$

  If you store a number into an accumulator that did not originate from an accumulator or is outside the given range, excessive invalid, overflow, or underflow exceptions may occur. Use the vrnd.sd instruction to load numbers into the accumulators that are within the specified range.

The vmac-type instructions are implemented to significantly increase the throughput of consecutive multiply accumulates. In order to ensure complete IEEE-754 standard compatibility without any input constraints for multiply accumulate, you should multiply with a double-precision register destination and then add/subtract with a double-precision register destination.

10.4 Program-Control and Context-Switching Instructions

The MP supports a variety of instructions that allow you to control program flow:

- Unconditionally transfer program control to new locations (branch unconditionally)
- Transfer program control to new locations according to a specific condition on a register (branch conditionally, compare to zero)
- Transfer program control to new locations according to the value of a bit in a register (branch conditionally, branch on bit)
- Call functions and subroutines and return

This section summarizes these types of program-control instructions. The section begins with a discussion of branch delay slots and branches with long immediates.

10.4.1 Branch Delay Slots

The first instruction following a branch instruction is called the branch delay slot. When developing your program, there are some precautions concerning branch delay slots that you should take.

For all nonannulled branches, you should not use another branch instruction in the branch delay slot because of the following concerns:

- The order of the instruction execution and the difficulties that can be introduced should an interrupt occur in the middle of the two branches,
- The delay-slot implications of the two branch instructions, and
- The recovery/return process in the interrupt service routines.

If you code two consecutive branches, assuming there are no interrupts and that both branches are taken, the MP executes one instruction from the branch location in the initial branch instruction, and program execution continues from the branch location in the next branch instruction. With the introduction of interrupts, the correct return using a standard return sequence is difficult to follow. Consider the following example (with no interrupts taken):

```
         bcnd     First,r0,eq0.w    ; 1st branch is
                                    ;  taken
         bcnd.a   Second,r0,eq0.b   ; 2nd branch is
                                    ;  also taken
         . . .
First:   . . .
         . . .
Second:  . . .
         . . .
```

The safest suggestion is for the program to conform to the rule: You should **not** use branches (or long-immediate operands) in the branch delay slot of nonannulled branches unless the branch instructions are mutually exclusive.

10.4.2 Branches with Long–Immediates

Most relative branches are handled easily with the 15-bit word offset value. However, there are environments where this is somewhat more difficult:

- When the program is sharing global variables from different source files
- When the 15-bit short-immediate offset is not sufficient

The following suggestions offer a solution.

Label **target** should be in this source file and within ± 16 383 32-bit instruction words in order to fit the 15-bit short-immediate format of all branch instructions. If **target** is not in this file or is greater than absolute 16 383 words relative displacement, then the 32-bit long-immediate format is required. Currently, the MP assembler (mpasm) does not support long-immediate branches. However, you can use the following to affect a long-distance absolute branch as long as **target** is declared to be a .global variable.

```
jsr   target(r0),link   ; target → PC, and the
                        ; return address → link
```

Assuming you want an unconditional branch, you can supply **link** as r0 (the discard register). If you want to return, you must save the contents of the original link register before you issue the jsr instruction.

10.4.3 Branching Unconditionally

You can unconditionally transfer program control to new locations by using the branch (br) instruction for programs or the branch control register (brcr) instruction for interrupt and trap service routines. The general form of unconditional branches is as follows:

Operation	Instruction	See page
Relative branch always	br *target*	MP:10-64
Annulled relative branch always	br.a *target*	MP:10-64
Branch control register	brcr *control*	MP:10-66

The first instruction following a nonannulled branch instruction is defined as the **branch delay slot** instruction.

- In br instructions, *target* → PC, and the delay slot instruction is executed before the first instruction at location target.
- In br.a instructions, *target* → PC, and a nop instruction is executed before the first instruction at location *target* → PC.

The brcr instructions are typically used to return from exceptions or traps. You should examine the FLTOP⟦R⟧ bit to see if a modified return sequence is required as the result of a long 32-bit immediate in the exception instruction stream (see Figure 3–8, *Memory Fault Operation Register—FLTOP*, and the application examples in subsection 9.3.2, *Interrupt Return (R=1)*).

In brcr instructions, the following events occur in supervisor mode (*control_register* = the contents of *target*):

- *control_register* & 0xFFFF FFFC → PC
- *control_register*⟦0⟧ → IE⟦ie⟧ (ie = 1 enables interrupts)
- *control_register target*⟦1⟧ → user mode latch (1 = user, 0 = supervisor)

In user mode, if the control register number that you specify with the brcr instruction is less than 0x4000, the brcr instruction is treated as a nop instruction, and you do not receive an error signal.

Note:

For all nonannulled branches, you cannot use a long-immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction).

10.4.4 Branching Conditionally: Compare to Zero

The branch conditional (bcnd) instruction tests the register for the condition specified, using the data size requested. There are two types of bcnd instructions:

Operation	Instruction	See page
Relative branch conditional	bcnd	MP: 10-61
Annulled relative branch conditional	bcnd.a	MP: 10-61

The condition *code* syntax is specified by the form *condition.size* (for example, **lt0.h**). The branch condition uses a 5-bit coding:

- The type of branch *condition* is represented in the lower three bits of assembled *code* field as (<, =, or > zero):

Syntax Prefix	Type of Condition	Bits 29	28	27
nev	Never (another form of nop)	0	0	0
gt0	Greater than zero	0	0	1
eq0	Equals zero	0	1	0
ge0	Greater than or equal to zero	0	1	1
lt0	Less than zero	1	0	0
ne0	Not equal to zero	1	0	1
le0	Less than or equal to zero	1	1	0
alw	Always (another form of br)	1	1	1
		<	0	>

- The data *size* is represented in the top two bits of assembled *code* field as:

Syntax Suffix	Data Size	Bits 31	30
.b	8 LSBs	0	0
.h	16 LSBs	0	1
.w	32-bit word	1	0
.d	Reserved	1	1

The bcnd instruction tests data in register *source* for condition and size:

- If *code* is true—branch to *target*.
- If *code* is false—continue to next instruction.

A branch delay slot instruction is the first instruction following any branch instruction. The bcnd instruction always executes the branch delay slot instruction before any (optional) branch is taken, regardless of the condition.

The bcnd.a instruction executes a nop instruction in the branch delay slot if the condition is true (for example, the branch is taken). If the condition is false, execution continues with the branch delay slot instruction.

Note:

For all nonannulled branches, you cannot use a long-immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction).

10.4.5 Branching Conditionally: Branch on Bit

The **bbz** and **bbo** instructions test the value of a bit in a register. The register is selected by the source field.

Operation	Instruction	See page
Relative branch if bit = 0	bbz	MP:10-58
Relative branch if bit = 1	bbo	MP:10-55

Often, a bbz or bbo instruction follows a cmp (compare) or fcmp (floating-point compare) instruction. The assembler provides mnemonics for all of the common comparisons according to the bits set by the cmp/fcmp instructions.

The *source* register's *bitnum* bit is tested (bbo tests for a 1; bbz tests for a 0).

- If the condition tested is true—branch to *target.*
- If the condition tested is false—continue to next instruction.

A branch delay slot instruction is the first instruction following any branch instruction. In the bbo or bbz instructions, the branch delay slot instruction is executed before any (optional) branch is taken, regardless of the condition.

In annulled branch instructions (bbo.a or bbz.a), a nop is executed in the branch delay slot if the condition is true (for example, the branch is taken). If the condition is false, execution continues with the branch delay slot instruction.

Note:

For all nonannulled branches, you cannot use a long-immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction).

10.4.6 Calling Functions and Subroutines and Returning

You can call functions or subroutines by using the relative branch (**bsr**, shown on page MP:10-68) or absolute jump (**jsr**, shown on page MP:10-111) instructions. Both instructions save the return address in register *link* (which is r31 by software convention); the PC is loaded with a new address *target* (for bsr) or *offset* + *base* (for jsr). The branch delay slot instruction (the first instruction following the bsr or jsr instruction) can be executed or annulled.

Each *branch* instruction (bsr or jsr) has two forms:

- *branch*—The 32-bit instruction following the *branch* instruction (branch delay slot) is executed. The *branch* is always taken.
- *branch***.a**—The instruction following the *branch***.a** is not executed (a nop cycle is executed instead). The branch is always taken. The *branch***.a** instructions take the same time as *branch* instructions but may require one less instruction memory word.

Note:

For all nonannulled branches, you cannot use a long-immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction).

10.5 Control Register Instructions

Table 2–1, *MP Control Register Numbers*, gives the register addresses of the accessible registers for the control register instructions. These instructions are:

Operation	Instruction	See page
Branch control register†	brcr	MP:10-66
Read control register	rdcr	MP:10-126
Swap control register†	swcr	MP:10-153
Write control register†	wrcr	MP:10-179

† Must be in supervisor mode if control register number < 0x4000

Notes:

1) In user mode, if the control register number that you specify with the brcr, swcr, or wrcr instruction is less than 0x4000, the instruction is treated as a nop instruction, and you do not receive an error signal.

2) In either mode, if the control register is an invalid number, the operation is undefined, and you do not receive an error signal.

10.6 Leftmost and Rightmost One Instructions

You can determine the leftmost or rightmost bit number by using these instructions:

Operation	Instruction	See page
Find leftmost one	lmo	MP:10-119
Find rightmost one	rmo	MP:10-127

10.7 Memory-Access Instructions

Data in memory (external or on-chip) can be referenced by loads and stores with various data sizes. The memory-access instructions are:

Operation	Instruction	See page
Signed load	ld	MP:10-113
Signed DEA load	dld	MP:10-76
Unsigned load	ld.u	MP:10-116
Unsigned DEA load	dld.u	MP:10-79
Vector load	vld	MP:10-159
Store	st	MP:10-148
DEA store	dst	MP:10-82
Vector store	vst	MP:10-175

Operations for loads and stores, as shown in Table 10–2, are summarized next.

- You can load registers from memory with the ld, dld, or vld instructions.
- You can store data in registers into memory with the st, dst, or vst instructions.
- Load or store instruction data sizes include:
  - 8-bit bytes use suffix .b (for example, ld.b),
  - 16-bit halfwords use suffix .h (for example, st.h),
  - 32-bit word is the default (for example, st); however, vector loads or stores use the suffix .s (for example, vst.s)
  - 64-bit doublewords use suffix .d (for example, st.d or vst.d). Note that when accessing TC and VC on-chip registers with a load or store instruction (ld/vld or st/vst, respectively), you cannot use a 64-bit doubleword data size.
- Byte loads can use left extensions of either sign bits or zero (ld.b or ld.ub, respectively).
- Halfword loads can use left extensions of either sign bits or zero (ld.h or ld.uh, respectively).
- Data must be aligned on a boundary that is a multiple of the size of the data item.

- Direct external access (DEA) provides a direct path to external memory and bypasses the cache controller protocol; contents of cache are not affected by the DEA operation, and DEA operations must be used with care.
- The vld and vst instructions require global interrupts to be enabled (IE[[ie]] = 1).

Table 10–2. Load and Store Instructions According to Data Size

	Load	DEA Load	Store	DEA Store	Vector Load	Vector Store
Signed byte	ld.b	dld.b	st.b	dst.b		
Unsigned byte	ld.ub	dld.ub	st.b	dst.b		
Signed halfword	ld.h	dld.h	st.h	dst.h		
Unsigned halfword	ld.uh	dld.uh	st.h	dst.h		
32-bit word	ld	dld	st	dst	vld.s	vst.s
64-bit doubleword	ld.d	dld.d	st.d	dst.d	vld.d	vst.d

Note: **vld** means that the **vld0** or **vld1** instruction uses the address in the IN0P or IN1P control register, respectively; the **vst** instruction uses the address in the OUTP control register.

Optionally, the instruction can specify that the address (*addr*) calculated by adding the (optionally scaled) *offset* to the *base* is written back into the *base* register at the end of the execute stage. This is the modify addressing mode specified by the **:m** syntax. These two options are summarized next:

- *offset* scaling by data size is provided if the suffix **:s** is used (for example, *offset***:s**).
- *base* register address update is available if the suffix **:m** is used (for example, *base***:m**) to pre-increment the base register.
- The vld and vst instructions post-increment the control register's contents by the number of bytes in the specified data size. The ld and st instructions optionally use pre-incrementing.

Note:

Register loads require an extra cycle before the data is available for use.

10.7.1 External Memory

The MVP can operate in either little-endian or big-endian format. This determination is made at power-up reset, which automatically sets the CONFIG⟦E⟧ bit to the correct endian value (from a hardware signal). For more information about the byte order used for accessing data or 32-bit instructions in external memory, see Section NO TAG, *Endian Ordering*, in the *MVP System-Level Synopsis*.

The normal MP mode of operation with external memory is to make use of the hardware caching for data (Section 6.2, *Data Cache*) and instructions (Section 6.1, *Instruction Cache*).

10.7.2 Direct External Memory Access (DEA)

Although the MP operates primarily on data that has been brought to on-chip cache by using TC data-cache miss service, there are a few cases when you may want to specify a direct external memory access (DEA) in the MP instruction.

There are versions of the load and store instructions (dld and dst) that bypass the data cache for addresses ≥ 0x0200 0000. These instructions are implemented by sending a modified cache request to the TC. This mechanism requires more time than an on-chip load or store (see NO TAG, *External Memory Latency of DEA Cycles*, in the *MVP System-Level Synopsis*) because it carries nearly all of the overhead of a cache subblock miss for each access. The modified request reads or writes an 8-bit byte, 16-bit halfword, 32-bit word, or 64-bit doubleword to or from external memory.

This mechanism allows values to be manipulated without loading or flushing the cache. For more information, see subsection 5.5.1, *Direct External Memory Access (DEA).*

10.7.3 Cache Flushes

The transfer controller (TC) provides MP cache-miss service for data and instruction caches. If data in the data cache has been modified, the TC performs any write-back of data required to service a new block or subblock load. The MP program has the option of forcing a write-back of data in the data cache (if the modified bit D = 1) into external memory by using the dcache instruction:

Operation	Instruction	See page
Flush data-cache subblock	dcachef	MP: 10-74
Clean data-cache subblock	dcachec	MP: 10-74

Note:

The cmnd instruction has bits to reset data cache and/or instruction-cache tags. Present (P) bits are reset to 0 (not resident) and cache tags are reset; dirty (D) bits are reset to 0 (not modified) for data cache only. The cmnd instruction does **not** writeback the contents of the data cache to external memory; therefore, any dirty data in the data cache is lost. See Section 11.10, *Clean Data Cache Using the dcachec Instruction*, to see how data-cache coherency can be preserved.

10.8 Shift Instructions

The shift instructions are implemented via a 32-bit rotate, followed optionally by a merge operation that is controlled by up to two masks: a shiftmask and an endmask. The shift instructions are:

Operation	Instruction	See page
General shift left	sl	MP: 10-135
Shift left, invert endmask	sli	MP: 10-139
General shift right	sr	MP: 10-141
Shift right, invert endmask	sri	MP: 10-144

The operands for these instructions include the following:

- *rotate* amount (either register or a constant)
- *endmask* code (always a constant)
- *source* register
- *destination* register

The rotate can be in either direction, according to the assembled value of bit n (bit 10) in the instruction. Rotation is defined here as rotating left naturally (when bit n = 0), or as the effect of rotating right if the shift amount is optionally negated (when bit n = 1). The *rotate* field determines the rotation amount for data in register *source* as:

source \\ *rotate*
source // *rotate* = *source* \\ (–*rotate*)

where **** is the rotate-left operator (as defined in the PP assembly language) and **//** is the rotate-right operator.

10.8.1 Mask Values for Shift Operations

The sl, sli, sr, and sri shift instructions optionally use a shiftmask and endmask to generate the result. These masks are determined as:

- Endmask options from {**m**|**s**|**z**} (merge fields, sign extend, or zero extend)
  - A 5-bit endmask value or register data determines the endmask from Table 10–3
  - An endmask code of 0 returns an endmask of all 1s
  - Optionally invert the endmask (sli.xx or sri.xx)
- Shiftmask options from {**d**|**e**|**i**} (disabled, enabled, or inverted)
  - A 5-bit shift or rotate amount determines the shiftmask from Table 10–3 if the shiftmask is enabled (sl.ex or sli.ex)
  - A 5-bit shift or rotate amount is negated to determine the shiftmask from Table 10–3 if the shiftmask is enabled (sr.ex or sri.ex)
  - All 1s are effectively used for the shiftmask if the shiftmask is disabled (sl.dx, sli.dx, sr.dx, or sri.dx)
  - A code of 0 returns a shiftmask of all 0s for the merge option (m) or if the shiftmask is to be inverted (i)
  - Optionally invert the shiftmask (sl.ix, sli.ix, sr.ix, or sri.ix)
- A composite mask (CM) is generated as CM = (shiftmask) & (endmask) to be used in forming the final output as for shift/rotate left:

  ```
  ((source \\ rotate) & CM) OR (dest & ~CM)→ dest
  ```

  or for shift/rotate right:

  ```
  ((source // rotate) & CM) OR (dest & ~CM) → dest
  ```

Table 10–3. Mask Values for Shift Operations

Code	Mask in Hex	Code	Mask in Hex	Code	Mask in Hex
0	0x0000 0000	11	0x0000 07FF	22	0x003F FFF
1	0x0000 0001	12	0x0000 0FFF	23	0x007F FFF
2	0x0000 0003	13	0x0000 1FFF	24	0x00FF FFFF
3	0x0000 0007	14	0x0000 3FFF	25	0x01FF FFFF
4	0x0000 000F	15	0x0000 7FFF	26	0x03FF FFFF
5	0x0000 001F	16	0x0000 FFFF	27	0x07FF FFFF
6	0x0000 003F	17	0x0001 FFFF	28	0x0FFF FFFF
7	0x0000 007F	18	0x0003 FFFF	29	0x1FFF FFFF
8	0x0000 00FF	19	0x0007 FFFF	30	0x3FFF FFFF
9	0x0000 01FF	20	0x000F FFFF	31	0x7FFF FFFF
10	0x0000 03FF	21	0x001F FFFF	32	0xFFFF FFFF

Instruction syntax consists of one of the four primary mnemonics (shown in Table 10–4), followed by one of the eight suffixes from Table 10–5. This gives a total of 32 mnemonics.

Table 10–4. Prefix to Shift Instruction Mnemonics

Mnemonic	n	i	Instruction
sl	0	0	Shift left
sli	0	1	Shift left and invert endmask
sr	1	0	Shift right
sri	1	1	Shift right and invert endmask

Note: **n** = Bit 10 in the 32-bit assembled instruction is left or right rotation.
i = Bit 11 in the 32-bit assembled instruction is true or inverted endmask.

Table 10–5. Suffix to Shift Instruction Mnemonics

Suffix	Shift Mask	0 means	Merge Option	Merge Field
.dz	Disabled		Zero	000
.dm	Disabled		Merge	001
.ds	Disabled		Sign	010
.ez	Enable	32	Zero	011
.em	Enable	0	Merge	100
.es	Enable	32	Sign	101
.iz	Invert	0	Zero	110
.im	Invert	0	Merge	111
.is	Not Applicable			

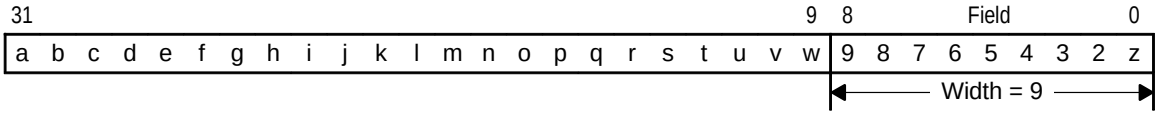
Note:

The merge field is the 3-bit binary suffix encoding for the assembled shift instructions.

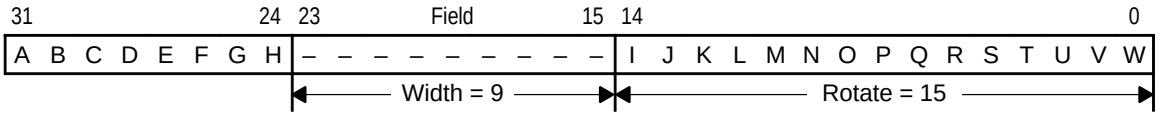
Example 10–2 illustrates examples of masks used in shift operations. The ins instructions use the sl.im shift instruction from Table 10–4 and Table 10–5.

Example 10–2. Masks Used in Shift Operations

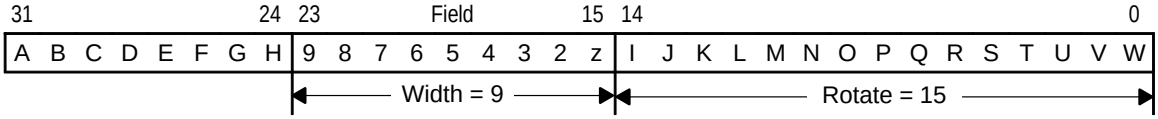
```
ins   15, 15+9, Source, Dest          ; insert Source bits 0–8 into Destination bits 15–23
```



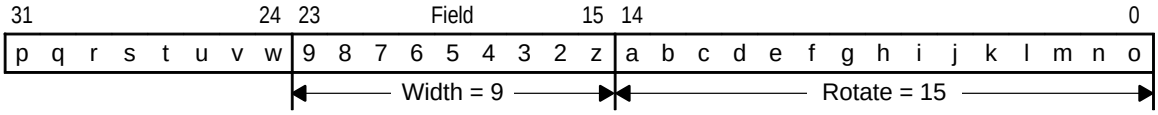
Input Source Register



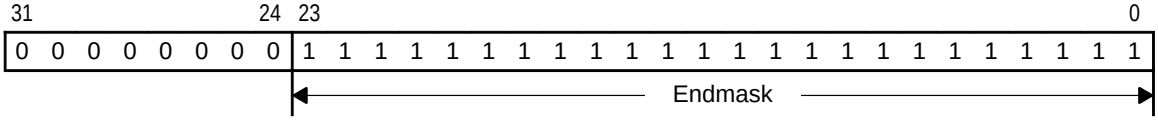
Input Destination Register



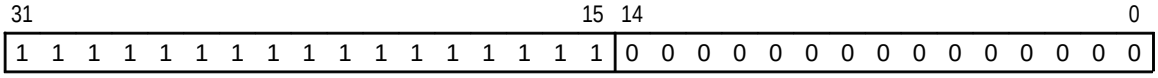
Desired Result in Destination Register



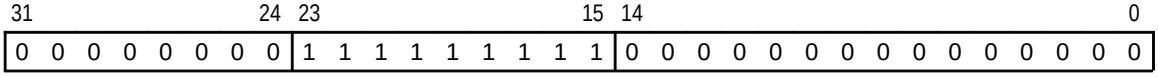
Input Source Data Rotated Left by 15 Bits



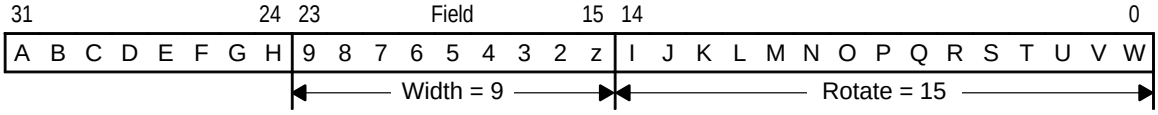
Endmask = Mask (Rotate + Width = 15 + 9 = 24)



Shiftmask = ~ Mask (Rotate = 15)



Composite Mask (CM) = Endmask & Shiftmask



(Rotated-Source & CM) OR (Destination & ~CM) → Destination

10.8.2 Alternative Shift Mnemonics

The assembler provides alternate mnemonics for all the normal shift operations (see Table 10–6).

Table 10–6. Alternate Shift Mnemonics

Operation	Alternate	Uses
Insert field	ins	sl.im
Extract signed field	exts	sr.ds
Extract unsigned field	extu	sr.dz
Rotate left	rotl	sl.iz
Rotate right	rotr	sr.dz
Shift left logical	shl	sl.iz
Shift right arithmetic	sra	sr.es
Shift right logical	srl	sr.ez

10.9 Considerations When Using the MP Instruction Set

When programming with the MP instruction set, you must be aware of how the MP handles the following:

- Conversion of target labels into offsets
- Special treatment of registers r0 and r1
- Using double-precision floating-point registers as parameters
- Exceptions—traps and interrupts
- Scoreboard delay
- Long immediates

10.9.1 Conversion of Target Labels Into Offsets

- If *target* is a label, the assembler computes the *offset* value between label *target* and the location of this instruction ($) as *offset* = (*target* – $) >> 2.
- Signed offset values are always stored as 32-bit word counts.

10.9.2 Special Treatment of Registers r0 and r1

- Integer read/write
  - If the source register is r0, the value read is 0.
  - If the destination register is r0, the result is discarded.
- Floating-point read
  - Double-precision reads from r0 put r1 on the most significant half of the bus and the value 0 on the least significant half of the bus.
  - Single-precision reads from r0 put the value 0 on the single-precision bus.
  - Single-precision reads from r1 put r1 on the single-precision bus.
  - Double-precision reads from r1 are illegal.
- Floating-point write
  - Double-precision writes to r0 **do not** change the contents of r1.
  - Double-precision writes to r0 **do not** scoreboard r0/r1.
  - Single-precision writes to r0 **do not** scoreboard r0.
  - Double-/single-precision writes to r1 are illegal.
  - For any instruction that can write to an accumulator (vmac, vmsc, vmsub, vrnd.dx, and vrnd.sx), r0 is an illegal destination register.
- Register loads using dld, dld.u, ld, ld.u, and vld instructions
  - For the doubleword data size, data discarded for r0, and r1 is illegal.
  - For word, halfword, and byte data sizes, data is discarded for r0; r1 is legal (however, halfword and byte data sizes are not recommended).
- Register stores using dst, st, and vst instructions
  - For the doubleword data size, zero data from r0 is written to the LSword, and data from r1 is written to the MSword.
  - For word, halfword, and byte data sizes, zero data from r0 is written; data from r1 is written (however, halfwords and bytes are not recommended).

10.9.3 Using Double-Precision Floating-Point Registers

- All double-precision arguments must be addressed as even registers.
- Double-precision arguments are stored in an even,odd register pair (for example, r6,r7); the odd register holds the 32 MSB of the 64-bit doubleword (sign, exponent, upper mantissa), and the even register holds the 32 LSBs (lower mantissa).
- A 64-bit double-precision register of r0 is invalid.

Note:

A load (ld.d or vld.d) or store (st.d or vst.d) of double-precision floating-point numbers is independent of big- or little-endian external memory organization.

10.9.4 Exceptions—Traps and Interrupts

Exception interrupts or traps can occur in the MP as a result of the following (see Table 9–1, *Maskable Interrupt Priorities and Vector Addresses*):

- Integer overflow
- Floating-point exceptions
- Illegal instruction or memory address faults
- MP, PP, TC, or VC interrupts
- External interrupts
- Emulation traps
- Program issues a trap instruction

10.9.5 Scoreboard Delay

The registers and vector accumulators are scoreboarded so that an instruction cannot continue when a source value is not yet available. Reasons for the source or memory scoreboard delay include:

- Previous load has not completed (data is not in cache—a data cache fill is needed)
- The dld and dst instructions require more cycles for direct external memory access
- Previous floating-point operation has not completed
- Previous vector operation for this accumulator has not completed

10.9.6 Long Immediates

If an integer constant exceeds 15 bits, then most (not all) instructions provide a 32-bit long immediate form (for example, add 0x12345678,r0,r7).

If the operation is floating-point, the long-immediate source1 must be single-precision (for example, fadd.sdd –0.75,r6,r8).

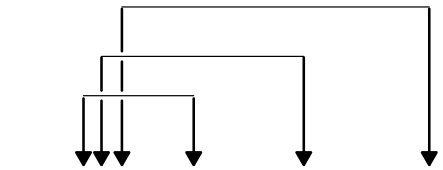
Note:

For all nonannulled branches, you cannot use a long-immediate operand in the branch delay slot instruction (the first instruction following the branch instruction).

10.10 Optional Vector Operation and Sample Precisions

Parallel vector operations are enclosed in double angular brackets "≪≫" when the instruction supports more than one vector operation. This is shown in Example 10–3.

Example 10–3. Optional Vector Operation and Sample Precisions



vmac.ssd *source1, source2,* **0**, *dest* ≪||*vector load or store*≫

- *source1* = single-precision floating point
- *source2* = single-precision floating point
- *dest* = double-precision floating point

The source and destination precisions are also illustrated in Example 10–3. Note that the source and destination precisions follow the period "." in the vmac instruction shown above (**i** is signed integer, **u** is unsigned integer, **s** is single-precision floating point, and **d** is double-precision floating point). Embedded blanks are prohibited between the mnemonic, decimal point, and precisions. Not all combinations are supported; refer to the specific instruction for combinations that are available.

10.10.1Mixing Precisions

The MP instruction set allows for many permutations of operand types within a single instruction. A subset of all possible permutations of mixed-precision operands is supported in the floating-point unit for any single arithmetic instruction (as shown in Example 10–4). However, not all instructions support all types (for example, fmpy.iii is supported, but fadd.iii is not); consult the individual instructions supported in Section 10.12 for specific types.

In addition to mixed types in regular operations, conversions between types is useful. Conversion between types is done in the floating-point add unit. Example 10–5 shows the conversions. These include converting a signed or an unsigned integer to either single- or double-precision floating-point format, converting it back, and converting between the two floating-point precisions.

Example 10–4. Floating-Point Operand Types

SP,SP→SP	DP,SP→DP	I,I→I
SP,SP→DP	DP,DP→DP	U,U→U
SP,DP→DP		

Note: SP = 32-bit single-precision floating point
DP = double-precision floating point
I = 32-bit signed integer
U = 32-bit unsigned integer

Example 10–5. Floating-Point Conversion Types

SP→SP	DP→SP	I→SP
SP→DP	DP→DP	I→DP
SP→I	DP→I	U→SP
SP→U	DP→U	U→DP

Notes: 1) SP = 32-bit single-precision floating point
DP = double-precision floating point
I = 32-bit signed integer
U = 32-bit unsigned integer

2) The definitive list of allowed precision combinations is given with the description of each instruction.

10.11 Numerical Summary of Instructions

Table 10–8 to Table 10–10 summarize the instruction set encoding. Detailed descriptions of all the instructions are included in Section 10.12. Table 10–7 provides a key to abbreviations used in these tables.

Table 10–7. Key to Format Tables

Notation	Definition
–	Reserved bit; code as 0
a	Floating-point accumulator select (bit 16 = MSB; bit 11 = LSB)
A	Annul instruction in delay slot if branch taken
C	Constant operands rather than register
d	Destination precision(s) for vector: ❑ For floating-point operations, 0 = single precision, 1 = double precision ❑ For integer operations, 0 = signed integer, 1 = unsigned integer
D	Direct external memory access bit
E	Emulation trap bit
F	Clear the present flags, as well as dirty flags
i	Invert the sense of the endmask
I	Long-immediate 32-bit data
m	Parallel memory operation specifier
M	Modify, write back modified address to register file
n	Rotate sense for shifting
s	Source precision(s) for vector: ❑ For floating-point operations, 0 = single precision, 1 = double precision ❑ For integer operations, 0 = signed integer, 1 = unsigned integer
S	Scale offset by data size
SZ	Size (byte, halfword, word, doubleword = 0, 1, 2, 3)
P	Destination precision for parallel load/store (0 = single, 1 = double)
P1	Precision of source1 operand†
P2	Precision of source2 operand†
PD	Precision of destination result†
RM	Rounding mode (0, 1, 2, 3 = N, Z, P, M, respectively)
u	Reserved for user/supervisor space indicator
U	Unsigned form
Z	Use 0 rather than accumulator (a)

† See the individual instructions for values.

Table 10–8. Short-Immediate Format Instructions

	31–27	26–22	21–15	14–11	9–5	4–0
illop0	Dest	Source	0 0 0 0 0 0 0	Unsigned Immediate		
trap	– – – – E	– – – – –	0 0 0 0 0 0 1	Unsigned Trap Number		
cmnd	– – – – –	– – – – –	0 0 0 0 0 1 0	Unsigned Immediate		
rdcr	Dest	– – – – –	0 0 0 0 1 0 0	Unsigned Control Register Number		
swcr	Dest	Source	0 0 0 0 1 0 1	Unsigned Control Register Number		
brcr	– – – – –	– – – – –	0 0 0 0 1 1 0	Unsigned Control Register Number		
shift.dz†	Dest	Source	0 0 0 1 0 0 0	– – – i n	Endmask	Rotate
shift.dm†	Dest	Source	0 0 0 1 0 0 1	– – – i n	Endmask	Rotate
shift.ds†	Dest	Source	0 0 0 1 0 1 0	– – – i n	Endmask	Rotate
shift.ez†	Dest	Source	0 0 0 1 0 1 1	– – – i n	Endmask	Rotate
shift.em†	Dest	Source	0 0 0 1 1 0 0	– – – i n	Endmask	Rotate
shift.es†	Dest	Source	0 0 0 1 1 0 1	– – – i n	Endmask	Rotate
shift.iz†	Dest	Source	0 0 0 1 1 1 0	– – – i n	Endmask	Rotate
shift.im†	Dest	Source	0 0 0 1 1 1 1	– – – i n	Endmask	Rotate
and.tt	Dest	Source2	0 0 1 0 0 0 1	Unsigned Immediate		
and.tf	Dest	Source2	0 0 1 0 0 1 0	Unsigned Immediate		
and.ft	Dest	Source2	0 0 1 0 1 0 0	Unsigned Immediate		
xor	Dest	Source2	0 0 1 0 1 1 0	Unsigned Immediate		
or.tt	Dest	Source2	0 0 1 0 1 1 1	Unsigned Immediate		
and.ff	Dest	Source2	0 0 1 1 0 0 0	Unsigned Immediate		
xnor	Dest	Source2	0 0 1 1 0 0 1	Unsigned Immediate		
or.tf	Dest	Source2	0 0 1 1 0 1 1	Unsigned Immediate		
or.ft	Dest	Source2	0 0 1 1 1 0 1	Unsigned Immediate		
or.ff	Dest	Source2	0 0 1 1 1 1 0	Unsigned Immediate		
ld	Dest	Base	0 1 0 0 M SZ	Signed Offset		
ld.u	Dest	Base	0 1 0 1 M SZ	Signed Offset		
st	Source	Base	0 1 1 0 M SZ	Signed Offset		
dcache	– – – – F	Source2	0 1 1 1 M 0 0	Signed Offset		
bsr	Link	– – – – –	1 0 0 0 0 0 A	Signed Offset		
jsr	Link	Base	1 0 0 0 1 0 A	Signed Offset		
bbz	BITNUM	Source	1 0 0 1 0 0 A	Signed Offset		
bbo	BITNUM	Source	1 0 0 1 0 1 A	Signed Offset		
bcnd	Cond	Source	1 0 0 1 1 0 A	Signed Offset		
cmp	Dest	Source2	1 0 1 0 0 0 0	Signed Immediate		
add	Dest	Source2	1 0 1 1 0 0 U	Signed Immediate		
sub	Dest	Source2	1 0 1 1 0 1 U	Signed Immediate		

† Alternate mnemonics for the shift instructions are listed on page MP: 10-135.
‡ Missing opcode numeric values are reserved.

Table 10–9. Register/Long-Immediate Format Instructions

	31–27	26–22	21–12	11–5	4–0
trap	– – – – E	– – – – –	1 1 0 0 0 0 0 0 0 1	I	– – – – – – –	IND TR
cmnd	– – – – –	– – – – –	1 1 0 0 0 0 0 0 1 0	I	– – – – – – –	Source1
rdcr	Dest	– – – – –	1 1 0 0 0 0 0 1 0 0	I	– – – – – – –	IND CR
swcr	Dest	Source2	1 1 0 0 0 0 0 1 0 1	I	– – – – – – –	IND CR
brcr	– – – – –	– – – – –	1 1 0 0 0 0 0 1 1 0	I	– – – – – – –	IND CR
shift.dz†	Dest	Source	1 1 0 0 0 0 1 0 0 0	0	i	n	Endmask	Rotate Reg.
shift.dm†	Dest	Source	1 1 0 0 0 0 1 0 0 1	0	i	n	Endmask	Rotate Reg.
shift.ds†	Dest	Source	1 1 0 0 0 0 1 0 1 0	0	i	n	Endmask	Rotate Reg.
shift.ez†	Dest	Source	1 1 0 0 0 0 1 0 1 1	0	i	n	Endmask	Rotate Reg.
shift.em†	Dest	Source	1 1 0 0 0 0 1 1 0 0	0	i	n	Endmask	Rotate Reg.
shift.es†	Dest	Source	1 1 0 0 0 0 1 1 0 1	0	i	n	Endmask	Rotate Reg.
shift.iz†	Dest	Source	1 1 0 0 0 0 1 1 1 0	0	i	n	Endmask	Rotate Reg.
shift.im†	Dest	Source	1 1 0 0 0 0 1 1 1 1	0	i	n	Endmask	Rotate Reg.
and.tt	Dest	Source2	1 1 0 0 1 0 0 0 0 1	I	– – – – – – –	Source1
and.tf	Dest	Source2	1 1 0 0 1 0 0 1 0	I	– – – – – – –	Source1
and.ft	Dest	Source2	1 1 0 0 1 0 1 0 0	I	– – – – – – –	Source1
xor	Dest	Source2	1 1 0 0 1 0 1 1 0	I	– – – – – – –	Source1
or.tt	Dest	Source2	1 1 0 0 1 0 1 1 1	I	– – – – – – –	Source1
and.ff	Dest	Source2	1 1 0 0 1 1 0 0 0	I	– – – – – – –	Source1
xnor	Dest	Source2	1 1 0 0 1 1 0 0 1	I	– – – – – – –	Source1
or.tf	Dest	Source2	1 1 0 0 1 1 0 1 1	I	– – – – – – –	Source1
or.ft	Dest	Source2	1 1 0 0 1 1 1 0 1	I	– – – – – – –	Source1
or.ff	Dest	Source2	1 1 0 0 1 1 1 1 0	I	– – – – – – –	Source1
ld	Dest	Base	1 1 0 1 0 0	M	SZ	I	S	D	– – – – –	Offset
ld.u	Dest	Base	1 1 0 1 0 1	M	SZ	I	S	D	– – – – –	Offset
st	Source	Base	1 1 0 1 1 0	M	SZ	I	S	D	– – – – –	Offset
dcache	– – – – F	Source2	1 1 0 1 1 1	M	0 0	I	0	0	– – – – –	Source1
bsr	Link	– – – – –	1 1 1 0 0 0 0 0	A	I	– – – – – – –	Offset
jsr	Link	Base	1 1 1 0 0 0 1 0	A	I	– – – – – – –	Offset
bbz	BITNUM	Source	1 1 1 0 0 1 0 0	A	I	– – – – – – –	Target
bbo	BITNUM	Source	1 1 1 0 0 1 0 1	A	I	– – – – – – –	Target
bcnd	Cond	Source	1 1 1 0 0 1 1 0	A	I	– – – – – – –	Target
cmp	Dest	Source2	1 1 1 0 1 0 0 0 0	I	– – – – – – –	Source1
add	Dest	Source2	1 1 1 0 1 1 0 0	U	I	– – – – – – –	Source1
sub	Dest	Source2	1 1 1 0 1 1 0 1	U	I	– – – – – – –	Source1

† Alternate mnemonics for the shift instructions are listed on page MP:10-135.
‡ Missing opcode numeric values are reserved.

Table 10–10. Miscellaneous Instructions

	31–27	26–22	21–16	15–13	12	11	10	9	8	7	6	5	4–0
vadd	Mem Dst	Source2/Dest	1 1 1 1 0 – 0 0	0	I	–	m	P	–	d	m	s	Source1
vsub	Mem Dst	Source2/Dest	1 1 1 1 0 – 0 0	1	I	–	m	P	–	d	m	s	Source1
vmpy	Mem Dst	Source2/Dest	1 1 1 1 0 – 0 1	0	I	–	m	P	–	d	m	s	Source1
vmsub	Mem Dst	Dest	1 1 1 1 0 a 0 1	1	I	a	m	P	Z	–	m	–	Source1
vrnd (FP)	Mem Dst	Dest	1 1 1 1 0 a 1 0	0	I	a	m	P	PD		m	s	Source1
vrnd (Int)	Mem Dst	Dest	1 1 1 1 0 – 1 0	1	I	–	m	P	–	d	m	s	Source1
vmac	Mem Dst	Source2	1 1 1 1 0 a 1 1	0	I	a	m	P	Z	–	m	–	Source1
vmsc	Mem Dst	Source2	1 1 1 1 0 a 1 1	1	I	a	m	P	Z	–	m	–	Source1
fadd	Dest	Source2	1 1 1 1 1 0 0 0	0	I	–	PD		P2		P1		Source1
fsub	Dest	Source2	1 1 1 1 1 0 0 0	1	I	–	PD		P2		P1		Source1
fmpy	Dest	Source2	1 1 1 1 1 0 0 1	0	I	–	PD		P2		P1		Source1
fdiv	Dest	Source2	1 1 1 1 1 0 0 1	1	I	–	PD		P2		P1		Source1
frndx	Dest	– – – – –	1 1 1 1 1 0 1 0	0	I	–	PD		RM		P1		Source1
fcmp	Dest	Source2	1 1 1 1 1 0 1 0	1	I	–	–	–	P2		P1		Source1
fsqrt	Dest	– – – – –	1 1 1 1 1 0 1 1	1	I	–	PD		–	–	P1		Source1
lmo	Dest	Source	1 1 1 1 1 1 0 0	0	–	–	–	–	–	–	–	–	–
rmo	Dest	Source	1 1 1 1 1 1 0 0	1	–	–	–	–	–	–	–	–	–
estop	– – – – –	– – – – –	1 1 1 1 1 1 1 1	0	–	–	–	–	–	–	–	–	–
illopF	– – – – –	– – – – –	1 1 1 1 1 1 1 1	1	C	–	–	–	–	–	–	–	–

Notes: 1) Missing opcode numeric values are reserved.
2) FP = Floating-point input
3) Int = Integer input

10.12 Alphabetical Instruction Reference

The remainder of this section is an alphabetical reference of the MP assembly language instructions. Most instruction descriptions begin on a new page and contain the following information:

- **Syntax:** Shows you how to enter an instruction. (The Preface describes the symbols used in instruction syntaxes.)
- **Operation:** Illustrates the effects of instruction execution on registers and memory.
- **Operands:** Defines the type of value that an operand can be.
  - **a** is a double-precision floating-point accumulator and represents a0, a1, a2, or a3
  - **I** is a short 15-bit immediate operand
  - **L** is a long 32-bit immediate operand
  - R is a 32-bit register (double-precision values require an even register)
  - **r0** is a special register—read returns 0, writes are discarded
  - **0x** is a prefix for hexadecimal constants (for example, 0x1234 = 1234_{16})
- **Encoding:** Shows the object code generated by an instruction. In general, encodes are provided in three forms: immediate, register, and long immediate. Any fields or bits that are not specified are reserved (see Section 2.1).
- **Description:** Discusses the purpose of the instruction and any other general information related to the instruction.
- **Latency:** Provided for floating-point and vector operations and represents the earliest time that subsequent instructions can begin. The time for instructions (assuming no cache misses or crossbar contention) is as follows:
  - Most instructions (no branches, long immediates, floating point, vector operations) require one cycle.
  - Instructions with long immediates require an additional cycle.
  - Branches or jumps require two cycles to (optionally) branch to label *target.*
  - Integer multiplies require three cycles (see Table 8–2).
  - Floating-point operations are shown in Table 8–2.
  - Vector operations are shown in Table 8–2.
- **Examples:** Show the effects of the instruction on memory and registers using various sets of data and initial conditions.

10.12.1 Summary of Instructions

Table 10–11 summarizes the MP instruction set. Descriptions of each of these instructions follow the table.

Table 10–11. Summary of MP Instructions

Instruction	Description	See Page
add	Signed integer add	MP:10-49
addu	Unsigned integer add	MP:10-50
and, and.tt	Bitwise AND	MP:10-51
and.ff	Bitwise AND with ones complement	MP:10-52
and.ft	Bitwise AND with ones complement	MP:10-53
and.tf	Bitwise AND with ones complement	MP:10-54
bbo, bbo.a	Branch bit one	MP:10-55
bbz, bbz.a	Branch bit zero	MP:10-58
bcnd, bcnd.a	Branch conditional	MP:10-61
br, br.a	Branch always	MP:10-64
brcr	Branch control register	MP:10-66
bsr, bsr.a	Branch and save return	MP:10-68
cmnd	Send command	MP:10-70
cmp	Integer compare	MP:10-72
dcache	Flush data-cache subblock	MP:10-74
dld	Direct load signed data into register	MP:10-76
dld.u	Direct load unsigned data into register	MP:10-79
dst	Direct store data into memory	MP:10-82
estop	Emulation stop	MP:10-85
etrap	Emulation trap	MP:10-86
exts	Extract signed field	MP:10-87
extu	Extract unsigned field	MP:10-89
fadd	Floating-point add	MP:10-91
fcmp	Floating-point compare	MP:10-93
fdiv	Floating-point divide	MP:10-96
fmpy	Integer or floating-point multiply	MP:10-97

Table 10–11. Summary of MP Instructions (Continued)

Instruction	Description	See Page
frndm	Convert/round to minus infinity	MP:10-99
frndn	Convert/round to nearest	MP:10-101
frndp	Convert/round to positive infinity	MP:10-102
frndz	Convert/round to zero	MP:10-103
fsqrt	Floating-point square root	MP:10-104
fsub	Floating-point subtract	MP:10-105
illop	Illegal instruction	MP:10-106
ins	Insert field	MP:10-107
jsr, jsr.a	Jump and save return	MP:10-111
ld	Load signed data into register	MP:10-113
ld.u	Load unsigned data into register	MP:10-116
lmo	Leftmost one	MP:10-119
nop	No operation	MP:10-121
or, or.tt	Bitwise OR	MP:10-122
or.ff	Bitwise OR with ones complement	MP:10-123
or.ft	Bitwise OR with ones complement	MP:10-124
or.tf	Bitwise OR with ones complement	MP:10-125
rdcr	Read control register	MP:10-126
rmo	Rightmost one	MP:10-127
rotl	Rotate register left	MP:10-129
rotr	Rotate register right	MP:10-131
shl	Shift register left logical	MP:10-133
sl.xx	Shift register left	MP:10-135
sli.xx	Shift register left with inverted endmask	MP:10-139
sr.xx	Shift register right	MP:10-141
sra	Shift register right arithmetic	MP:10-142
sri.xx	Shift register right with inverted endmask	MP:10-144
srl	Shift register right logical	MP:10-146
st	Store data into memory	MP:10-148

Table 10–11. Summary of MP Instructions (Continued)

Instruction	Description	See Page
sub	Signed integer subtract	MP:10-151
subu	Unsigned integer subtract	MP:10-152
swcr	Swap control register	MP:10-153
trap	Trap	MP:10-155
vadd	Vector floating-point add	MP:10-157
vld0, vld1	Vector load data into register	MP:10-159
vmac	Vector floating-point multiply and add to accumulator	MP:10-161
vmpy	Vector floating-point multiply	MP:10-164
vmsc	Vector floating-point multiply and subtract from accumulator	MP:10-166
vmsub	Vector floating-point subtract accumulator from source	MP:10-169
vrnd.sx, vrnd.dx	Vector round with floating-point input	MP:10-171
vrnd.ix, vrnd.ux	Vector round with integer input	MP:10-173
vst	Vector store data into memory	MP:10-175
vsub	Vector floating-point subtract	MP:10-177
wrcr	Write control register	MP:10-179
xnor	Bitwise exclusive NOR	MP:10-181
xor	Bitwise exclusive OR	MP:10-182

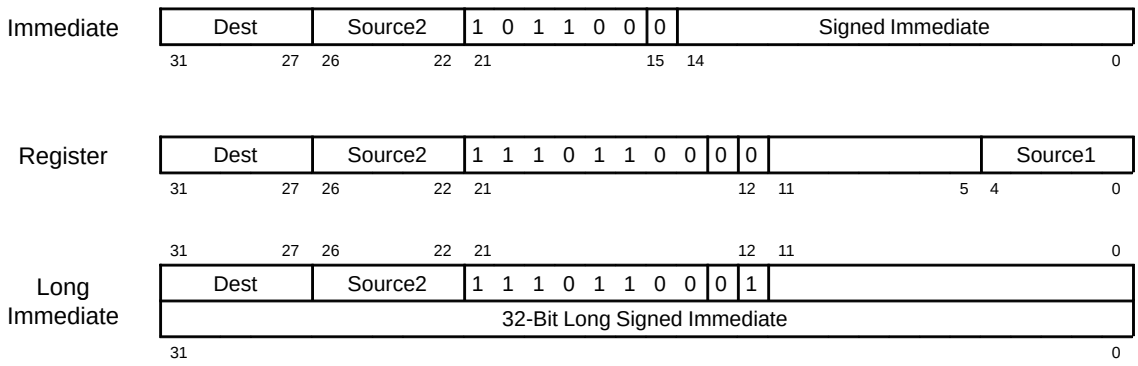
Syntax **add** *source1,source2,dest*

Operation *source1* + *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from the *source1* register (or the immediate data) is added to data from the *source2* register, and the result is written into the *dest* register. If the add operation overflows, an integer overflow is signaled (INTPEN[[io]] is set to 1).

Example

```
                        ;     r7          r8          r9
                        ;   before      before      after

add  r7,r8,r9           ;  12345678    12345678    2468ACF0
add  -3,r8,r9           ;      -       54545478    54545475
add  0x12345678,r8,r9   ;      -       12345678    2468ACF0
add  0x789abcde,r8,r9   ;      -       12345678    8ACF1356  (overflows)
                        ;                                    INTPEN[[io]] = 1
```

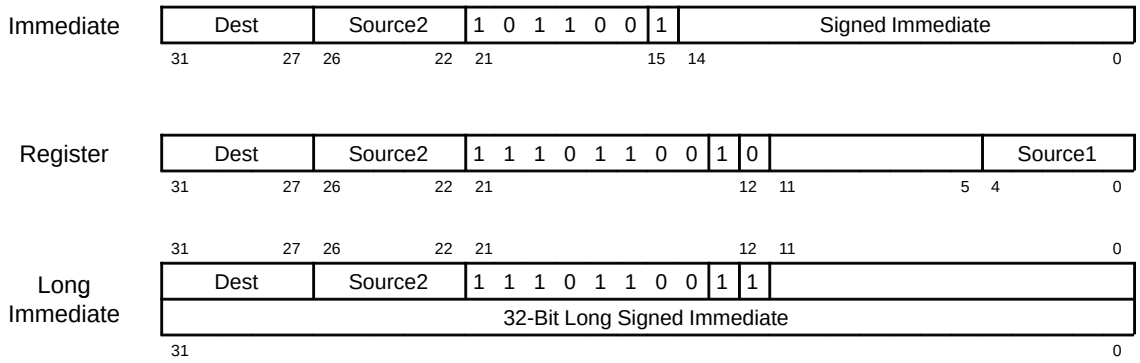
Syntax **addu** *source1,source2,dest*

Operation *source1* + *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description

Data from the *source1* register (or the immediate data) is added to data from the *source2* register, and the result is written into the *dest* register. If the addu operation overflows, **no** integer overflow is signaled (INTPEN[[io]] remains unchanged).

Using signed immediate data allows the instruction

addu –2,r8,r9

to perform r8 – 2 = r9. Additionally,

subu 2,r8,r9

returns 2 – r8 = r9. Moreover, a constant can be either operand of the subtraction operator.

To prevent unwanted integer overflows, use the addu instruction when initializing or combining signed and unsigned integers, pointers, and addresses.

Example

```
                          ;     r7         r8         r9
                          ;   before     before     after

addu  r7,r8,r9            ;  12345678   12345678   2468ACF0
addu  0x0001,r8,r9        ;      –      54545478   54545479
addu  0x12345678,r8,r9    ;      –      12345678   2468ACF0
addu  0x789abcde,r8,r9    ;      –      12345678   8ACF1356  (no overflow)
                          ;                                 INTPEN[[io]] is
                          ;                                 unchanged
```

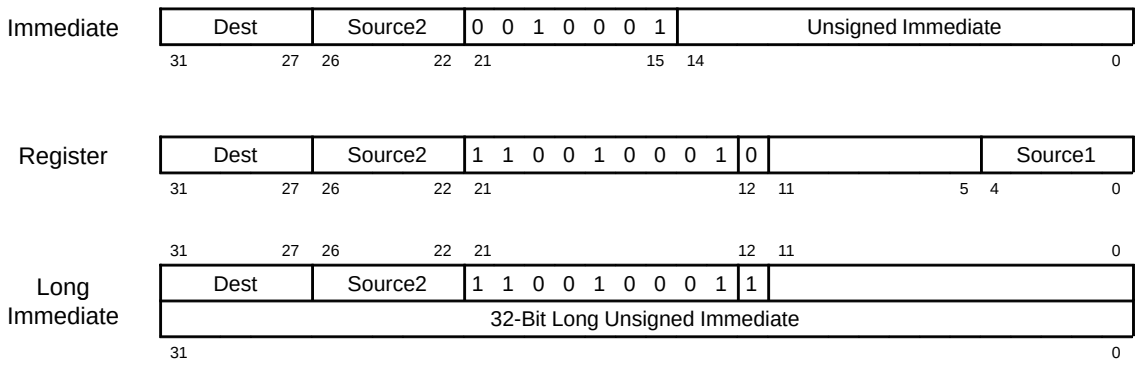
Syntax **and**[**.tt**] *source1,source2,dest*

Operation *source1* & *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description The bitwise AND of data from the *source1* register (or the immediate data) and data from the *source2* register is written into the *dest* register.

Example

```
                              ;       r7              r8              r9
                              ;     before          before          after

and     r7,r8,r9              ;    0000000F        12345678        00000008
and     0x7fff,r8,r9          ;        -           12345678        00005678
and     0xff0000ff,r8,r9      ;        -           12345678        12000078

and.tt  r7,r8,r9              ;    0000000F        12345678        00000008
and.tt  0x7fff,r8,r9          ;        -           12345678        00005678
and.tt  0xff0000ff,r8,r9      ;        -           12345678        12000078
```

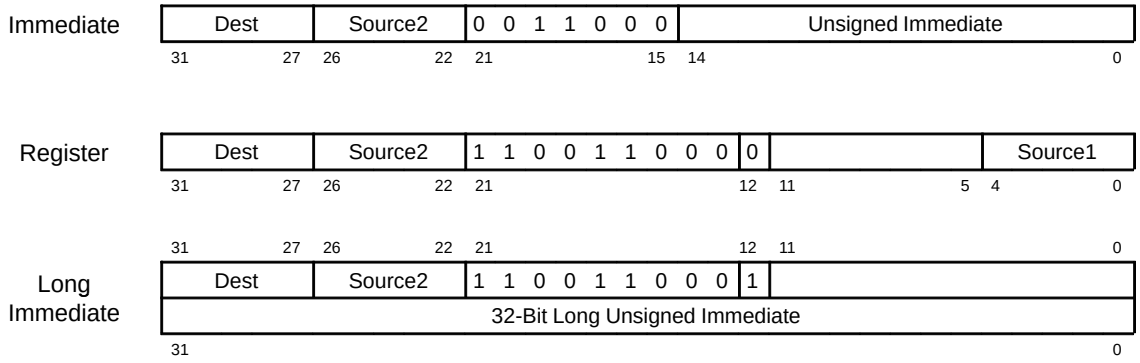
Syntax **and.ff** *source1,source2,dest*

Operation ~(*source1*) & ~(*source2*) → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from the *source1* register (or the immediate data) and data from the *source2* register are both inverted (ones complement), and a bitwise AND of these inverted values is written into the *dest* register.

Example

```
                              ;       r7              r8              r9
                              ;     before          before          after

and.ff  r7,r8,r9              ;    0000000F        12345678        EDCBA980
and.ff  0x7fff,r8,r9          ;        -           12345678        EDCB8000
and.ff  0xff0000ff,r8,r9      ;        -           12345678        00CBA900
```

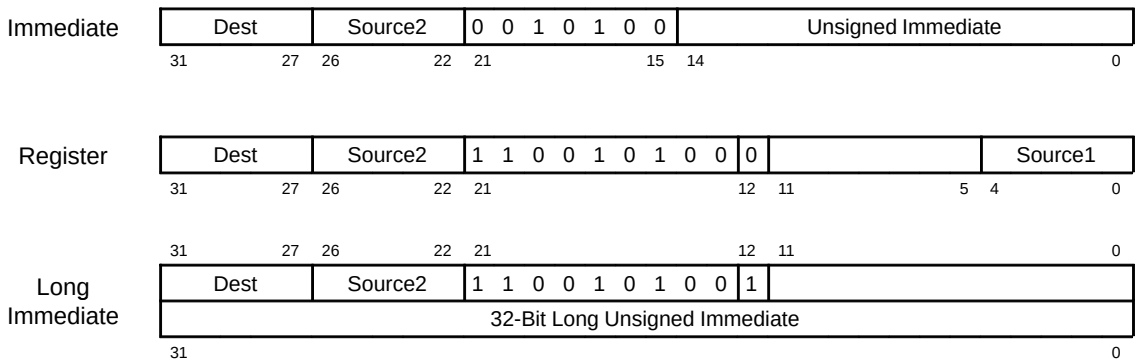
Syntax **and.ft** *source1,source2,dest*

Operation ~(*source1*) & *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from the *source1* register (or the immediate data) is inverted (ones complement), and a bitwise AND of this inverted value and the data from the *source2* register is written into the *dest* register.

Example

```
                              ;       r7              r8              r9
                              ;     before          before          after

and.ft  r7,r8,r9              ;    0000000F        12345678        12345670
and.ft  0x7fff,r8,r9          ;       -            12345678        12340000
and.ft  0xff0000ff,r8,r9      ;       -            12345678        00345600
```

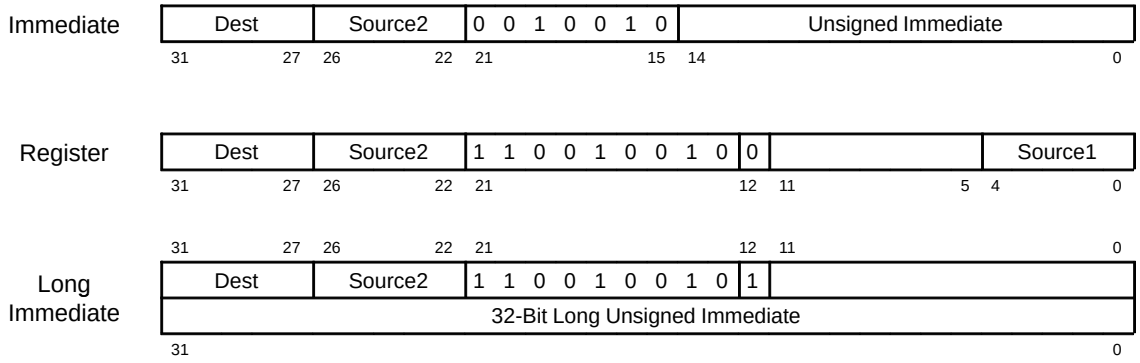
Syntax **and.tf** *source1,source2,dest*

Operation *source1* & ~(*source2*) → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from the *source2* register is inverted (ones complement), and a bitwise AND of this inverted value and the data from the *source1* register (or the immediate data) is written into the *dest* register.

Example

```
                              ;       r7              r8              r9
                              ;     before          before          after

and.tf  r7,r8,r9              ;    0000000F        12345678        00000007
and.tf  0x7fff,r8,r9          ;       -            12345678        00002987
and.tf  0xff0000ff,r8,r9      ;       -            12345678        ED000087
```

Syntax **bbo**[**.a**] *target,source,bitnum*

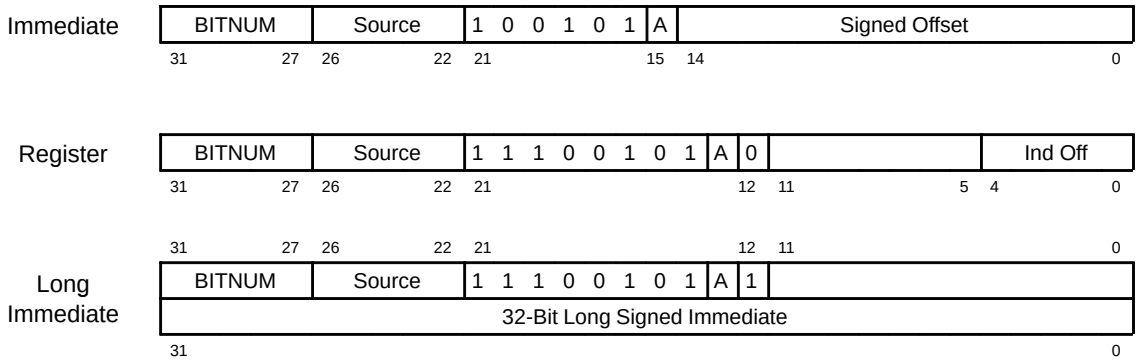
Operation Test *bitnum* of *source* for:

- 1—relative branch to *target*
- 0—continue to next instruction

Operands

I,R,I	Immediate Form
R,R,I	Register Form
L,R,I	Long-Immediate Form

Encoding



Note: A = Annul the next instruction if branch is taken.

Description The value of bit number *bitnum* from the data in the *source* register is tested.

- If that bit value is 1, a relative branch to label *target* is taken (IP + 4*offset → PC).
- If the bit value is 0, the program continues to the next sequential instruction.

The *source* bits are numbered from right to left in the range 0 (LSB) to 31 (MSB). In the register form, the indirect signed offset is held in the *target* register as the number of 32-bit instruction words. There are two forms of this instruction:

- **bbo**—The 32-bit instruction following the bbo instruction (a branch delay slot) is executed. If the test condition is true, then the relative branch is taken.
- **bbo.a**—The instruction following the bbo.a is not executed if the relative branch is taken (a nop cycle is executed instead).

When *target* is a label, the assembler computes the offset value between label *target* and the location of this instruction.

The BITNUM field of the bbo instruction is set to the ones complement of the bit number value (*bitnum*) that you supply (BITNUM is used by the hardware rotator).

Note:

For all nonannulled branches, you cannot use a long-immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction). See subsections 10.4.1 and 10.4.2.

If you perform a cmp (integer compare) instruction before the bbo instruction, you can use a *bitnum* value specified by the form *condition.size*:

- The *condition* prefix specifies the bit offset, as shown in Table 10–12.
- The *size* suffix specifies the LSB of the group of bits, as shown in Table 10–13.

The *bitnum* value is determined by:

bitnum = *source* LSB + *offset* bit number

For example, the *bitnum* le.h is equal to 10 + 3, which is bit 13.

See Figure 10–2, *Bit Fields for the cmp Destination Register*, for an illustration of how the bit fields are configured by byte, halfword, and word groups.

Table 10–12. Types of Conditions

Syntax Prefix	Type of Condition	Offset Bit Number
eq.sz	source1 data equal to source2	size + 0
ne.sz	source1 data not equal to source2	size + 1
gt.sz	source1 data greater than source2	size + 2
le.sz	source1 data less than or equal to source 2	size + 3
lt.sz	source1 data less than source2	size + 4
ge.sz	source1 data greater than or equal to source2	size + 5
hi.sz	source1 data higher than source2	size + 6
ls.sz	source1 data lower than or same as source2	size + 7
lo.sz	source1 data lower than source2	size + 8
hs.sz	source1 data higher or same as source2	size + 9

Table 10–13. Data Sizes

Syntax Suffix	Data Size	Source LSB
sz = b	8 LSBs	size = 00
sz = h	16 LSBs	size = 10
sz = w	all 32 bits	size = 20

If you perform a floating-point compare (fcmp) instruction, the bbo instruction can use a *bitnum* value specified from Table 10–18, *Condition Codes Generated by the fcmp Instruction*.

Example 1

```
                        ; If r7 = 0x1234 5678, value in PC becomes:

bbo     Path8,r7,3      ; IP + 4 × offset (branch to label Path8),
add     0x1234,r0,r9    ; execute this add before the instruction at Path8

bbo.a   r8,r7,3         ; IP + (4 × r8) bytes (Annul branch is taken),
add     0x1234,r0,r9    ; execute nop (not add) before the instruction at
                        ; IP + 4 × r8

bbo.a   r8,r7,31        ; PC + 4 bytes (annul branch is not taken),
add     0x1234,r0,r9    ; this add is executed next
```

Example 2

```
bbo.a   Path8,r7,le.b   ; bit 3 = le.b following an integer compare
                        ; instruction

bbo.a   r8,r7,le.b      ; bit 3 = le.b following an integer compare
                        ; instruction

bbo.a   r8,r7,or.f      ; bit 31 = or.f following a floating-point compare
                        ; instruction

. . .

Path8: . . .
```

Syntax **bbz**[**.a**] *target,source,bitnum*

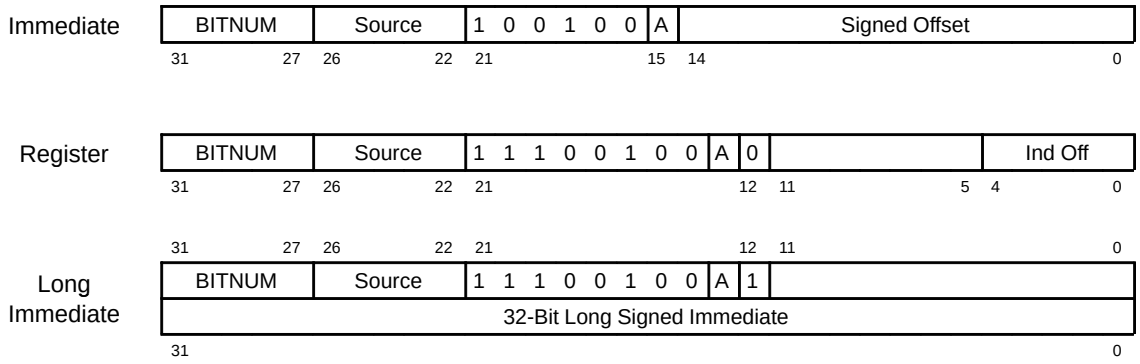
Operation Test *bitnum* of *source* for:

- 0—relative branch to *target*
- 1—continue to next instruction

Operands

I,R,I	Immediate Form
R,R,I	Register Form
L,R,I	Long-Immediate Form

Encoding



Note: A = Annul the next instruction if branch is taken.

Description The value of bit number *bitnum* from the data in the *source* register is tested.

- If that bit value is 0, a relative branch to label *target* is taken (IP + 4*offset → PC).
- If the bit value is 1, the program continues to the next sequential instruction.

The *source* bits are numbered from right to left in the range 0 (LSB) to 31 (MSB). In the register form, the indirect signed offset is held in the *target* register as the number of 32-bit instruction words. There are two forms of this instruction:

- **bbz**—The 32-bit instruction following the bbz instruction (a branch delay slot) is executed. Then if the test condition is true, then the relative branch is taken.
- **bbz.a**—The instruction following the bbz.a is not executed if the relative branch is taken (a nop cycle is executed instead).

When *target* is a label, the assembler computes the offset value between label *target* and the location of this instruction.

The BITNUM field of the bbz instruction is set to the ones complement of the bit number value (*bitnum*) that you supply (BITNUM is used by the hardware rotator).

Note:

For all nonannulled branches, you cannot use a long immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction). See subsections 10.4.1 and 10.4.2.

If you perform a cmp (integer compare) instruction before the bbz instruction, you can use a *bitnum* value specified by the form *condition.size*:

- The *condition* prefix specifies the bit offset, as shown in Table 10–12.
- The *size* suffix specifies the LSB of the group of bits, as shown in Table 10–13.

The *bitnum* value is determined by:

bitnum = *source* LSB + *offset* bit number

For example, the *bitnum* le.h is equal to 10 + 3, which is bit 13.

See Figure 10–2, *Bit Fields for the cmp Destination Register*, for an illustration of how the bit fields are configured by byte, halfword, and word groups.

If you perform a floating-point compare (fcmp) instruction, the bbz instruction can use a *bitnum* value specified from Table 10–18, *Condition Codes Generated by the fcmp Instruction*.

Example 1

```
                        ; If r7 = 0xEDCB A987, value in PC becomes:

bbz     Path9,r7,3      ; IP + 4 × offset (branch to label Path9),
add     0x1234,r0,r9    ; execute this add before the instruction at Path9

bbz.a   r8,r7,3         ; IP + (4 × r8) bytes (Annul branch is taken),
add     0x1234,r0,r9    ; execute nop (not add) before the instruction at
                        ; IP + 4 × r8

bbz.a   r8,r7,31        ; PC + 4 bytes (annul branch is not taken),
add     0x1234,r0,r9    ; this add is executed next
```

Example 2

```
bbz.a  Path9,r7,le.b  ; bit 3 = le.b following an integer compare
                      ; instruction

bbz.a  r8,r7,le.b     ; bit 3 = le.b following an integer compare
                      ; instruction

bbz.a  r8,r7,or.f     ; bit 31 = or.f following a floating-point compare
                      ; instruction

. . .

Path9: . . .
```

Syntax **bcnd**[**.a**] *target,source,code*

Operation Test the condition *code* of the *source* for:

- ❑ true—relative branch to *target*
- ❑ false—continue to next instruction

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding

Note: A = Annul next instruction if branch is taken.

Description Data from the *source* register is tested for condition *code*.

- ❑ If the condition tested is true, then a relative branch to label *target* is taken (IP + 4*offset → PC).
- ❑ If the condition tested is false, then the program continues to the next sequential instruction.

In the register form, the indirect signed offset is held in the *target* register as the number of 32-bit instruction words. There are two forms of this instruction:

- ❑ **bcnd**—The 32-bit instruction following the bcnd instruction (a branch delay slot) is executed. Then, if the test condition is true, the relative branch is taken.
- ❑ **bcnd.a**—The instruction following the bcnd.a is not executed if the relative branch is taken (a nop cycle is executed instead).

If *target* is a label, the assembler computes the *offset* value between label *target* and the location of this instruction.

Note:

For all nonannulled branches, you cannot use a long immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction). See subsections 10.4.1 and 10.4.2.

Condition Codes

Condition *code* syntax is specified by the form *condition.size*:

- The type of *condition* is represented in the lower three bits of assembled *code* field as (<, =, or > 0):

		Bits		
Syntax Prefix	**Type of Condition**	**29**	**28**	**27**
nev	Never	0	0	0
gt0	Greater than zero	0	0	1
eq0	Equals zero	0	1	0
ge0	Greater than or equal to zero	0	1	1
lt0	Less than zero	1	0	0
ne0	Not equal to zero	1	0	1
le0	Less than or equal to zero	1	1	0
alw	Always	1	1	1
	source is	<	0	>

- The data *size* is represented in the top two bits of assembled *code* field as:

		Bits	
Syntax Suffix	**Data Size**	**31**	**30**
.b	8-bit LSbyte	0	0
.h	16-bit LShalfword	0	1
.w	32-bit word	1	0
.d	Reserved	1	1

Example

```
                                ; If r7 = 0xEDCB A987, value in PC
                                ; becomes:

        bcnd    Path7,r7, alw.w ; IP + 4 × offset (branch to label
                                ; Path7 always),
        add     0x1234,r0,r9    ; execute this add before the
                                ; instruction at Path7

        bcnd.a  $+0x320,r7,le0.w ; IP + (0x320) bytes (Annul branch is
                                ; taken if word ≤ 0),
        add     0x1234,r0,r9    ; execute nop (not add) before the
                                ; instruction at IP + 0x320

        bcnd.a  r8,r7,ne0.h     ; IP + (4 × r8) bytes (Annul branch is
                                ; taken if LShalfword ≠ 0),
        add     0x1234,r0,r9    ; execute nop (not add) before the
                                ; instruction at IP + 4 × r8

        bcnd.a  r8,r7,eq0.b     ; PC + 4 bytes (annul branch is not
                                ; taken since byte = 0 is false),
        add     0x1234,r0,r9    ; this add is executed next
Path7:  . . .
```

Note: In the bcnd.a $+0x320 instruction, the assembler encodes the offset as 0x320 ÷ 4 = 0x0C8.

Syntax **br**[**.a**] *target*

Operation Relative branch to *target*

Operands

I	Immediate Form
R	Register Form
L	Long Form

Encoding

Note: A = Annul the next instruction if branch is taken.

Description The assembler uses the **bbz** *target,r0,31* instruction to affect an unconditional relative branch to label *target* (IP + 4*offset → PC). In the register form, the indirect signed offset is held in the *target* register as the number of 32-bit instruction words. There are two forms of this instruction:

- **br**—The 32-bit instruction following the br instruction (a branch delay slot) is executed; then the relative branch is taken.
- **br.a**—The instruction following the br.a is not executed, because the relative branch is always taken (a nop cycle is executed instead).

If *target* is a label, the assembler computes the offset value between label *target* and the location of this instruction.

> **Note:**
>
> For all nonannulled branches, you cannot use a long-immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction). See subsections 10.4.1 and 10.4.2.

Example

```
br.a   Path6     ; unconditional annul branch to Path6,
add    r7,r8,r9  ; this add is not executed in the branch delay slot;
                 ; instead, a nop is executed before the first
                 ; instruction at Path6

br     Path7     ; branch to Path7 and also perform an unsigned add
addu   r7,r8,r9  ; in the branch delay slot before the first instruction
                 ; at Path7
. . .
Path6: . . .
. . .
Path7: . . .
```

Syntax **brcr** *control*

Operation

control & 0x0000 0001 → ie
control & 0xFFFF FFFC → PC
control & 0x0000 0002 → user/supervisor mode latch

Operands

I	Immediate Form
R	Register Form
L	Long-Immediate Form

Encoding

Note: CRN = Control register number

Description

The LSB of the data in the *control* register is copied into IE⟦ie⟧. Bit 1 of the *control* register is copied to the user/supervisor mode latch (which is not visible to you) in order to select the write-access mode; bit 1 = 0 selects the supervisor mode, and bit 1 = 1 selects the user mode. The top data bits in *control* are copied into the program counter (PC). The instruction following the brcr instruction is executed. This results in a delayed branch to the address specified by the data in the *control* register. In the register form, the indirect control register number is held in the *control* register.

Legal control registers that you can use in supervisor mode are: EIP, EPC, MIP, MPC, SYSSTK, and SYSTMP. Control registers IN0P, IN1P, and OUTP are acceptable in supervisor or user mode but are not recommended because these registers contain addresses of floating-point data instead of MP instructions; therefore, the MP operation is unpredictable. The behavior when any other control register or reserved address is used is not predictable (see Table 2–1, *MP Control Register Numbers*, for a list of control registers).

In user mode, if the control register number that you specify with the brcr instruction is less than 0x4000, the brcr instruction is treated as a nop instruction, and you do not receive an error signal.

If you use an undefined control register number, the operation is undefined, and you do not receive an error signal.

Notes:

1) You can use this instruction to return from exceptions.

2) If a memory-fault interrupt has occurred, you must examine the value of the FLTOP⟦R⟧ bit to determine whether a modified return sequence is required (see Figure 3–8, *Memory Fault Operation Register—FLTOP*, and Section 9.3, *Returning From Interrupts and Traps*).

Example 1

Return from an interrupt; execute instruction at EIP when FLTOP⟦R⟧ = 0. See subsection 9.3.2, *Interrupt Return (R = 1)*, for the FLTOP⟦R⟧ = 1 in the memory-fault interrupt case.

```
brcr   EIP      ; execute one instruction at EIP and then
brcr   EPC      ; branch to EPC
```

Example 2

Return from a trap instruction or a floating-point interrupt when the MP is in the sequential mode (FPST⟦sm⟧ = 1); skip over the trap instruction at EIP

```
brcr   EPC      ; branch to EPC
nop             ; delay slot instruction of branch control register
                ; (brcr)
```

Syntax **bsr**[**.a**] *target,link*

Operation Return address → *link*
Relative branch to *target*

Operands

I,R	Immediate Form
R,R	Register Form
L,R	Long-Immediate Form

Encoding

Note: A = Annul the next instruction if branch is taken.

Description The return address is copied into the *link* register. A relative branch to label *target* is taken (IP + 4*offset → PC). In the register form, the indirect signed offset is held in the *target* register as the number of 32-bit instruction words. There are two forms of this instruction:

- **bsr**—The 32-bit instruction following the bsr instruction (a branch delay slot) is executed; then the relative branch is taken. The value that is saved in the *link* register is the address of the second instruction after the bsr instruction.
- **bsr.a**—The instruction following the bsr.a is not executed before the relative branch is taken (a nop cycle is executed instead). In this case, the *link* register is loaded with the address of the first instruction following the bsr.a instruction.

If *target* is a label, the assembler computes the offset value between label *target* and the location of this instruction.

Notes:

1) By software convention, *link* is r31.
2) For all nonannulled branches, you cannot use a long immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction). See subsections 10.4.1 and 10.4.2.

Example

```
bsr    $+0x1234,r31  ; The return address is saved in r31, and IP is
                     ; incremented by 0x1234 bytes; the delay slot
add    0x321,r0,r9   ; add is executed before subroutine entry.
xor    r7,r8,r9      ; Subroutine returns to this xor instruction.

bsr.a  ArcTAN,r31    ; The return address is saved in r31, and the Annul
                     ; delay slot nop instruction is executed before
                     ; branching to the ArcTAN function.
addu   r7,r8,r9      ; ArcTAN function returns to this addu instruction.
. . .
ArcTAN: . . .
```

Note: In the bsr $+0x1234 instruction, the assembler divides 0x1234 by 4 and stores the encoded offset as 0x048D.

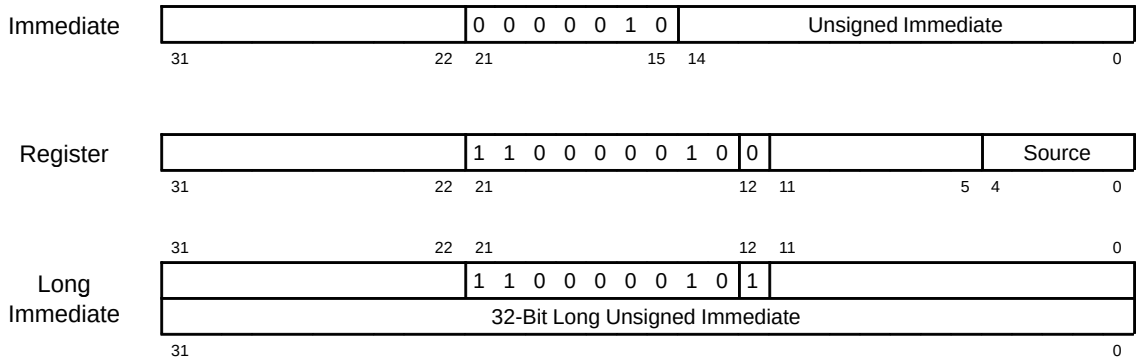
Syntax **cmnd** *source*

Operation INTERRUPT_PROCESSORS(*source*)

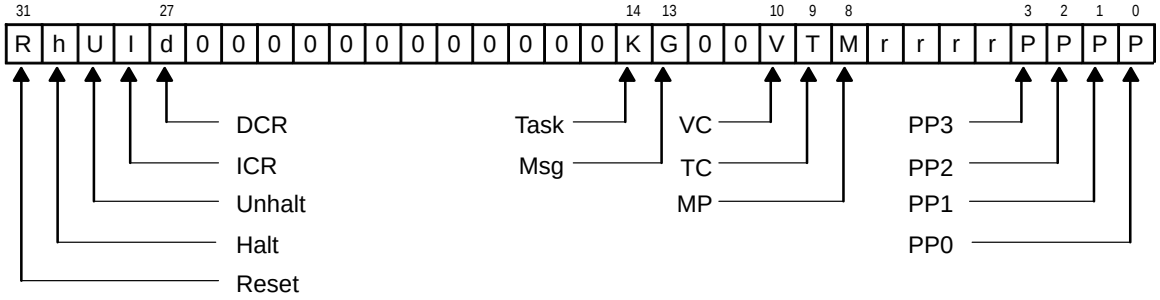
Operands

I	Immediate Form
R	Register Form
L	Long-Immediate Form

Encoding



Description Data from the *source* register (or immediate source data) is used to send commands to the various MVP processors. The bit pattern of the *source* operand determines the command type and the processors involved as shown in the following diagram:



Note: r = reserved
DCR = data-cache reset DTAG registers and DLRU register
ICR = instruction-cache reset ITAG registers and ILRU register

Bits 0–3 and 8–10 select the processors that receive the command. Bits 13–14 and 27–31 determine the commands that are sent. When appropriate, multiple commands can be sent in a single instruction; however, setting both **halt** and **unhalt** bits simultaneously is an example of a combination that should not be used.

All processors do not support all commands. Unsupported commands, if issued, are ignored by the selected processors. The MP does not support **unhalt** of the MP.

- ❑ The MP can issue a **reset**, **halt**, **ICR** (**instruction-cache reset**), **DCR** (**data-cache reset**), or **msg** to itself.
- ❑ The MP can issue a **reset** to the VC or TC.
- ❑ The MP can issue a **reset**, **halt**, **unhalt**, **ICR**, **task**, or **msg** to any PP.
- ❑ The PP can issue a **halt, ICR,** or **msg** to itself.
- ❑ The PP can issue a **msg** to the MP or to another PP.

Notes:

1) Interrupt bits should be enabled so that the target processors (MP or PPs) can service this command request (see Figure 3–2, *MP Interrupt Registers—IE and INTPEN*, and Table 9–1, *Maskable Interrupt Priorities and Vector Addresses*, for the MP).

2) An ICR (instruction-cache reset) resets all instruction tag registers and the ILRU register for the selected processor(s). A DCR (data-cache reset) resets all data tag registers and the DLRU register for the master processor only. An ICR or a DCR also resets cache-resident flags to **not present**, and the cache's LRU (least recently used) is reset (for the MP data cache, dirty flags are reset to 0 for **not modified**). This does **not** perform any write-back of modified source data-cache blocks or subblocks. See Section 11.10, *Clean Data Cache Using the dcachec Instruction*, for an example of saving MP data-cache information.

3) There is one delay slot after the MP cmnd instruction issues an MP self interrupt (INTPEN⟦mi⟧ is set) before the MP self interrupt is signaled, assuming no instruction-cache misses or interrupts are taken.

Example

```
                    ; assume r7 = 0x0000400F
cmnd   r7           ; send TSK interrupt to PP0, PP1, PP2, and PP3
cmnd   0x2001       ; send MSG interrupt to PP0
cmnd   0x400A       ; send TSK interrupt to PP1 and PP3
cmnd   0x8000070F   ; reset VC, TC, MP, PP0, PP1, PP2, and PP3
cmnd   0x10000100   ; Reset MP's instruction cache tags and ILRU
cmnd   0x08000100   ; Reset MP's data cache tags and DLRU
cmnd   0x20000004   ; Unhalt PP2 following a "reset" or resume PP2
                    ;  operation after it has been "halted".
```

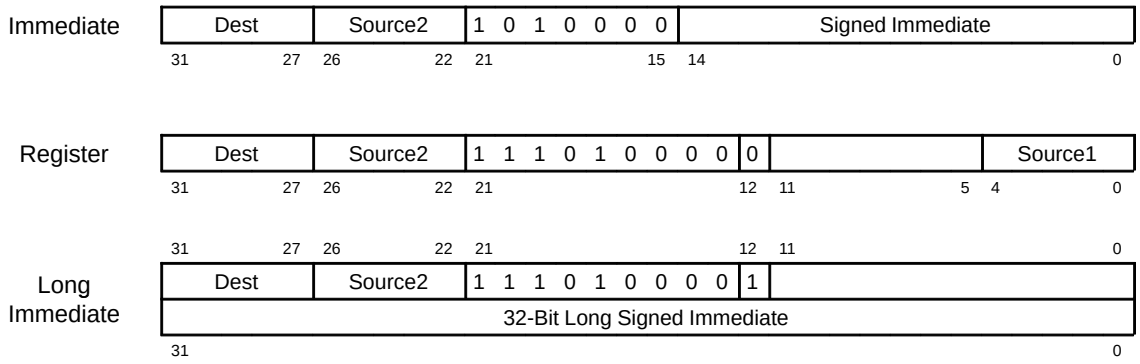
Syntax **cmp** *source1,source2,dest*

Operation status(*source1* – *source2*) → dest

Operands

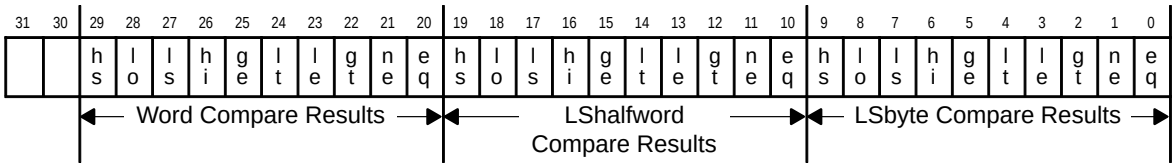
I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data in the *source1* register (or the immediate *source1* data) is compared with data in *source2* register, and various bits are set or cleared in the *dest* register. See Figure 10–2.

Figure 10–2. Bit Fields for the cmp Destination Register



Notes:

1) The compare instruction (cmp) does not alter INTPEN[[io]], even if the subtraction operation overflows.

2) See Table 10–14 for the Boolean definitions of the condition code bits.

Ten condition codes are presented in Table 10–14. 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 bits (negative, carry, overflow, and zero, respectively).

Table 10–14. Boolean Definitions of Condition Codes

Syntax	Description	Status Register Combination
eq	equal (zero)	Z
ne	not equal (not zero)	~Z
gt	greater than	(N&V&~Z)|(~N&~V&~Z)
le	less than or equal	(N&~V)|(~N&V)|Z
lt	less than	(N&~V)|(~N&V)
ge	greater than or equal	(N&V)|(~N&~V)
hi	higher than	C&~Z
ls	lower than or same	~C|Z
lo	lower than (no carry)	~C
hs	higher than or same (carry)	C

Example

```
                          ;       r7             r8             r9
                          ;     before         before          after

cmp     r7,r8,r9          ;    12345678       54545478       19A99AA9
bbo.a   PathA,r9,lt.w     ; branch to PathA as bit 24 (lt.w) = 1
cmp     0xd3c4,r8,r9      ;        –           54545478       19A96A5A
bbo.a   PathB,r9,ne.h     ; branch to PathB as bit 11 (ne.h) = 1
cmp     0x54545478,r7,r9  ;    12345678           —          26666AA9
bbo.a   PathC,r9,eq.b     ; branch to PathC as bit 0 (eq.b) = 1
cmp     0x13C4,r8,r9      ;        —           54545478       19A66A5A
bbo.a   PathD,r9,lo.h     ; branch to PathD as bit 18 (lo.h) = 1
. . .
PathA: . . .
. . .
PathB: . . .
. . .
PathC: . . .
. . .
PathD: . . .
```

Syntax **dcache**{**c**|**f**} *source1*(*source2*[**:m**])

Operation DATA_FLUSH(*source1* + *source2*)

IF (*source2***:m**), then *source1* + *source2* → *source2*

Operands

I(R)	Immediate Form
R(R)	Register Form
L(R)	Long-Immediate Form

Encoding

Note: F = Clear the present flag, as well as the dirty flag.
M = Modify *source2* after data-cache subblock flush has been initiated.

Description If the data at the memory location formed by adding the operands from the *source1* register (or the immediate data) and *source2* register is resident in the data cache and its subblock dirty flag (D bit) is 1, that data cache subblock is written back to memory, and the corresponding dirty flag cleared to 0 (not modified). In the register form, the indirect signed offset is held in *source1* register. The instruction has two forms:

- **dcachef**—clears the subblock present flag corresponding to the subblock that was flushed (P bit in DTAG*n* register, where $0 \leq n \leq 15$).
- **dcachec**—leaves the subblock present flag set, marking the data that was flushed as still resident in the data cache.

The data in the *source2* register can be modified when you specify **:m**. The sum of the operands in registers *source1* (or the immediate data) and *source2* is written back into the *source2* register after the data-cache subblock flush has been initiated.

Example

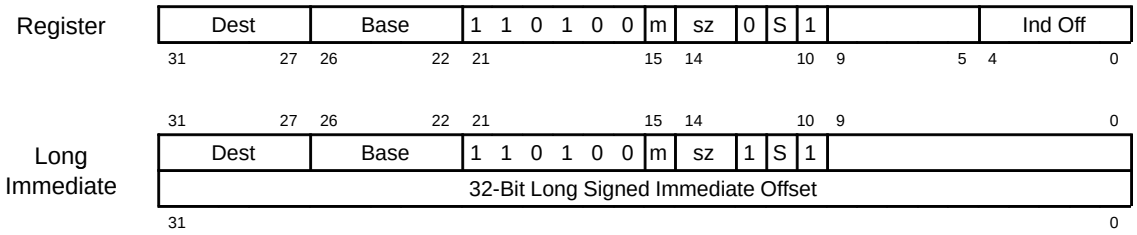
```
                          ;       r7            r8            r8
                          ;     before        before        after

dcachec  r7(r8)           ;    0000000C       12345670
dcachec  0x0004(r8)       ;       –           12345670
dcachef  r7(r8:m)         ;    0000000C       12345670      1234567C
dcachef  0x400(r8:m)      ;       –           12345674      12345A7C

dcachef  0x12345678(r0)   ; absolute address = 0x12345678
```

Syntax **dld**[{**.b**|**.h**|**.d**}] *offset*[**:s**](*base*[**:m**]),*dest*

Operation
IF *offset*, then form (*offset* + *base*) → *addr*
IF *offset***:s**, then scale by data size as (*offset*<<*sz*) + *base* → *addr*
IF *base***:m**, then update *base* as *addr* → *base*
SIGN_EXTEND(MEM(*addr*)) → *dest*

Operands
R(R),R Register Form
L(R),R Long-Immediate Form

Encoding



Notes: 1) m = Modify base address; S = scale offset by data size; sz = data size.
2) sz can have a value of 0, 1, 2, or 3. 0 indicates an 8-bit byte, 1 indicates a 16-bit halfword, 2 indicates a 32-bit word (default), and 3 indicates a 64-bit doubleword.

Description If you specify offset scaling by using the **:s** suffix, then the data in the indirect *offset* register (or long immediate data) is shifted left 0, 1, 2, or 3 for byte, halfword, word, or doubleword sizes, respectively. The word size is the default, and you can specify a byte, halfword, or doubleword by using **.b**, **.h**, or **.d** in the command syntax. See Table 10–15.

Table 10–15. Data Size and Offset Values for dld Instruction

Opcode Syntax	Data Size	offset:s left shift	sz
dld.b	8-bit least significant byte	0	0
dld.h	16-bit least significant halfword	1	1
dld	32-bit word	2	2
dld.d	64-bit doubleword	3	3

The optionally scaled offset is then added to data in the *base* register to form direct memory address *addr*. If you specify modify base address with the **:m** suffix, then the sum *addr* is written back into *base* register as the memory access is issued.

Bypassing the data cache, the data at the direct memory address *addr* is loaded into the least significant part of the *dest* register with left sign extension, if byte or halfword. If the data is a 64-bit doubleword, then the data is loaded into an (even,odd) register pair. The LSword is loaded into the even *dest* register, and the MSword is loaded into the odd *dest*+1 register. See subsection 10.9.2 for additional information about using r0 and r1.

Memory is accessed according to the data size alignment. Halfwords ignore memory address LSB, words ignore two LSBs, doublewords ignore three LSBs.

When you use the dld instruction, memory is accessed **directly** (11–18 cycles), rather than via the data cache (one cycle if the data in *addr* is resident in data cache or on-chip SRAM). Moreover, data in memory and its image in data cache may differ. The dld instruction should be used infrequently because of the longer external memory access times. Data cache values are not altered.

For addresses less than 0x0200 0000, the load and direct load instructions operate in the same manner.

Notes:

1) Loads require one extra cycle before data is available for use. Since the *dest* register is scoreboarded, a nop instruction is not needed; however, the program often can place a useful instruction in the scoreboard time slot.

2) Double-precision loads and stores are independent of the external big- or little-endian memory organization. Endian ordering is transparent to the program when using 64-bit loads or stores—the most significant word is always in the odd (even + 1) register, and the least significant word is always in the even register (≥ 2).

3) When accessing TC or VC on-chip registers with a load instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

4) If the *dest* and *base* are the same register, the *base* value will be overwritten when the *dest* register is loaded. For example, assume r13 = 0x10 and memory location 0xC = 8. In this example,

   ```
   dld –4(r13:m),r13 ; destination r13 = MEM(0xC)
   ```

   memory location 0xC (0x10 – 4) is written to r13 (the *dest* register), and the value 8 is written over the modified *base* value (0xC).

Example Assume the following values before each instruction:

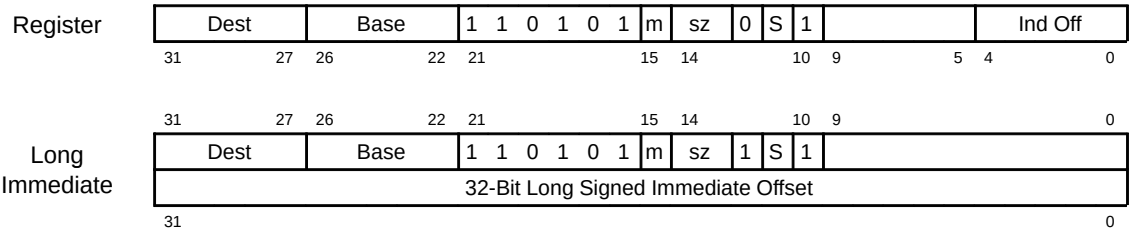
```
                    MEMORY        63                0
                    02000180:     A1340009   74911222
                    02000188:     9001c002   A0031004

                    REGISTERS
                    r5:     02000180
                    r6:     00000004
                    r8:     9ABCDEF0
                    r9:     AAAAAAAA

                      ;            Big-Endian Memory   Little-Endian Memory
                      ;    r5         r8        r9         r8        r9
                      ;   After       After     After      After     After

dld.h   r6(r5),r8          ; 02000180   00007491  AAAAAAAA   00000009  AAAAAAAA
dld     0x2000184(r0),r8   ; 02000180   74911222  AAAAAAAA   A1340009  AAAAAAAA
dld.d   8(r5),r8           ; 02000180   A0031004  9001C002   A0031004  9001C002
dld.b   3(r5:m),r8         ; 02000183   00000009  AAAAAAAA   00000074  AAAAAAAA

dld     r6(r5),r8          ; direct load r8 with a 32-bit word
                           ; one or more cycles scoreboard delay allows dld to
                           ;   complete
add     r8,r9,r10          ; use data from dld

dld     r6(r5),r8          ; direct load r8 with a 32-bit word
xor     0x1234,r9,r9       ; instruction in scoreboard delay slot allows dld to
                           ;   complete if data is in SRAM
add     r8,r9,r10          ; use data from dld

addu    0x2000188,r0,r12   ; r12 = 0x0200 0188
addu    1,r0,r14           ; r14 = 1
dld.h   0x200018A(r0),r11  ; r11 = 0xFFFF C002 (big endian)
                           ; r11 = 0xFFFF A003 (little endian)
dld.h   r14:s(r12),r11     ; r11 is the same as the previous instruction
```

Syntax **dld.u**[{**.b**|**.h**}] *offset*[**:s**](*base*[**:m**]),*dest*

Operation
IF *offset*, then form (*offset* + *base*) → *addr*
IF *offset***:s**, then scale by data size as (*offset*<<*sz*) + *base* → *addr*
IF *base***:m**, then update *base* as *addr* → *base*
ZERO_EXTEND(MEM(*addr*)) → *dest*

Operands
R(R),R Register Form
L(R),R Long-Immediate Form

Encoding



Notes: 1) m = Modify base address; S = scale offset by data size; sz = data size.
2) sz can have a value of 0, 1, 2, or 3. 0 indicates an 8-bit byte, 1 indicates a 16-bit halfword, and 2 and 3 are reserved.

Description

If you specify offset scaling by using the **:s** suffix, then the data in the indirect *offset* register (or long-immediate data) is shifted left 0 or 1 for byte or halfword sizes, respectively. You can specify a byte or halfword by using **.b** or **.h** in the command syntax. See Table 10–16.

Table 10–16. Data Size and Offset Values for dld.u Instruction

Opcode Syntax	Data Size	offset:s left shift	sz
dld.ub	8-bit least significant byte	0	0
dld.uh	16-bit least significant halfword	1	1
	reserved	2	2
	reserved	3	3

The optionally scaled offset is then added to data in register *base* to form direct memory address *addr*. If you specify modify base address with the **:m** suffix, then the sum *addr* is written back into register *base* as the memory access is issued.

Bypassing the data cache, the data at direct memory address *addr* is loaded into the least significant part of the *dest* register with left zero extension for byte or halfword.

Memory is accessed according to the data size alignment; for example, halfwords ignore memory address LSB.

When you use the dld.u instruction, memory is accessed **directly** (11–18 cycles), rather than via the data cache (one cycle if the data in *addr* is resident in data cache or on-chip SRAM). Moreover, data in memory and its image in data cache may differ. The dld.u instruction should be used infrequently because of the longer external memory access times. Data cache values are not altered.

For addresses less than 0x0200 0000, the load and direct load instructions operate in the same manner.

Notes:

1) Loads require one extra cycle before data is available for use. Since the *dest* register is scoreboarded, a nop instruction is not needed; however, the program often can place a useful instruction in the scoreboard time slot.

2) If *offset, base*, and *dest* are not available, execution stalls until they are all available.

3) When accessing TC or VC on-chip registers with a load instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

4) If the *dest* and *base* are the same register, the *base* value will be overwritten when the *dest* register is loaded. For example, assume r13 = 0x10 and memory location 0xC = 8. In this example,

```
dld.uh –4(r13:m),r13 ; destination r13 = MEM(0xC)
```

memory location 0xC (0x10 – 4) is written to r13 (the *dest* register), and the value 8 is written over the modified *base* value (0xC).

5) See subsection 10.9.2 for additional information about using r0 and r1.

Example Assume the following values before each instruction:

```
MEMORY        63                0
02000180:     A1340009  74911222
02000188:     9001c002  A0031004

REGISTERS
r5:     02000180
r6:     00000004
r8:     9ABCDEF0
r9:     AAAAAAAA
```

```
                        ;            Big-Endian Memory   Little-Endian Memory
                        ;    r5          r8       r9          r8       r9
                        ;  After       After    After       After    After

dld.uh  r6(r5),r8       ; 02000180    00007491 AAAAAAAA    00000009 AAAAAAAA
dld.ub  0x2000184(r0),r8; 02000180    00000074 AAAAAAAA    00000009 AAAAAAAA
dld.uh  8(r5),r8        ; 02000180    00009001 AAAAAAAA    00001004 AAAAAAAA
dld.ub  3(r5:m),r8      ; 02000183    00000009 AAAAAAAA    00000074 AAAAAAAA

dld.uh  r6(r5),r8       ; direct load 32-bit r8's LShalfword
                        ; one or more cycles scoreboard delay allows dld to
                        ;   complete
addu    r8,r9,r10       ; use data from dld instruction

dld.uh  r6(r5),r8       ; direct load 32-bit r8's LShalfword
xor     0x1234,r9,r9    ; instruction in scoreboard delay slot if data is in SRAM

addu    r8,r9,r10       ; use data from dld

addu    1,r0,r14        ; r14 = 1
addu    0x2000188,r0,r12; r12 = 0x0200 0188
dld.uh  0x200018A(r0),r11; r11 = 0x0000 C002 (big endian)
                        ; r11 = 0x0000 A003 (little endian)
dld.uh  r14:s(r12),r11  ; r11 is the same as the previous instruction
```

Syntax **dst**[{**.b**|**.h**|**.d**}] *offset*[**:s**] (*base*[**:m**]),*source*

Operation
IF *offset*, then form (*offset* + *base*) → *addr*
IF *offset***:s**, then scale by data size as (*offset*<<*sz*) + *base* → *addr*
IF *base***:m**, then update *base* as addr → *base*
source → MEM(*addr*)

Operands
R(R),R Register Form
L(R),R Long-Immediate Form

Encoding

Notes: 1) m = Modify base address; S = scale offset by data size; sz = data size.

2) sz can have a value of 0, 1, 2, or 3. 0 indicates an 8-bit byte, 1 indicates a 16-bit halfword, 2 indicates a 32-bit word (default), and 3 indicates a 64-bit doubleword.

Description If you specify offset scaling by using the **:s** suffix, then the data in the indirect *offset* register (or long immediate data) is shifted left 0, 1, 2, or 3 for byte, halfword, word, or doubleword sizes, respectively. The word size is the default, and you can specify a byte, halfword, or doubleword by using **.b**, **.h**, or **.d** in the command syntax. See Table 10–17.

Table 10–17. Data Size and Offset Values for dst Instruction

Opcode Syntax	Data Size	offset:s left shift	sz
dst.b	8-bit least significant byte	0	0
dst.h	16-bit least significant halfword	1	1
dst	32-bit word	2	2
dst.d	64-bit doubleword	3	3

The optionally scaled offset is then added to the data in the *base* register to form a direct memory address *addr*. If you specify modify base address with the **:m** suffix, then the sum *addr* is written back into the *base* register as the memory access is issued.

Bypassing the data cache, the data from the least significant part (for byte or halfword) of the *source* register is written into direct memory address *addr*. If the data is a 64-bit doubleword, then the data originates from an (even,odd) register pair. The LSword is read from the even *source* register, and the MSword is read from the odd *source*+1 register. See subsection 10.9.2 for additional information about using r0 and r1.

Memory is accessed according to the data size alignment. Half-words ignore memory address LSB, words ignore two LSBs, doubleword ignore three LSBs.

When you use the dst instruction, memory is accessed **directly** (8–12 cycles), rather than via the data cache (one cycle if the data in *addr* is resident in data cache or on-chip SRAM). Moreover, data in memory and its image in data cache may differ. For cache coherency, use a program similar to the program shown in Example 11–10, if necessary. The dst instruction should be used infrequently because of the longer external memory access times. Data cache values are not altered.

For addresses less than 0x0200 0000, the store and direct store instructions operate in the same manner.

Notes:

1) Double-precision loads and stores are independent of the external big- or little-endian memory organization. Endian ordering is transparent to the program when using 64-bit loads or stores—the most significant word is always in the odd (even + 1) register, and the least significant word is always in the even register (≥ 2).

2) If *offset, base,* and *source* registers are not available, execution stalls until they are available.

3) When accessing TC or VC on-chip registers with a store instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

4) If the *source* and *base* are the same register, the original source value is written into the preincremented memory location. For example, assume r13 = 8, then

```
dst 4(r13:m),r13  ; MEM(0xC) = 8 (original source)
```

stores 8 (the original *source* value) into memory location 0xC (4 + 8), and 0xC is written back as the modified *base* value.

Example 1 Memory and registers before instructions:

```
020000C0:   00000000   00000000          r7     00000006
020000C8:   00000000   00000000          r8     12345678
020000D0:   00000000   00000000          r9     9ABCDEF0
020000D8:   00000000   00000000         r10     020000C0


                                         ;     r10 after
dst.d   16(r10),r8                       ;      020000C0
dst     8(r10),r9                        ;      020000C0
dst.h   r7(r10),r8                       ;      020000C0
dst.b   3(r10:m),r8                      ;      020000C3
```

Memory after instructions:

```
            Big Endian    Little Endian
020000C0:   00005678      78000000
020000C4:   00000078      56780000
020000C8:   00000000      9ABCDEF0
020000CC:   9ABCDEF0      00000000
020000D0:   12345678      12345678
020000D4:   9ABCDEF0      9ABCDEF0

            31     0      31     0 (bit)
```

Example 2

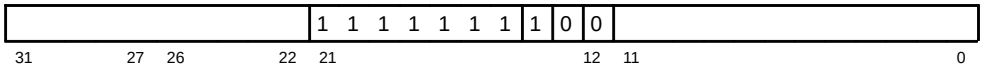
```
addu    1,r0,r14            ; r14 = 1
addu    0x20000c8,r0,r11    ; r12 = 0x0200 00C8
addu    0xc002,r0,r20       ; r20 = 0xC002
dst.h   0x20000ca(r0),r20   ; 0x20000ca = 0x9ABC C002 (big endian)
                            ; 0x20000ca = 0xC002 DEF0 (little endian)
dst.h   r14:s(r11),r20      ; same result as the previous instruction
```

Syntax **estop**

Operation An emulation stop instruction is issued

Operands None

Encoding



Description This instruction is reserved for emulation.

> **Note:**
>
> Do **not** use the estop instruction in your program.

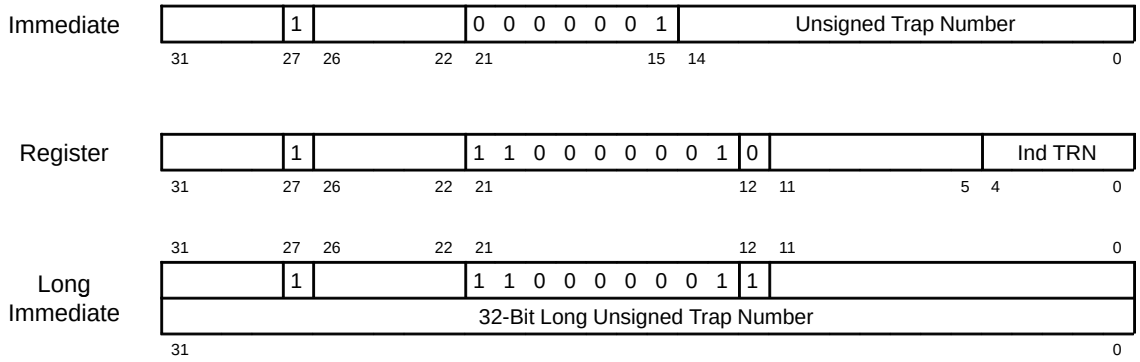
Syntax **etrap** *trapnumber*

Operation MEM(0x0101 0180 + 4**trapnumber*) → PC
and disable analysis block

Operands

I	Immediate Form
R	Register Form
L	Long-Immediate Form

Encoding



Description This instruction is reserved for emulation.

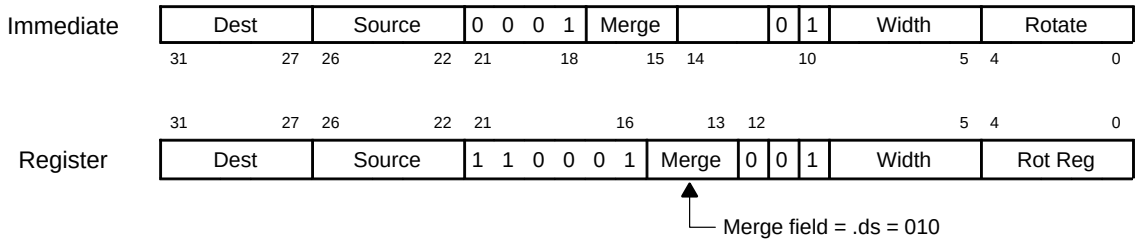
> **Note:**
>
> Do **not** use the etrap instruction in your program.

Syntax **exts** *rotate,width,source,dest*

Operation SIGN_EXTEND((*source* // *rotate*) & *endmask*) → *dest*

Operands
I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding



Description The *width* low-order bits of the rotated data from the *source* register are written into the *dest* register using left sign extension. The assembler uses the **sr.ds** instruction for the exts mnemonic:

- The shiftmask option uses disabled (d) (see **sr.**{**d**|**e**|**i**}{...})
- The endmask option uses sign extend (s) (see **sr.**{...}{**m**|**s**|**z**})

Data in register *source* is rotated right by:

- **immediate form**—the value in the *rotate* field
- **register form**—the five LSBs of data in register *rotate*

The *endmask* is generated by:

- **immediate form**—from the *width* field as shown in Table 10–24, *Mask Values as* $2^{Code}-1$ *in Shift Operations*
- **register form**—from the value in the *width* field:
  - 0, 32: endmask is all 1s
  - 1–31: endmask from five LSBs of odd register *rotate*+1, as shown in Table 10–24. Note that in this case, *rotate* must be an even register; do not supply *rotate* as an odd register.

A bitwise AND of the rotated data in the *source* register and the endmask is written into the *dest* register with left sign extension.

In summary, the extract signed field operation exts is accomplished by

dest = SIGN_EXTEND((*source* // *rotate*) & *endmask*)

Example 1 Extract r7 bits 5–12, write left sign extended into r9 bits 0–7

```
                              ; rotate = 5, width = 8 bits (input bits 5–12)
exts 5, 8, r7, r9             ; r7 input bits 5–12 are rotated right 5 bits
                              ; to form r9 output bits 0–7 with left sign
                              ; bit extension

31                      12       5     0      ; input LSB = bit 5
                          |<------>|           ; input extract field
abcdefghijklmnopqrstuvwxyzABCDEF              ; r7 input register


BCDEFabcdefghijklmnopqrstuvwxyzA              ; r7 // 5 (LSB = bit 0)
                               |<------>|     ; rotate r7 right 5 bits

00000000000000000000000011111111              ; endmask = Table(Width)
                                              ; where Width = 8

tttttttttttttttttttttttuvwxyzA                ; r9 = (r7//r5) & 0xff
                               |<------>|     ; 8-bit signed output with
31                              7       0     ; left sign bit extension
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 5 (rotate amount)
- r5 = 8 (width field value)
- r7 = 0x1020 3040 (original value)
- r9 = 0x0F0E 0D0C (original value)

Either of these instructions returns r9 = 0xFFFF FF82:

```
exts  5,8,r7,r9   ; Immediate form
```

or

```
exts  r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0x0000 0068:

```
exts  5,8,r9,r7   ; Immediate form
```

or

```
exts  r4,31,r9,r7 ; Register form
```

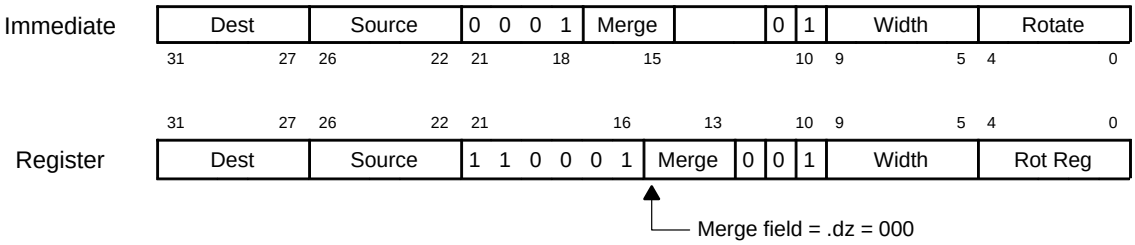
Syntax **extu** *rotate,width,source,dest*

Operation ((*source* // *rotate*) & *endmask*) → *dest*

Operands

I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding



Description

The *width* low-order bits of the rotated data from the *source* register are written into the *dest* register. The assembler uses the **sr.dz** instruction for the extract unsigned field extu mnemonic:

- The shiftmask option uses disabled (d) (see **sr.**{**d**|**e**|**i**}{...})
- The endmask option uses zero extend (z) (see **sr.**{...}{**m**|**s**|**z**})

Data in register *source* is rotated right by:

- **immediate form**—the value in the *rotate* field
- **register form**—the five LSBs of data in register *rotate*

The *endmask* is generated by:

- **immediate form**—from the *width* field as shown in Table 10–24, *Mask Values as* $2^{Code}-1$ *in Shift Operations*
- **register form**—from the value in the *width* field:
  - 0, 32: endmask is all 1s
  - 1–31: endmask from five LSBs of odd register *rotate*+1 as shown in Table 10–24. Note that in this case, *rotate* must be an even register; do not supply *rotate* as an odd register.

A bitwise AND of the rotated data in register *source* and the endmask is written into the *dest* register with left zero extension.

In summary, the extract unsigned field operation (extu) is accomplished by

dest = ((*source* // *rotate*) & *endmask*)

Example 1 Extract r7 bits 5–12, write left extended into r9 bits 0–7

```
                          ; rotate = 5, width = 8 bits (input bits 5–12)
extu 5, 8, r7, r9         ; r7 input bits 5–12 are rotated right 5 bits
                          ; to form r9 output bits 0–7 with left zero
                          ; extension

31                   12       5     0     ; input LSB = bit 5
                      |<------>|           ; input extract field
abcdefghijklmnopqrstuvwxyzABCDEF     ; r7 input register


BCDEFabcdefghijklmnopqrstuvwxyzA     ; r7 // 5 (LSB = 0)
                              |<---->|     ; rotate r7 right 5 bits

00000000000000000000000011111111     ; endmask = Table(Width)
                                     ; where Width = 8

00000000000000000000000tuvwxyzA      ; r9 = (r7//r5) & 0xff
                              |<---->|     ; 8-bit unsigned output
31                            7       0     ; left zero extension
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 5 (rotate amount)
- r5 = 8 (width field value)
- r7 = 0x1020 3040 (original value)
- r9 = 0x0F0E 0D0C (original value)

Either of these instructions returns r9 = 0x0000 0082:

```
extu  5,8,r7,r9   ; Immediate form
```

or

```
extu  r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0x0000 0068:

```
extu  5,8,r9,r7   ; Immediate form
```

or

```
extu  r4,31,r9,r7 ; Register form
```

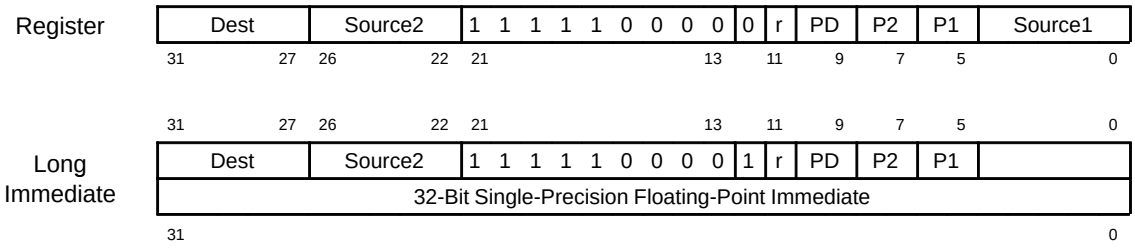
Syntax **fadd.**{**s**|**d**}{**s**|**d**}{**s**|**d**} *source1,source2,dest*

Operation *source1* + *source2* → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source1* precision; P2 = *source2* precision; PD = *dest* precision; r = reserved.

2) P1, P2, and PD can each have a value of 0, 1, 2, or 3. For the fadd instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, and 2 and 3 are reserved.

Description The fadd instruction performs floating-point addition on the operands from the *source1* register (or immediate data) and the *source2* register and writes the result into the *dest* register (using the default rounding mode, set with the drm field of the FPST register).

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the fadd operation.

Precisions The following combinations of precisions are legal:

```
fadd.sss = SP + SP → SP
fadd.ssd = SP + SP → DP
fadd.sdd = SP + DP → DP
fadd.dsd = DP + SP → DP
fadd.ddd = DP + DP → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the fadd instruction).

Example

```
fadd.sss   –0.75, r8, r9     ; –0.75 + r8 → r9 (all SP FP)

fadd.ssd   r6, r8, r9        ; Error: DP FP register must be even

fadd.ssd   r4, r9, r6        ; r4 + r9 = r6 [ convert SP FP input to
                             ;     DP FP and perform DP FP addition,
                             ;     write DP FP result to (r6,r7) ]
```

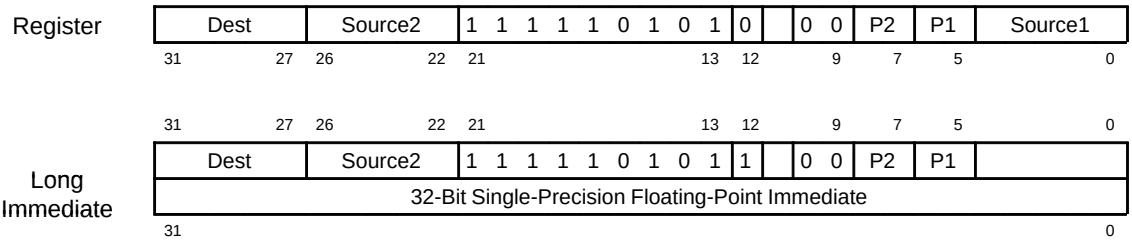
Syntax **fcmp.**{**s**|**d**}{**s**|**d**} *source1,source2,dest*

Operation status(*source1* – *source2*) → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source1* precision; P2 = *source2* precision

2) P1 and P2 can each have a value of 0, 1, 2, or 3. For the fcmp instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, and 2 and 3 are reserved.

Description Operands from the *source1* register (or the immediate data) and the *source2* register are compared using floating-point subtract, and various bits are set or cleared in the *dest* register, as shown in Table 10–18.

The bits in the destination register are shown in Figure 10–3.

Figure 10–3. fcmp Destination Register Bits

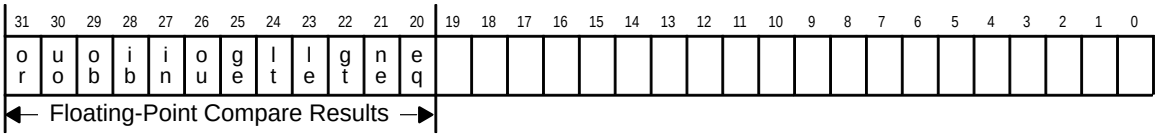


Table 10–18. Condition Codes Generated by the fcmp Instruction

Code	Dest Bit No.	Type of Condition
eq.f	20	*source1* = *source2*
ne.f	21	*source1* ≠ *source2*
gt.f	22	*source1* > *source2*
le.f	23	*source1* ≤ *source2*
lt.f	24	*source1* < *source2*
ge.f	25	*source1* ≥ *source2*
ou.f	26	*source1* < 0, or *source1* > *source2* means *source1* is out of range.
in.f	27	0 < *source1* < *source2* means *source1* is in range.
ib.f	28	0 ≤ *source1* ≤ *source2* means *source1* is in range or on boundary
ob.f	29	*source1* ≤ 0, or *source1* ≥ *source2* means *source1* is out of range or on boundary.
uo.f	30	Either *source1* or *source2* unordered. Unordered means floating-point value is QNaN (quiet not-a-number) or SNaN (signaling not-a-number).
or.f	31	Both *source1* and *source2* ordered. Ordered means floating-point value is not QNaN or SNaN.

Latency

Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the fcmp operation.

Precisions

The following combinations of precisions are legal:

```
fcmp.ss = SP – SP → status
fcmp.sd = SP – DP → status
fcmp.ds = DP – SP → status
fcmp.dd = DP – DP → status
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Notes:

1) Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the fcmp instruction).

2) If uo.f (bit 30) = 1, then all other ordering bits = 0.

3) If *source2* < 0, all of the range bits = 0.

Example

```
                          ; r7 = 2.375, r8 = 0xA5A0 0000
fcmp.ss   –0.75,r7,r8     ; STATUS(–0.75 – r7) → r8 from SP FP subtract
bbo.a     PathE,r8,ne.f   ; Branch to PathE as bit 21 (ne.f) = 1
. . .
                          ; r8 = double-precision floating-point 4.0
fcmp.sd   r7,r8,r6        ; Convert SP FP r7 to internal DP value
                          ; STATUS(r7 – 4.0) → r6 from DP FP subtract =
                          ; 0x99A0 0000
bbo.a     PathF,r6,lt.f   ; Branch to PathF as bit 24 (lt.f) = 1
fcmp.ds   r8,r7,r6        ; r8 = 4.0, r6 = 0xA660 0000
bbo.a     PathG,r6,ge.f   ; Branch to PathG as bit 25 (ge.f) = 1
. . .
PathE: . . .
. . .
PathF: . . .
. . .
PathG: . . .
```

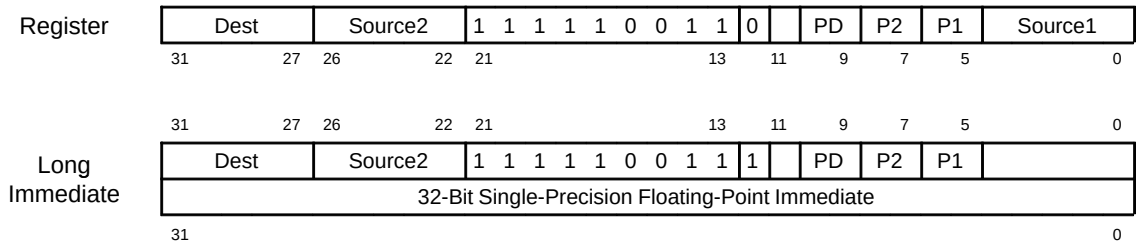
Syntax **fdiv.**{**s**|**d**}{**s**|**d**}{**s**|**d**} *source1,source2,dest*

Operation *source1* ÷ *source2* → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source1* precision; P2 = *source2* precision; PD = *dest* precision.

2) P1, P2, and PD can each have a value of 0, 1, 2, or 3. For the fdiv instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, and 2 and 3 are reserved.

Description The fdiv instruction performs a floating-point division on data from the *source1* register (or the immediate data) by data in the *source2* register and writes the result (using the default rounding mode, set with the drm field of the FPST register) into the *dest* register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the fdiv operation.

Precisions The following combinations of precisions are legal:

```
fdiv.sss = SP ÷ SP → SP
fdiv.ssd = SP ÷ SP → DP
fdiv.sdd = SP ÷ DP → DP
fdiv.dsd = DP ÷ SP → DP
fdiv.ddd = DP ÷ DP → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the fdiv instruction).

Example

```
fdiv.sss   -0.75,r8,r9       ; -0.75 / r8 → r9 (all is SP FP)

fdiv.ssd   r6,r8,r9          ; Error: DP FP register must be even

fdiv.ssd   r8,r9,r6          ; Converts SP r8 and r9 to internal DP FP,
                             ;     form r8 / r9 using DP FP division
                             ;     and write DP FP (r6,r7)
```

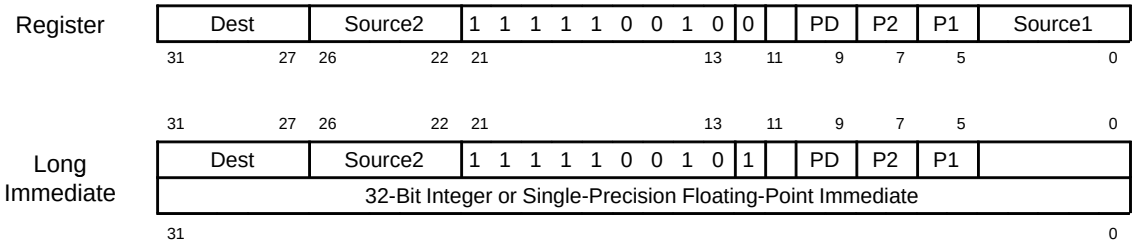
Syntax **fmpy.{s|d|i|u}{s|d|i|u}{s|d|i|u}** *source1,source2,dest*

Operation *source1* × *source2* → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source1* precision; P2 = *source2* precision; PD = *dest* precision.

2) P1, P2, and PD can each have a value of 0, 1, 2, or 3. For the fmpy instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, 2 indicates a 32-bit signed integer, and 3 indicates a 32-bit unsigned integer.

Description The fmpy instruction performs an integer or a floating-point multiplication on the operands from the *source1* register (or the immediate data) and the *source2* register and writes the result (using the default rounding mode, set with the drm field of the FPST register) into the *dest* register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the fmpy operation.

Precisions The following combinations of precisions are legal:

```
fmpy.sss = SP × SP → SP
fmpy.ssd = SP × SP → DP
fmpy.sdd = SP × DP → DP
fmpy.dsd = DP × SP → DP
fmpy.ddd = DP × DP → DP
fmpy.iii = I × I → I          32 LSBs†
fmpy.uuu = U × U → U          32 LSBs†
```

† Sets FPST⟦mo⟧ to 1 if the product exceeds 32 bits. No other floating-point exceptions are signaled.

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the fmpy instruction).

Example

```
fmpy.sss   –0.75,r8,r9     ; –0.75 * r8 → r9 (all in SP FP)

fmpy.ssd   r6,r8,r9        ; Error: DP FP register must be even

fmpy.ssd   r8,r9,r6        ; converts SP FP r8 and r9 to internal DP FP,
                           ;     form r8 * r9 using DP FP multiply and
                           ;     write DP FP (r6,r7)

fmpy.uuu   423,r8,r9       ; r9 = 32 LSBs of the unsigned integer
                           ;     product 423 * r8

fmpy.iii   r7,r8,r9        ; r9 = 32 LSBs of the signed integer
                           ;     product r7 * r8
```

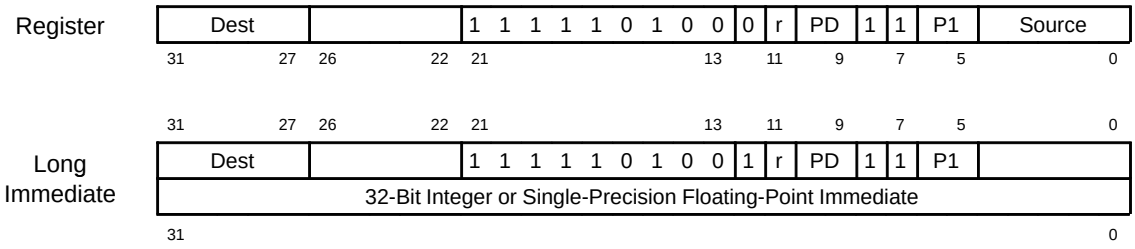
Syntax **frndm.{s|d|i|u}{s|d|i|u}** *source,dest*

Operation round_minus(*source*) → *dest*

Operands

R,R	Register Form
L,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; r = reserved.

2) P1 and PD can each have a value of 0, 1, 2, or 3. For the frndm instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, 2 indicates a 32-bit signed integer, and 3 indicates a 32-bit unsigned integer.

Description The operand in the *source* register (or the immediate data) is converted from the source precision to the destination precision. The result is rounded toward minus infinity and written into the *dest* register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the frndm operation.

Precisions The following combinations of precisions are legal:

```
frndm.ss = SP → SP          frndm.ds = DP → SP
frndm.sd = SP → DP          frndm.dd = DP → DP
frndm.si = SP → I           frndm.di = DP → I
frndm.su = SP → U           frndm.du = DP → U
frndm.is = I → SP           frndm.us = U → SP
frndm.id = I → DP           frndm.ud = U → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the frndm instruction).

Example

```
frndm.ds   r6,r9          ; Convert DP FP (r6,r7) to SP FP r9 after
                          ;  rounding to minus infinity

frndm.id   r7,r6          ; Convert signed integer r7 to DP FP (r6,r7)
                          ;  after rounding to minus infinity

frndm.us   r7,r6          ; Convert unsigned integer r7 to SP FP r6
                          ;  after rounding to minus infinity
```

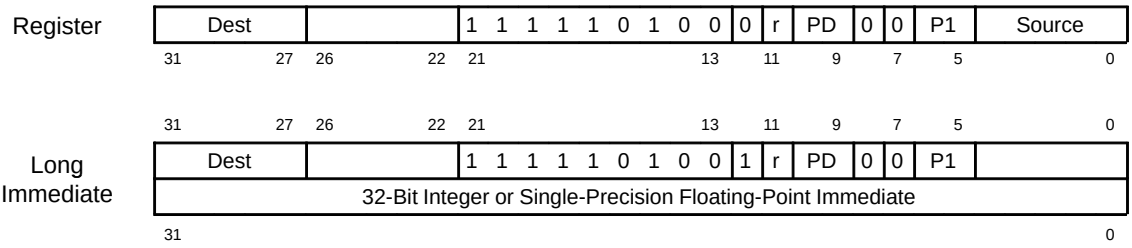
Syntax **frndn.**{**s**|**d**|**i**|**u**}{**s**|**d**|**i**|**u**} *source,dest*

Operation round_nearest(*source*) → *dest*

Operands

R,R	Register Form
L,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; r = reserved.

2) P1 and PD can each have a value of 0, 1, 2, or 3. For the frndn instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, 2 indicates a 32-bit signed integer, and 3 indicates a 32-bit unsigned integer.

Description The operand in the *source* register (or the immediate data) is converted from the source precision to the destination precision. The result is rounded to nearest and written into the *dest* register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the frndn operation.

Precisions The following combinations of precisions are legal:

```
frndn.ss = SP → SP          frndn.ds = DP → SP
frndn.sd = SP → DP          frndn.dd = DP → DP
frndn.si = SP → I           frndn.di = DP → I
frndn.su = SP → U           frndn.du = DP → U
frndn.is = I → SP           frndn.us = U → SP
frndn.id = I → DP           frndn.ud = U → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the frndn instruction).

Example

```
frndn.ds    r6,r9           ; Convert DP FP (r6,r7) to SP FP r9 after
                            ;  rounding to nearest number

frndn.id    r7,r6           ; Convert signed integer r7 to DP FP (r6,r7)
                            ;  after rounding to nearest number

frndn.us    r7,r6           ; Convert unsigned integer r7 to SP FP r6
                            ;  after rounding to nearest number
```

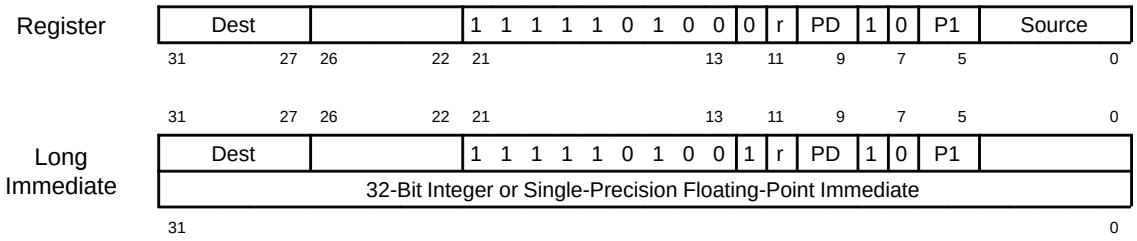
Syntax **frndp.**{**s**|**d**|**i**|**u**}{**s**|**d**|**i**|**u**} *source,dest*

Operation round_positive(*source*) → *dest*

Operands

R,R	Register Form
L,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; r = reserved.

2) P1 and PD can each have a value of 0, 1, 2, or 3. For the frndp instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, 2 indicates a 32-bit signed integer, and 3 indicates a 32-bit unsigned integer.

Description The operand in the *source* register (or the immediate data) is converted from the source precision to the destination precision. The result is rounded toward positive infinity and written into the *dest* register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the frndp operation.

Precisions The following combinations of precisions are legal:

```
frndp.ss = SP → SP          frndp.ds = DP → SP
frndp.sd = SP → DP          frndp.dd = DP → DP
frndp.si = SP → I           frndp.di = DP → I
frndp.su = SP → U           frndp.du = DP → U
frndp.is = I → SP           frndp.us = U → SP
frndp.id = I → DP           frndp.ud = U → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

> **Note:**
>
> Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the frndp instruction).

Example

```
frndp.ds   r6,r9          ; Convert DP FP (r6,r7) to SP FP r9 after
                          ;  rounding to positive infinity

frndp.id   r7,r6          ; Convert signed integer r7 to DP FP (r6,r7)
                          ;  after rounding to positive infinity

frndp.us   r7,r6          ; Convert unsigned integer r7 to SP FP r6
                          ;  after rounding to positive infinity
```

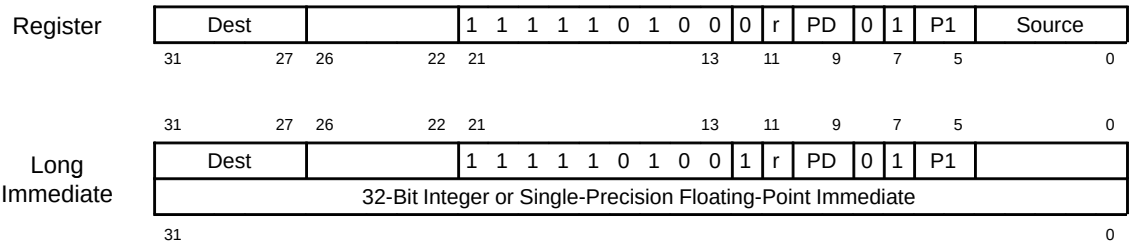
Syntax **frndz.**{**s**|**d**|**i**|**u**}{**s**|**d**|**i**|**u**} *source,dest*

Operation round_zero(*source*) → *dest*

Operands

R,R	Register Form
L,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; r = reserved

2) P1 and PD can each have a value of 0, 1, 2, or 3. For the frndz instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, 2 indicates a 32-bit signed integer, and 3 indicates a 32-bit unsigned integer.

Description The operand in the *source* register (or the immediate data) is converted from the source precision to the destination precision. The result is rounded toward zero and written into the *dest* register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the frndz operation.

Precisions The following combinations of precisions are legal:

```
frndz.ss = SP → SP          frndz.ds = DP → SP
frndz.sd = SP → DP          frndz.dd = DP → DP
frndz.si = SP → I           frndz.di = DP → I
frndz.su = SP → U           frndz.du = DP → U
frndz.is = I → SP           frndz.us = U → SP
frndz.id = I → DP           frndz.ud = U → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the frndz instruction).

Example

```
frndz.ds   r6,r9          ; Convert DP FP (r6,r7) to SP FP r9 after
                          ;  rounding to zero

frndz.id   r7,r6          ; Convert signed integer r7 to DP FP (r6,r7)
                          ;  after rounding to zero

frndz.us   r7,r6          ; Convert unsigned integer r7 to SP FP r6
                          ;  after rounding to zero
```

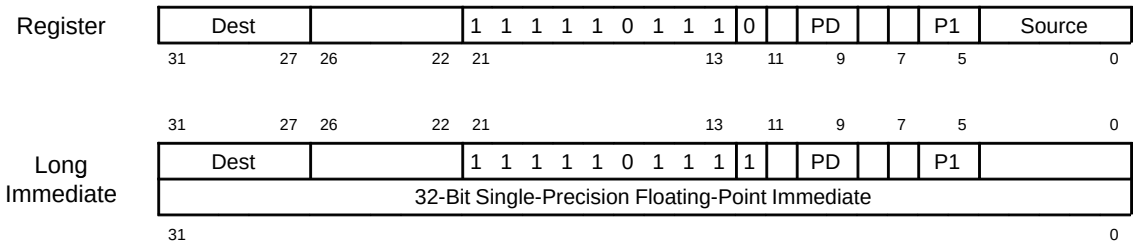
Syntax **fsqrt.**{**s**|**d**}{**s**|**d**} *source,dest*

Operation square_root(*source*) → *dest*

Operands

R,R	Register Form
L,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision.

2) P1 and PD can each have a value of 0, 1, 2, or 3. For the fsqrt instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, and 2 and 3 are reserved.

Description The floating-point square root of the operand in the *source* register (or the immediate data) is calculated. The result (using the default rounding mode, set with the drm field of the FPST register) is written into the *dest* register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the fsqrt operation.

Precisions The following combinations of precisions are legal:

```
fsqrt.ss = √SP → SP
fsqrt.sd = √SP → DP
fsqrt.dd = √DP → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note that fsqrt.ds is **not** supported.

> **Note:**
>
> Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the fsqrt instruction).

Example

```
fsqrt.ss   6.0,r9          ; [all SP FP] square root (6.0) → r9

fsqrt.dd   r6,r8           ; square root of DP FP (r6,r7) is
                           ;  written to DP FP (r8,r9)
```

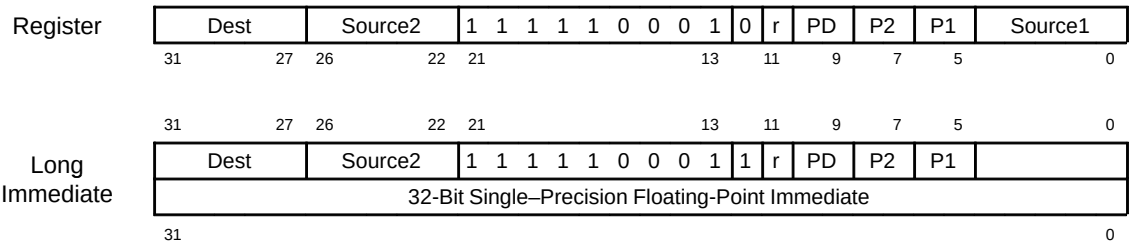
Syntax **fsub.**{**s**|**d**}{**s**|**d**}{**s**|**d**} *source1,source2,dest*

Operation *source1* – *source2* → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source1* precision; P2 = *source2* precision; PD = *dest* precision; r = reserved.

2) P1, P2, and PD can each have a value of 0, 1, 2, or 3. For the fsub instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, and 2 and 3 are reserved.

Description The fsub instruction performs a floating-point subtraction on the data from the *source1* register (or the immediate data) by the data from the *source2* register and writes the result (using the default rounding mode, set with the drm field of the FPST register) into the destination register.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the fsub operation.

Precisions The following combinations of precisions are legal:

```
fsub.sss = SP – SP → SP
fsub.ssd = SP – SP → DP
fsub.sdd = SP – DP → DP
fsub.dsd = DP – SP → DP
fsub.ddd = DP – DP → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the fsub instruction).

Example

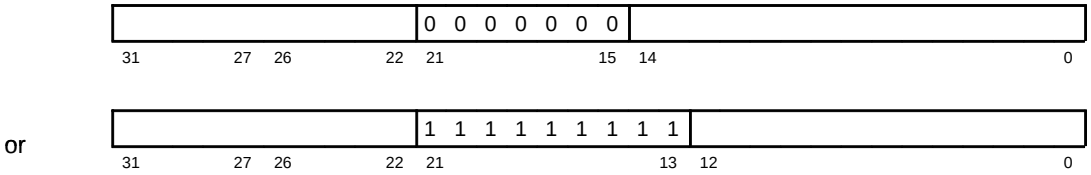
```
fsub.sss   –0.75,r8,r9       ; –0.75 – r8 → r9 (All SP FP)

fsub.ssd   r6,r8,r9          ; Error: DP FP register must be even

fsub.ssd   r9,r8,r6          ; r9 – r8 = r6 [ convert SP FP input to
                             ;  DP FP and perform DP FP subtraction,
                             ;  write DP FP result to (r6,r7) ]
```

Syntax Not applicable

Operation Illegal instruction trap is taken

Operands None

Encoding



Description The opcodes above are defined as illegal instructions. They cause an illegal instruction trap to be taken.

The choice of these two instructions is to make the bit patterns 0x0000 0000 and 0xFFFF FFFF illegal. If external hardware forces either of these values into the bus when an access is made to unimplemented memory, then this programming error can be detected. Note that in the case of 0xFFFF FFFF, a long immediate is required. When this long-immediate data is fetched, one cycle delay occurs before the exception is taken.

Example

```
.word      0              ; data word = 0 is not a legal instruction
.word      –1             ; data word = –1 is an illegal instruction
```

> **Note:**
>
> Either one of these examples produces a trap 38 (illegal instruction) on the MP.

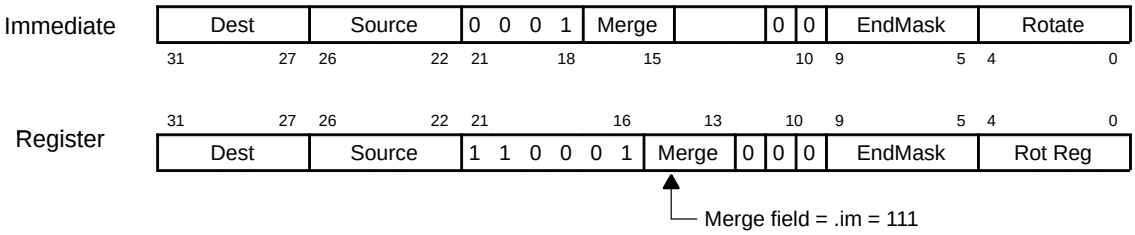
Syntax **ins** *rotate,endmask,source,dest*

where *endmask* = *rotate* amount + *width*

Operation ((*source* \\ *rotate*) & CM) OR (*dest* & ~CM) → *dest*

Operands
I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding



Description The *width* low-order bits of the data from the *source* register are rotated and inserted into the target bits of the *dest* register. Note that *endmask* must be the sum of *rotate* amount and width of the insertion field.

The assembler uses the **sl.im** instruction for the insert field ins mnemonic:

- The shiftmask option uses inverted (i) (see **sl.{d|e|i}**{...})
- The endmask option uses merge fields (m) (see **sl.**{...}{**m|s|z**})

Data in register *source* is rotated right by:

- **immediate form**—the value in the *rotate* field
- **register form**—the five LSBs of data in register *rotate*

The *shiftmask* is generated by:

- **immediate form**—from the *rotate* field as shown in Table 10–24, *Mask Values as* $2^{Code}-1$ *in Shift Operations*
- **register form**—the five LSBs of data in register *rotate* as shown in Table 10–24

The *endmask* is generated by:

- **immediate form**—from the *endmask* field as shown in Table 10–24
- **register form**—from the value in the *endmask* field:
  - 0, 32: endmask is all 1s
  - 1–31: endmask from five LSBs of odd register *rotate*+1, as shown in Table 10–24. Note that in this case, *rotate* must be an even register; do not supply *rotate* as an odd register.

A composite mask CM is formed as the bitwise AND of the inverted (ones complement) shiftmask and the endmask.

CM = (~*shiftmask*) AND *endmask*

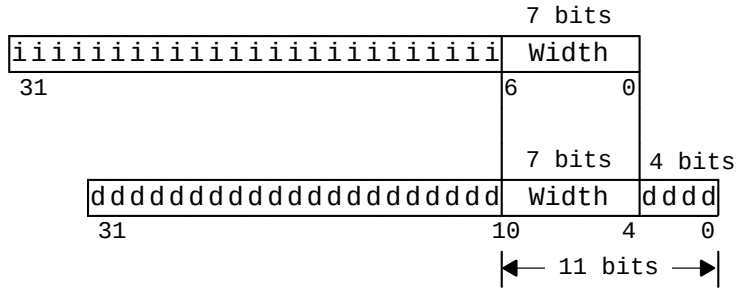
The rotated data in the *source* register and the data from *dest* register are merged using CM with the result written into the *dest* register.

In summary, the insert field operation ins is implemented by

dest = ((*source* \\ *rotate*) & CM) OR (*dest* & ~CM)

Example 1 Insert r7 bits 0–6 into r9 bits 4–10

```
ins        4,11,r7,r9        ; Rotate = 4, Width = 7 (input bits 0–6)
                             ; r7 input bits 0–6 = r9 output bits 4–10
                             ; EndMask = 11 = Rotate (4) + Width (7)
```




```
                                 ; Width = 7 bits
                                 ; Input source
                                 ; Insert field
                                 ; Rotate = 4
                                 ; Destination
                                 ; EndMask = 11 =
                                 ; Rotate + Width

31                  12      6      0   ; LSB = 0
                            |<---->|   ; Width = 7 (input bits 0–6)
abcdefghijklmnopqrstuvwxy9876543     ; r7 input register

efghijklmnopqrstuvwxy9876543abcd     ; r7 \\ 4 (LSB = 4)
                        |<---->|     ; Rotate r7 left 4 bits

00000000000000000000000001111        ; shiftmask = Table(Rotate)
                                     ; where Rotate = 4

11111111111111111111111110000        ; Inverted shiftmask

00000000000000000000011111111111     ; endmask = Table(EndMask)
                                     ; where EndMask = 11 = 4 + 7
                                     ; Rotate = 4, Width = 7

00000000000000000000011111110000     ; CM = ~shiftmask & endmask

ABCDEFGHIJKLMNOPQRSTU.......VWXY     ; r9 input register

ABCDEFGHIJKLMNOPQRSTU9876543VWXY     ; ((r7\\4) & CM) OR (r9 & ~CM)
                        |<---->|     ; = r9 output
31                     10     4  0
```

Example 2

Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 4 (rotate amount)
- r5 = 11 (width field value)
- r7 = 0x123 4567 (original value)
- r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0xFEDC BE78:

```
ins   4,11,r7,r9  ; Immediate form
```

or

```
ins   r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0x0123 4187:

```
ins   4,11,r9,r7  ; Immediate form
```

or

```
ins   r4,31,r9,r7 ; Register form
```

Syntax **jsr**[**.a**] *offset(base),link*

Operation Return address → *link*
absolute *offset* + *base* → PC

Operands
I(R),R Immediate Form
R(R),R Register Form
L(R),R Long-Immediate Form

Encoding

Note: A = Annul the next instruction if branch is taken.

Description The return address is copied into the *link* register. Signed *offset* (or immediate data) is added to the data in the *base* register and written to the PC, performing a branch to an absolute address. Note that the value written into PC must have the two LSBs equal to 0 (word boundary). In the register form, the indirect signed offset is held in the *offset* register. There are two cases of this instruction:

- **jsr**—The 32-bit instruction following the jsr instruction (a branch delay slot) is executed; then the absolute branch is taken. Thus, the value that is saved in the *link* register is the address of the second instruction after the jsr instruction.
- **jsr.a**—The instruction following the jsr.a is not executed before the absolute branch is taken (a nop cycle is executed instead). In this case, the *link* register is loaded with the address of the first instruction following the jsr.a instruction.

jsr[.a] *link* (r0),r0 returns from a subroutine, assuming that *link* has not been altered. By software convention, *link* is r31.

Note:

For all nonannulled branches, you cannot use a long immediate operand or branches in the branch delay slot instruction (the first instruction following the branch instruction). See subsections 10.4.1 and 10.4.2.

Example

```
jsr.a   r31(r0),r0      ; standard subroutine exit with annul jump.

jsr     Path8(r0),r31   ; The return address is saved in r31 and
                        ;  PC is set to the byte address of Path8,
add     0x321,r0,r9     ; Execute this add before Path8 subroutine entry.
sub     r7,r8,r9        ; Execute this sub after Path8 subroutine returns.

jsr.a   8(r6),r31       ; Annul delay slot ”nop” instruction executed
                        ;  before call of subroutine at (r6 + 8) bytes.
addu    r7,r8,r9        ; Execute this addu after subroutine returns.
. . .
Path8: . . .
```

Syntax **ld**[{**.b**|**.h**|**.d**}] *offset*[**:s**](*base*[**:m**]),*dest*

Operation
IF *offset*, then form (*offset* + *base*) → *addr*
IF *offset***:s**, then scale by data size as (*offset*<<*sz*) + *base* → *addr*
IF *base***:m**, then update *base* as *addr* → *base*
SIGN_EXTEND(MEM(*addr*)) → *dest*

Operands
I(R),R Immediate Form (does not support **:s** scaling)
R(R),R Register Form
L(R),R Long-Immediate Form

Encoding

Notes: 1) m = Modify base address; S = scale offset by data size; sz = data size.

2) sz can have a value of 0, 1, 2, or 3. 0 indicates an 8-bit byte, 1 indicates a 16-bit halfword, 2 indicates a 32-bit word (default), and 3 indicates a 64-bit doubleword.

Description If you specify offset scaling by using the **:s** suffix, then the data in the indirect *offset* register (or long immediate data) is shifted left 0, 1, 2, or 3 for byte, halfword, word, or doubleword sizes, respectively. The word size is the default, and you can specify a byte, halfword, or doubleword by using **.b**, **.h**, or **.d** in the command syntax. See Table 10–19.

Table 10–19. Data Size and Offset Values for ld Instruction

Opcode Syntax	Data Size	offset:s left shift	sz
ld.b	8-bit least significant byte	0	0
ld.h	16-bit least significant halfword	1	1
ld	32-bit word	2	2
ld.d	64-bit doubleword	3	3

The optionally scaled offset is then added to data in the *base* register to form memory address *addr*. If you specify modify base address with the **:m** suffix, then the sum *addr* is written back into *base* register as the memory access is issued.

If the data at memory address *addr* is not resident, a data-cache miss is issued to the TC. Once the data is resident in data cache, the data at the memory address *addr* is loaded into the least significant part of the *dest* register with left sign extension, if byte or halfword. If the data is a 64-bit doubleword, then the data is loaded into an (even,odd) register pair. The 32 LSBs are loaded into the even *dest* register, and the 32 MSBs are loaded into the odd *dest*+1 register. See subsection 10.9.2 for additional information about using r0 and r1.

Memory is accessed according to the data size alignment. Halfwords ignore memory address LSB, words ignore two LSBs, doublewords ignore three LSBs.

Notes:

1) Loads require one extra cycle before data is available for use. Since the *dest* register is scoreboarded, a nop instruction is not needed; however, the program often can place a useful instruction in the scoreboard time slot.

2) Double-precision loads and stores are independent of the external big- or little-endian memory organization. Endian ordering is transparent to the program when using 64-bit loads or stores—the most significant word is always in the odd (even + 1) register, and the least significant word is always in the even register (≥ 2).

3) When accessing TC or VC on-chip registers with a load instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

4) If the *dest* and *base* are the same register, the *base* value will be overwritten when the *dest* register is loaded. For example, assume r13 = 0x10 and memory location 0xC = 8. In this example,

   ```
   ld –4(r13:m),r13  ; destination r13 = MEM(0xC)
   ```

   memory location 0xC (0x10 – 4) is written to r13 (the *dest* register), and the value 8 is written over the modified *base* value (0xC).

Example Assume the following values before each instruction:

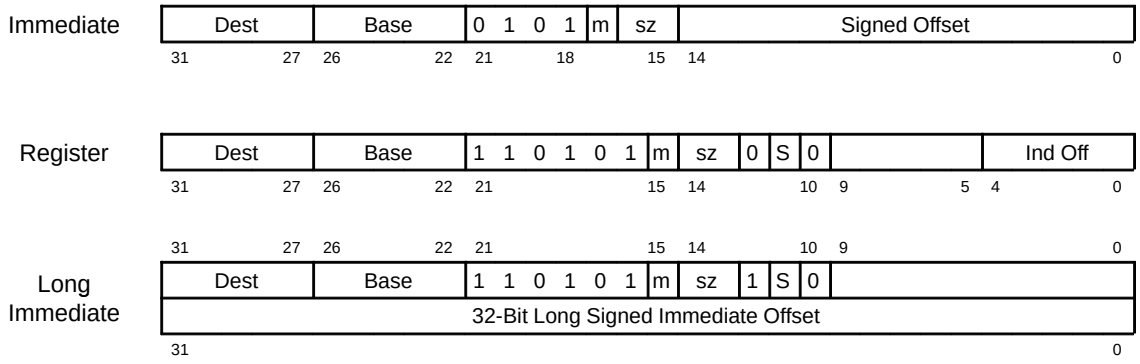
```
                        MEMORY        63                0
                        02000180:     A1340009  74911222
                        02000188:     9001c002  A0031004

                        REGISTERS
                        r5:     02000180
                        r6:     00000004
                        r8:     9ABCDEF0
                        r9:     AAAAAAAA

                          ;             Big-Endian Memory   Little-Endian Memory
                          ;    r5          r8        r9          r8        r9
                          ;   After       After     After       After     After

ld.h    r6(r5),r8         ; 02000180    00007491  AAAAAAAA    00000009  AAAAAAAA
ld      0x2000184(r0),r8  ; 02000180    74911222  AAAAAAAA    A1340009  AAAAAAAA
ld.d    8(r5),r8          ; 02000180    A0031004  9001C002    A0031004  9001C002
ld.b    3(r5:m),r8        ; 02000183    00000009  AAAAAAAA    00000074  AAAAAAAA

ld      r6(r5),r8         ; load r8 with a 32-bit word
                          ; one cycle scoreboard delay allows ld to complete
add     r8,r9,r10         ; use data from ld

ld      r6(r5),r8         ; load r8 with a 32-bit word
xor     0x1234,r9,r9      ; instruction in scoreboard delay slot
                          ;   allows ld to complete
add     r8,r9,r10         ; use data from ld

addu    1,r0,r14          ; r14 = 1
addu    0x2000188,r0,r12  ; r12 = 0x0200 0188
ld.h    0x200018A(r0),r11 ; r11 = 0xFFFF C002 (big endian)
                          ; r11 = 0xFFFF A003 (little endian)
ld.h    r14:s(r12),r11    ; r11 is the same as the previous instruction
```

Syntax **ld.u**[{**.b**|**.h**}] *offset*[**:s**](*base*[**:m**]),*dest*

Operation
IF *offset*, then form (*offset* + *base*) → *addr*
IF *offset***:s**, then scale by data size as (*offset*<<*sz*) + *base* → *addr*
IF *base***:m**, then update *base* as *addr* → *base*
ZERO_EXTEND(MEM(*addr*)) → *dest*

Operands
I(R),R Immediate Form (does not support **:s** scaling)
R(R),R Register Form
L(R),R Long-Immediate Form

Encoding



Notes: 1) m = Modify base address; S = scale offset by data size; sz = data size.

2) sz can have a value of 0, 1, 2, or 3. 0 indicates an 8-bit byte, 1 indicates a 16-bit halfword, and 2 and 3 are reserved.

Description If you specify offset scaling by using the **:s** suffix, then the data in the indirect *offset* register (or long-immediate data) is shifted left 0 or 1 for byte or halfword sizes, respectively. You can specify a byte or halfword by using **.b** or **.h** in the command syntax. See Table 10–20.

Table 10–20. Data Size and Offset Values for ld.u Instruction

Opcode Syntax	Data Size	offset:s left shift	sz
ld.ub	8-bit least significant byte	0	0
ld.uh	16-bit least significant halfword	1	1
	reserved	2	2
	reserved	3	3

The optionally scaled offset is then added to data in register *base* to form memory address *addr*. If you specify modify base address with the **:m** suffix, then the sum *addr* is written back into register *base* as the memory access is issued.

If the data at memory address *addr* is not resident, a data-cache miss is issued to the TC. Once the data is resident in data cache, the data at the memory address *addr* is loaded into the least significant part of the *dest* register with left zero extension for byte or halfword.

Memory is accessed according to the data size alignment; for example, halfwords ignore memory address LSB.

Notes:

1) Loads require one extra cycle before data is available for use. Since the *dest* register is scoreboarded, a nop instruction is not needed; however, the program often can place a useful instruction in the scoreboard time slot.

2) When accessing TC or VC on-chip registers with a load instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

3) If *offset, base*, and *dest* are not available, execution stalls until they are all available.

4) Double-precision loads and stores are independent of the external big- or little-endian memory organization. Endian ordering is transparent to the program when using 64-bit loads or stores—the most significant word is always in the odd (even + 1) register, and the least significant word is always in the even register (≥ 2).

5) If the *dest* and *base* are the same register, the *base* value will be overwritten when the *dest* register is loaded. For example, assume r13 = 0x10 and memory location 0xC = 8. In this example,

   ```
   ld.uh –4(r13:m),r13  ; destination r13 = MEM(0xC)
   ```

   memory location 0xC (0x10 – 4) is written to r13 (the *dest* register), and the value 8 is written over the modified *base* value (0xC).

6) See subsection 10.9.2 for additional information about using r0 and r1.

Example Assume the following values before each instruction:

```
MEMORY          63                0
02000180:       A1340009  74911222
02000188:       9001c002  A0031004

REGISTERS
r5:       02000180
r6:       00000004
r8:       9ABCDEF0
r9:       AAAAAAAA
```

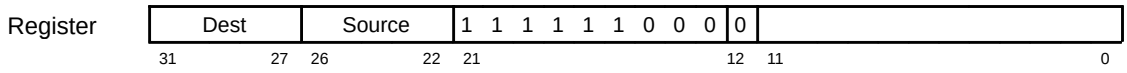
```
                        ;              Big-Endian Memory    Little-Endian Memory
                        ;     r5           r8        r9          r8        r9
                        ;   After        After     After       After     After

ld.uh   r6(r5),r8       ; 02000180     00007491  AAAAAAAA    00000009  AAAAAAAA
ld.ub   0x2000184(r0),r8; 02000180     00000074  AAAAAAAA    00000009  AAAAAAAA
ld.uh   8(r5),r8        ; 02000180     00009001  AAAAAAAA    00001004  AAAAAAAA
ld.ub   3(r5:m),r8      ; 02000183     00000009  AAAAAAAA    00000074  AAAAAAAA

ld.uh   r6(r5),r8       ; load 32-bit r8's LShalfword
                        ; one cycle scoreboard delay allows ld to complete
addu    r8,r9,r10       ; use data from ld instruction

ld.uh   r6(r5),r8       ; load 32-bit r8's LShalfword
xor     0x1234,r9,r9    ; instruction in scoreboard delay slot

addu    r8,r9,r10       ; use data from ld

addu    1,r0,r14        ; r14 = 1
addu    0x2000188,r0,r12; r12 = 0x0200 0188
ld.uh   0x200018A(r0),r11; r11 = 0x0000 C002 (big endian)
                        ; r11 = 0x0000 A003 (little endian)
ld.uh   r14:s(r12),r11  ; r11 is the same as the previous instruction
```

Syntax **lmo** *source,dest*

Operation 31–*n* → *dest* where *n* = bit number of the leftmost 1 in *source*

Operands R,R Register Form

Encoding



Description The lmo instruction first determines the bit number (number 31–0) of the leftmost one of the data in the *source* register. Next, this bit number is subtracted from 31, and the result is written into the *dest* register. If the data in the *source* register is 0, the result is 32. The result is a shift amount that left-shifts the leftmost 1 to bit position 31 (for example, to the MSB in the 32-bit word). See Table 10–21.

Table 10–21. Bit Position of Leftmost One

Bit Position of Leftmost One (Binary Representation)								
31	24	23	16	15	8	7	0	**Result**
10000000		00000000		00000000		00000000		0
01000000		00000000		00000000		00000000		1
00100000		00000000		00000000		00000000		2
00010000		00000000		00000000		00000000		3
00001000		00000000		00000000		00000000		4
00000100		00000000		00000000		00000000		5
00000010		00000000		00000000		00000000		6
00000001		00000000		00000000		00000000		7
00000000		10000000		00000000		00000000		8
00000000		01000000		00000000		00000000		9
00000000		00100000		00000000		00000000		10
00000000		00010000		00000000		00000000		11
00000000		00001000		00000000		00000000		12
00000000		00000100		00000000		00000000		13
00000000		00000010		00000000		00000000		14
00000000		00000001		00000000		00000000		15
00000000		00000000		10000000		00000000		16
00000000		00000000		01000000		00000000		17
00000000		00000000		00100000		00000000		18

Table 10–21. Bit Position of Leftmost One (Continued)

Bit Position of Leftmost One (Binary Representation)								
31	24	23	16	15	8	7	0	**Result**
00000000		00000000		00010000		00000000		19
00000000		00000000		00001000		00000000		20
00000000		00000000		00000100		00000000		21
00000000		00000000		00000010		00000000		22
00000000		00000000		00000001		00000000		23
00000000		00000000		00000000		10000000		24
00000000		00000000		00000000		01000000		25
00000000		00000000		00000000		00100000		26
00000000		00000000		00000000		00010000		27
00000000		00000000		00000000		00001000		28
00000000		00000000		00000000		00000100		29
00000000		00000000		00000000		00000010		30
00000000		00000000		00000000		00000001		31
00000000		00000000		00000000		00000000		32

Example 1 Shift the leftmost one to MSB

```
                        ;      r7          r9          r9
                        ;    before      before      after
lmo     r7,r9           ;   00089ABC        –        0000000C
shl     r9,32,r7,r9     ;   00089ABC    0000000C    89ABC000   LMO to MSB
```

Example 2 Shift off the leftmost one

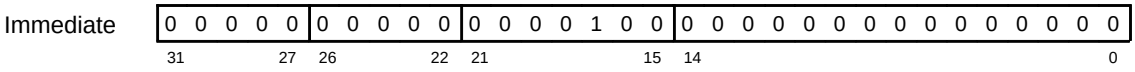
```
        bcnd.a  ALL0,r7,eq0.w   ; branch if 0, otherwise do LMO
                                ;     r7        r9        r9
                                ;   before    before    after
        lmo     r7,r9           ;  00089ABC      –     0000000C
        addu    +1,r9,r9        ;  00089ABC  0000000C  0000000D
        shl     r9,32,r7,r9     ;  00089ABC  0000000D  13578000  LMO lost
ALL0:   nop                     ;
```

Syntax **nop**

Operation No operation

Operands None

Encoding



Description The instruction set does not contain a specific nop instruction, but many of the instructions can be used to perform no operation. The encoding shown above is **rdcr 0,r0**, which is the value generated by the assembler. Your code should not rely on a particular bit pattern for the nop mnemonic.

Example

```
nop                     ; Let assembler choose no operation.

xnor    r0,r0,r0        ; This instruction may be used.

or      0x1234,r5,r0    ; So can this.
```

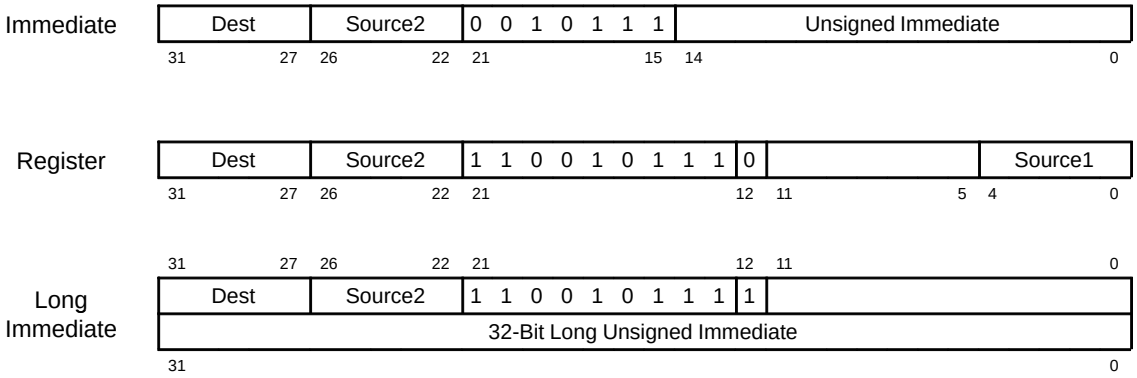
Syntax **or[.tt]** *source1,source2,dest*

Operation *source1* OR *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description The bitwise OR of data from the *source1* register (or the immediate data) and data from the *source2* register is written into the *dest* register.

Example

```
                              ;       r7              r8              r9
                              ;     before          before          after

or      r7,r8,r9              ;    0000000F        12345678        1234567F

or      0x7FFF,r8,r9          ;       -            12345678        12347FFF

or      0xFF0000FF,r8,r9      ;       -            12345678        FF3456FF

or.tt   r7,r8,r9              ;    0000000F        12345678        1234567F

or.tt   0x7FFF,r8,r9          ;       -            12345678        12347FFF

or.tt   0xFF0000FF,r8,r9      ;       -            12345678        FF3456FF
```

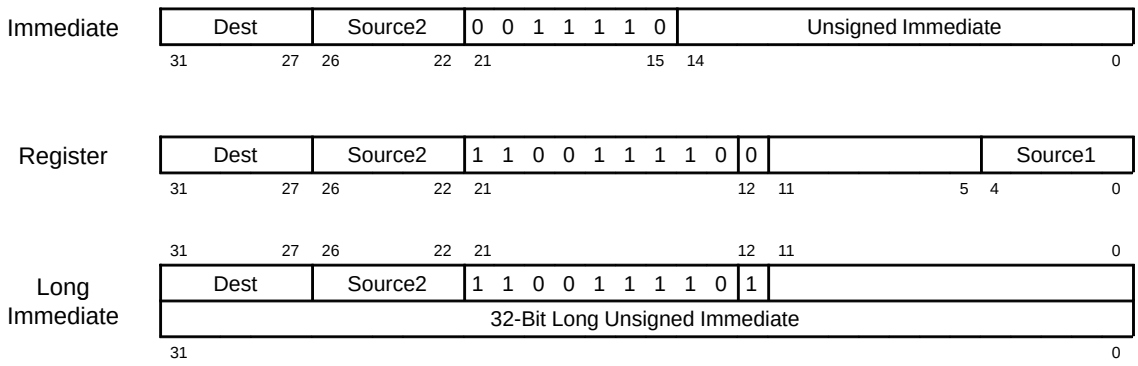
Syntax **or.ff** *source1,source2,dest*

Operation ~(*source1*) OR ~(*source2*) → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from the *source1* register (or the immediate data) and data from the *source2* register are both inverted (ones complement), and a bitwise OR of these inverted values is written into the *dest* register.

Example

```
                              ;       r7              r8              r9
                              ;     before          before          after

or.ff   r7,r8,r9              ;    0000000F        12345678        FFFFFFF7

or.ff   0x7FFF,r8,r9          ;        -           12345678        FFFFA987

or.ff   0xFF0000FF,r8,r9      ;        -           12345678        EDFFFF87
```

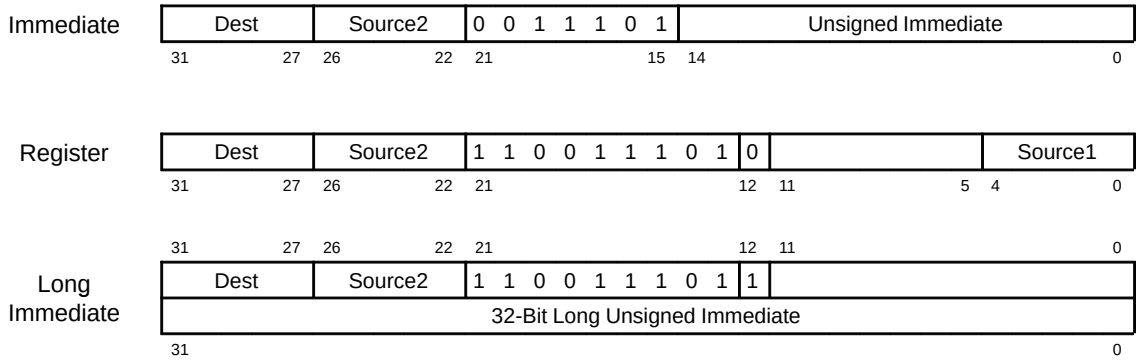
Syntax **or.ft** *source1,source2,dest*

Operation ~(*source1*) OR *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from *source1* register (or the immediate data) is inverted (ones complement); a bitwise OR of this inverted value and the data from the *source2* register is written into the *dest* register.

Example

```
                                ;       r7              r8              r9
                                ;    before          before          after

or.ft   r7,r8,r9                ;   0000000F        12345678        FFFFFFF8

or.ft   0x7FFF,r8,r9            ;       -           12345678        FFFFD678

or.ft   0xFF0000FF,r8,r9        ;       -           12345678        12FFFF78
```

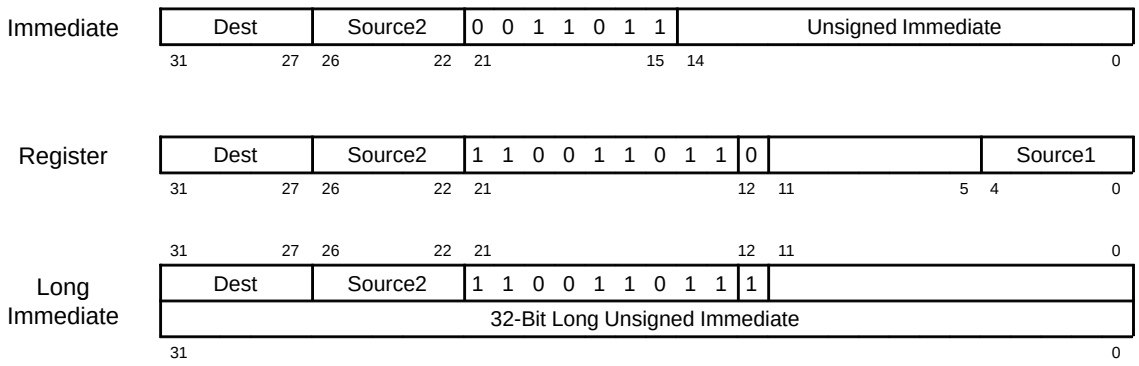
Syntax **or.tf** *source1,source2,dest*

Operation *source1* & ~(*source2*) → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from the *source2* register is inverted (ones complement), a bitwise OR of this inverted value and the data from the *source1* register (or the immediate data) is written into the *dest* register.

Example

```
                              ;       r7          r8          r9
                              ;    before      before       after
or.tf   r7,r8,r9              ;  0000000F    12345678    EDCBA98F
or.tf   0x7FFF,r8,r9          ;        -     12345678    EDCBFFFF
or.tf   0xFF0000FF,r8,r9      ;        -     12345678    FFCBA9FF
```

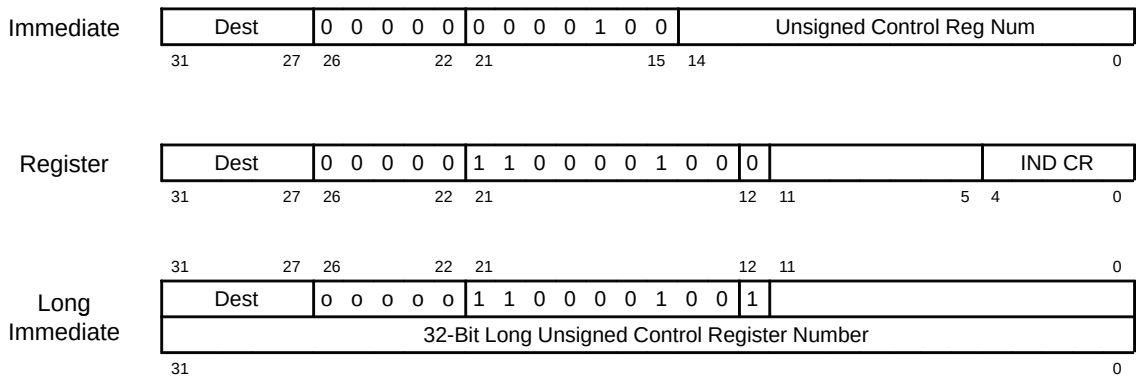
Syntax **rdcr** *creg,dest*

Operation *creg* → *dest*

Operands

I,R	Immediate Form
R,R	Register Form
L,R	Long-Immediate Form

Encoding



Description

The *creg* field (or the immediate data) is used to select a control register. The value in that control register is written into the *dest* register. In the register form, the indirect control register number is held in the *creg* register.

If you use an undefined control register number, the operation in undefined, and you do not receive an error signal.

> **Note:**
>
> For a list of control registers, see Table 2–1, *MP Control Register Numbers*.

Example

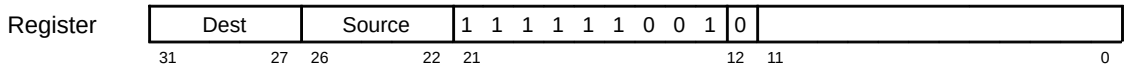
```
                              ;       IE              r9              r9
                              ;    before         before          after

rdcr    IE,r9                 ;    00000007       12345678        00000007

rdcr    IN0P,r2               ; r2 = Input 0 Pointer
rdcr    IN1P,r4               ; r4 = Input 1 Pointer
rdcr    OUTP,r6               ; r6 = Output Pointer

addu    0x400,r0,r7           ; r7 = 0x400 (DTAG0)
rdcr    r7,r8                 ; r8 = DTAG0
```

Syntax **rmo** *source,dest*

Operation 31–*n* → *dest* where *n* = bit number of the rightmost 1 in *source*

Operands R,R Register Form

Encoding



Description The rmo instruction first determines the bit number (number 31–0) of the rightmost one of the data in the *source* register. Next, this bit number is subtracted from 31 (forms ones complement), and the result written into the *dest* register. If the data in the *source* register is 0, then the result is 32. See Table 10–22.

Table 10–22. Bit Position of Rightmost One

Bit Position of Rightmost One (Binary Representation)								
31	24	23	16	15	8	7	0	**Result**
10000000		00000000		00000000		00000000		0
01000000		00000000		00000000		00000000		1
00100000		00000000		00000000		00000000		2
00010000		00000000		00000000		00000000		3
00001000		00000000		00000000		00000000		4
00000100		00000000		00000000		00000000		5
00000010		00000000		00000000		00000000		6
00000001		00000000		00000000		00000000		7
00000000		10000000		00000000		00000000		8
00000000		01000000		00000000		00000000		9
00000000		00100000		00000000		00000000		10
00000000		00010000		00000000		00000000		11
00000000		00001000		00000000		00000000		12
00000000		00000100		00000000		00000000		13
00000000		00000010		00000000		00000000		14
00000000		00000001		00000000		00000000		15
00000000		00000000		10000000		00000000		16
00000000		00000000		01000000		00000000		17
00000000		00000000		00100000		00000000		18
00000000		00000000		00010000		00000000		19
00000000		00000000		00001000		00000000		20

Table 10–22. Bit Position of Rightmost One (Continued)

Bit Position of Rightmost One (Binary Representation)								
31	24	23	16	15	8	7	0	**Result**
00000000		00000000		00000100		00000000		21
00000000		00000000		00000010		00000000		22
00000000		00000000		00000001		00000000		23
00000000		00000000		00000000		10000000		24
00000000		00000000		00000000		01000000		25
00000000		00000000		00000000		00100000		26
00000000		00000000		00000000		00010000		27
00000000		00000000		00000000		00001000		28
00000000		00000000		00000000		00000100		29
00000000		00000000		00000000		00000010		30
00000000		00000000		00000000		00000001		31
00000000		00000000		00000000		00000000		32

Example 1 Shift the rightmost one to LSB

```
                        ;      r7          r9          r9
                        ;    before      before      after
rmo     r7,r9           ;   DCBA9800        –       00000014
xor     0x3f,r9,r9      ; Ones complement of r9 (5 LSBs)
srl     r9,32,r7,r9     ;   DCBA9800    0000002B    001B9753   RMO to LSB
```

Example 2 Shift off the rightmost one

```
        bcnd.a  ALL0,r7,eq0.w   ; branch if 0, otherwise do RMO
                                ;     r7        r9        r9
                                ;   before    before    after
        rmo     r7,r9           ;  DCBA9800      –      00000014
        subu    0,r9,r9         ;  DCBA9800  00000014  FFFFFFEC
        srl     r9,32,r7,r9     ;  DCBA9800  FFFFFFEC  000DCBA9  RMO lost
ALL0:   nop                     ;
```

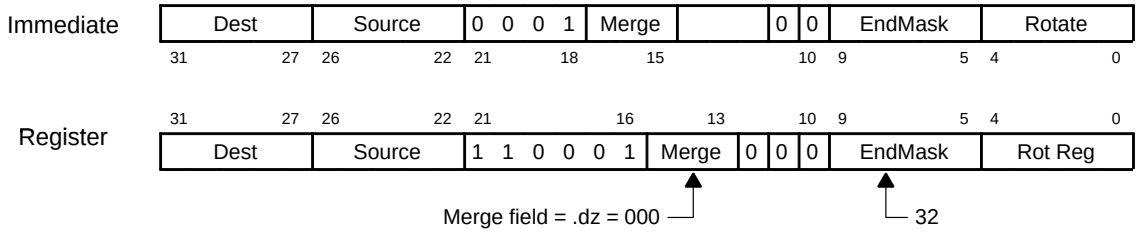
Syntax **rotl** *rotate,***32**,*source,dest*

Operation (*source* \\ *rotate*) → *dest*

Operands

I,I,R,R	Immediate Form
R,I,R,R	Register Form

Encoding



Description Data from the rotated *source* register is written into the *dest* register. Note that 32 is used for *endmask*. The assembler uses the **sl.dz** instruction for the rotate register left rotl mnemonic:

- The shiftmask option uses disabled (d) (see **sl.**{**d**|**e**|**i**}{...})
- The endmask option uses zero extend (z) (see **sl.**{...}{**m**|**s**|**z**})

Data in register *source* is rotated left by:

- **immediate form**—the value in the *rotate* field
- **register form**—the five LSBs of data in register *rotate*

The *endmask* is generated by:

- **immediate form**—from the *endmask* field as shown in Table 10–24, *Mask Values as* $2^{Code}-1$ *in Shift Operations*
- **register form**—from the value in the *endmask* field:
  - 0, 32: endmask is all 1s (default when *endmask* is supplied as 32 in the source statement).
  - 1–31: endmask from five LSBs of odd register *rotate*+1 as shown in Table 10–24. Note that in this case, *rotate* must be an even register. Do not supply *rotate* as an odd register.

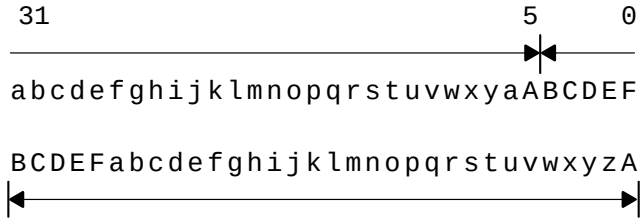
A bitwise AND of the rotated data in the *source* register and the endmask is written into the *dest* register.

In summary, the rotate register left operation rotl is implemented by

dest = (*source* \\ *rotate*) & *endmask*

Example 1 Rotate r7 left 27 bits, write result into r9

```
rotl    27,32,r7,r9       ; r7 \\ 27 = r9
```




```
                                  ; Input LSB = 5
abcdefghijklmnopqrstuvwxyaABCDEF  ; r7 input register

BCDEFabcdefghijklmnopqrstuvwxyzA  ; r7\\27 (LSB = 0)
                                  ; rotate r7 left 27 bits

11111111111111111111111111111111  ; EndMask = Table(EndMask)
                                  ; EndMask of 32 = 0 in 5 bits

BCDEFabcdefghijklmnopqrstuvwxyzA  ; r9 = (r7\\27) & (–1)
                                  ; Output LSB is bit 0
 31                            0  ; for r9 = r7\\27
                                  ; 0x01234567\\27 = 0x38091A2B
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- ❑ r4 = 27 (rotate amount)
- ❑ r5 = 32 (width field value)
- ❑ r7 = 0x123 4567 (original value)
- ❑ r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0x3809 1A2B:

```
rotl  27,32,r7,r9 ; Immediate form
```

or

```
rotl  r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0xC7F6 E5D4:

```
rotl  27,32,r9,r7 ; Immediate form
```

or

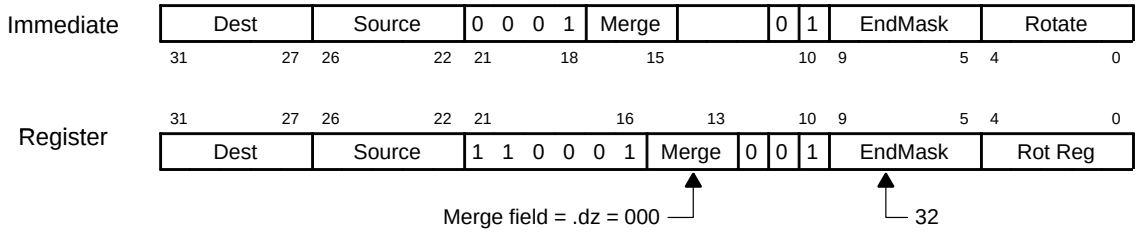
```
rotl  r4,31,r9,r7 ; Register form
```

Syntax **rotr** *rotate*,**32**,*source*,*dest*

Operation (*source* // *rotate*) → *dest*

Operands
I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding



Description Data from the rotated *source* register is written into the *dest* register. Note that 32 is used for *endmask*. The assembler uses the **sr.dz** instruction for the rotate register right rotr mnemonic:

- The shiftmask option uses disabled (d) (see **sr.**{**d**|**e**|**i**}{...})
- The endmask option uses zero extend (z) (see **sr.**{...}{**m**|**s**|**z**})

Data in register *source* is rotated right by:

- **immediate form**—the value in the *rotate* field
- **register form**—the five LSBs of data in register *rotate*

The *endmask* is generated by:

- **immediate form**—from the *endmask* field as shown in Table 10–24, *Mask Values as* $2^{Code}-1$ *in Shift Operations*
- **register form**—from the value in the *endmask* field:
  - 0, 32—endmask is all 1s (default when *endmask* is supplied as 32).
  - 1–31—endmask from five LSBs of odd register *rotate*+1, as shown in Table 10–24. Note that in this case, *rotate* must be an even register. Do not supply *rotate* as an odd register.

A bitwise AND of the rotated data in the *source* register and the endmask is written into the *dest* register.

In summary, the rotate register right operation rotr is implemented by

dest = (*source* // *rotate*) & *endmask*

Example 1 Rotate r7 right 5 bits, write result into r9

```
rotr    5,32,r7,r9        ; r7 // 5 = r9

 31                            5    0
                                        ; Input LSB = 5
abcdefghijklmnopqrstuvwxyaABCDEF        ; r7 input register

BCDEFabcdefghijklmnopqrstuvwxyzA        ; r7//5 (LSB = 0)
                                        ; rotate r7 right 5 bits

11111111111111111111111111111111        ; EndMask = Table(EndMask)
                                        ; EndMask of 32 = 0 in 5 bits

BCDEFabcdefghijklmnopqrstuvwxyzA        ; r9 = (r7//5) & (–1)
                                        ; Output LSB is bit 0
 31                                 0   ; for r9 = r7//5
                                        ; 0x01234567//5 = 0x38091A2B
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 5 (rotate amount)
- r5 = 32 (width field value)
- r7 = 0x123 4567 (original value)
- r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0x3809 1A2B:

```
rotr  5,32,r7,r9  ; Immediate form
```

or

```
rotr  r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0xC7F6 E5D4:

```
rotr  5,32,r9,r7  ; Immediate form
```

or

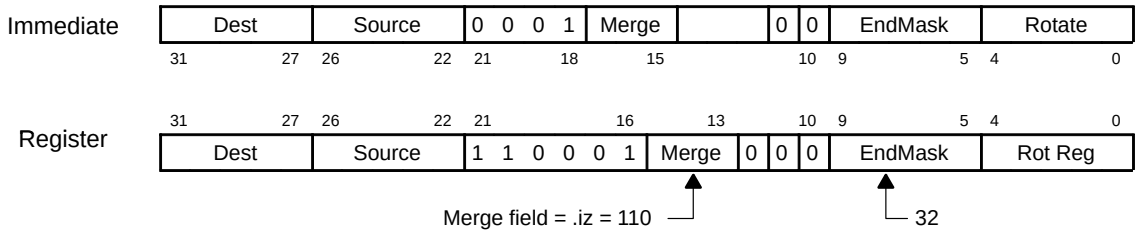
```
rotr  r4,31,r9,r7 ; Register form
```

Syntax **shl** *rotate*,**32**,*source*,*dest*

Operation (*source* << *rotate*) → *dest*

Operands
I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding



Description Data from the rotated *source* register is written into the *dest* register. Note that 32 is used for *endmask*. The assembler uses the **sl.iz** instruction for the rotate register left shl mnemonic:

- ❑ The shiftmask option uses inverted (i) (see **sl.{d|e|i}**{...})
- ❑ The endmask option uses zero extend (z) (see **sl.**{...}{**m**|**s**|**z**})

Data in register *source* is rotated left by:

- ❑ **immediate form**—the value in the *rotate* field
- ❑ **register form**—the five LSBs of data in register *rotate*

The *shiftmask* is generated by:

- ❑ **immediate form**—from the *rotate* field as shown in Table 10–24, *Mask Values as* 2^{Code}*–1 in Shift Operations*
- ❑ **register form**—the five LSBs of data in register *rotate*, as shown in Table 10–24

A bitwise AND of the rotated data in the *source* register and the inverted (ones complement) shiftmask is written into the *dest* register.

In summary, the shift register left operation shl is implemented by

dest = (*source* \\ *rotate*) & ~*shiftmask*

Example 1 Shift r7 left 5 bits, write result into r9

```
shl     5,32,r7,r9        ; r7 << 5 = r9

 31   26                          5     0
      |<--------------------------------|   ; Source MSB = 26
 abcdefghijklmnopqrstuvwxyaABCDEF   ; r7 input register

 fghijklmnopqrstuvwxyzABCDEFabcde   ; r7 \\ 5 (MSB = 31)
|<-------------------------------|          ; Rotate r7 left 5 bits

 00000000000000000000000000011111   ; Shiftmask = Table(Rotate)
                                    ; where Rotate = 5

 11111111111111111111111111100000   ; Inverted shiftmask

 fghijklmnopqrstuvwxyzABCDEF00000   ; r9 = (r7\\5) & (~0x1F)
|<-------------------------------|          ; output MSB = 31
 31                          5     0   ; for r9 = r7 << 5
                                    ; 0x01234567 << 5 = 0x2468ACE0
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 5 (rotate amount)
- r5 = 32 (width field value)
- r7 = 0x123 4567 (original value)
- r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0x2468 ACE0:

```
shl   5,32,r7,r9  ; Immediate form
```

or

```
shl   r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0xDB97 5300:

```
shl   5,32,r9,r7  ; Immediate form
```

or

```
shl   r4,31,r9,r7 ; Register form
```

Syntax **sl.{d|e|i}{m|s|z}** *rotate,endmask,source,dest*

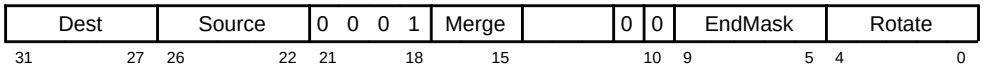
Operation ((*source* \\ *rotate*) & CM) OR (*dest* & ~CM) → *dest*

where the composite mask CM = *shiftmask* & *endmask*

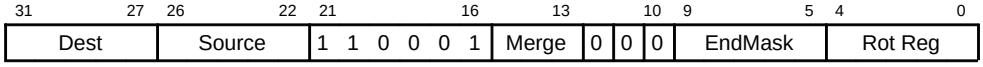
Operands I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding

Immediate



Register



Description Table 10–23 lists the alternate mnemonics that the assembler supports.

Table 10–23. Alternate Shift Mnemonics Supported by the Assembler

Operation	Alternate	Uses
Insert field	ins	sl.im
Extract signed field	exts	sr.ds
Extract unsigned field	extu	sr.dz
Rotate left	rotl	sl.dz
Rotate right	rotr	sr.dz
Shift left logical	shl	sl.iz
Shift right arithmetic	sra	sr.es
Shift right logical	srl	sr.ez

The data in the *source* register is rotated left by the amount specified in the *rotate* field (immediate form) or by the five LSBs of the data in the *rotate* register (register form).

A composite mask (CM) is formed from the shiftmask and the endmask. This CM is used to merge the rotated *source* with the data in the *dest* register.

Endmask is generated from {**m**|**s**|**z**} (merge fields, sign extend, or zero extend):

- **immediate form**—from the *endmask* field as shown in Table 10–24, *Mask Values as 2^{Code}–1 in Shift Operations*
- **register form**—from the value in the *endmask* field:
  - 0, 32—endmask is all ones
  - 1–31—endmask from five LSBs of odd register *rotate*+1 as shown in Table 10–24. Note that in this case, *rotate* must be an even register. Do not supply *rotate* as an odd register.
- **optionally invert the endmask** (sli.xx or sri.xx instruction)

Shift or rotate amount determination is:

- **immediate form**—from the *rotate* field
- **register form**—from the five LSBs of data in the *rotate* register

Shiftmask is generated from {**d**|**e**|**i**} (disabled, enabled, or inverted):

- 5-bit shift or rotate amount determines shiftmask from Table 10–24 if shiftmask is enabled (sl.ex, sl.ix, sli.ex, or sli.ix)
- 5-bit shift or rotate amount is negated to determine shiftmask from Table 10–24 if shiftmask is enabled (sr.ex, sr.ix, sri.ex, or sri.ix)
- All 1s is effectively used for the shiftmask if the shiftmask is disabled (sl.dx, sli.dx, sr.dx, or sri.dx)
- Code of 0 returns a shiftmask of all 0s for the merge option (sxx.xm), or if the shiftmask is to be inverted (sxx.ix)
- Optionally invert the shiftmask (sl.ix, sli.ix, sr.ix, or sri.ix)

Table 10–24. Mask Values as $2^{Code}-1$ in Shift Operations

Code	Mask in Hex	Code	Mask in Hex	Code	Mask in Hex
0	0x0000 0000	11	0x0000 07FF	22	0x003F FFF
1	0x0000 0001	12	0x0000 0FFF	23	0x007F FFF
2	0x0000 0003	13	0x0000 1FFF	24	0x00FF FFFF
3	0x0000 0007	14	0x0000 3FFF	25	0x01FF FFFF
4	0x0000 000F	15	0x0000 7FFF	26	0x03FF FFFF
5	0x0000 001F	16	0x0000 FFFF	27	0x07FF FFFF
6	0x0000 003F	17	0x0001 FFFF	28	0x0FFF FFFF
7	0x0000 007F	18	0x0003 FFFF	29	0x1FFF FFFF
8	0x0000 00FF	19	0x0007 FFFF	30	0x3FFF FFFF
9	0x0000 01FF	20	0x000F FFFF	31	0x7FFF FFFF
10	0x0000 03FF	21	0x001F FFFF	32	0xFFFF FFFF

The merge field encoding is derived from the suffix, as shown in Table 10–25.

Table 10–25. Encoding for Merge Field (Shift Instructions)

Suffix	Shift Mask	0 Means	Merge Option	Merge Field
.dz	Disabled		Zero	000
.dm	Disabled		Merge	001
.ds	Disabled		Sign	010
.ez	Enable	32	Zero	011
.em	Enable	0	Merge	100
.es	Enable	32	Sign	101
.iz	Invert	0	Zero	110
.im	Invert	0	Merge	111
.is	Not Applicable			

A composite mask (CM) is formed as:

CM = *shiftmask* AND *endmask*

The CM is then used to merge the rotated *source* with the data in the *dest* register. Bit n (n = 0,1,2,...,31) in the CM is tested for:

- 1, then *dest* bit n is replaced by rotated *source* bit n
- 0, then *dest* bit n is a function of {**m**|**s**|**z**} (merge, sign, or zero):
  - Merge—*dest* bit n is unchanged
  - Sign—*dest* bit n = *dest* bit n–1 (bit numbers < 0 are defined as 0)
  - Zero—*dest* bit n = 0

In summary, the result is generated by

dest = ((*source* \\ *rotate*) & CM) OR (*dest* & ~CM)

Example 1 See insert field **ins** example: insert r7 bits 0–6 r9 bits 4–10

```
ins:    sl.im   4,11,r7,r9      ; Insert r7 bits 0–6 into r9 bits 4–10
                                ; EndMask = 11 = Rotate (4) + Width (7)
                                ; ( (r7 \\ 4) & CM ) OR (r9 & ~CM) = r9
```

Example 2 See rotate register left **rotl** example: rotate r7 left 27 bits = r9

```
rotl:   sl.dz   27,32,r7,r9     ; Rotate r7 left 27 bits = r9
                                ; ( r7 \\ 27 ) = r9
```

Example 3 See shift left logical **shl** example: shift r7 left logical 5 bits = r9

```
shl:    sl.iz   5,32,r7,r9      ; Shift r7 left logical 5 bits = r9
                                ; ( r7 << 5 ) = r9
```

Syntax **sli.**{**d**|**e**|**i**}{**m**|**s**|**z**} *rotate,endmask,source,dest*

Operation ((*source* \\ *rotate*) & CM) OR (*dest* & ~CM) → *dest*

where the composite mask CM = *shiftmask* & ~*endmask*

Operands

I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding

Immediate

Dest	Source	0 0 0 1	Merge		1	0	EndMask	Rotate
31 27	26 22	21 18	15		10	9 5		4 0

Register

Dest	Source	1 1 0 0 0 1	Merge	0	1	0	EndMask	Rot Reg
31 27	26 22	21 16	13		10	9 5		4 0

Description

The sli.xx instruction performs in the same way as the sl.xx instruction performs, except that the endmask generated is inverted (ones complement). The composite mask (CM) is formed as

CM = *shiftmask* AND (~*endmask*)

In summary, the result is generated by

dest = ((*source* \\ *rotate*) & CM) OR (*dest* & ~CM)

Example 1 Insert r7 bits 7–27 into r9 bits 11–31 (compare this to the insert field **ins** example)

```
sli.im  4,11,r7,r9        ; Insert r7 bits 7-27 into r9 bits 11-31
                          ; EndMask = 11 = Rotate (4) + Width (7)
                          ; ( (r7 \\ 4) & CM) OR ( r9 & ~CM ) = r9

 31  27                      7       0   ; MSB = 27
     |<----------------------|           ; Input bits 7-27
abcdefghijklmnopqrstuvwxyaABCDEF        ; r7 input register

efghijklmnopqrstuvwxy9876543abcd        ; r7 \\ 4 (MSB = 31)
|<--------------------|                  ; Rotate r7 left 4 bits

00000000000000000000000000001111        ; Shiftmask = Table(Rotate)
                                        ; where Rotate = 4

11111111111111111111111110000           ; Inverted shiftmask

11111111111111111111100000000000        ; Inverted endmask

11111111111111111111100000000000        ; CM = ~shiftmask & ~endmask

.......ABCDEFGHIJKLMNPQRSTUVWXYZ        ; r9 input register

efghijklmnopqrstuvwxyPQRSTUVWXYZ        ; ((r7\\4) & CM) OR (r9 & ~CM)
|<--------------------|<----------|     ; Merged field = r9 output
 31                   10          0     ; r7 = 0x01234567
                                        ; r9 = 0xFEDCBA98 (original)
                                        ; r9 = 0x12345298 (output)
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 4 (rotate amount)
- r5 = 11 (width field value)
- r7 = 0x123 4567 (original value)
- r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0x1234 5298:

```
sli.im   4,11,r7,r9  ; Immediate form
```

or

```
sli.im   r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0xEDCB AD67:

```
sli.im   4,11,r9,r7  ; Immediate form
```

or

```
sli.im   r4,31,r9,r7 ; Register form
```

Syntax **sr.{d|e|i}{m|s|z}** *rotate,endmask,source,dest*

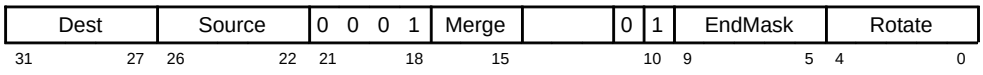
Operation ((*source* // *rotate*) & CM) OR (*dest* & ~CM) → *dest*

where the composite mask CM = *shiftmask* & *endmask*

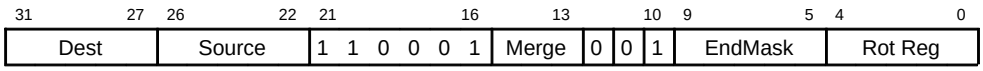
Operands I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding

Immediate



Register



Description The sr.xx instruction performs in the same way as the sl.xx instruction performs, except that the rotate amount is used as a right rotation. If the shiftmask is enabled, the negated (twos complement) rotate amount is used to generate the shiftmask from Table 10–24, *Mask Values as 2^{Code}–1 in Shift Operations*. The composite mask (CM) is formed as

CM = *shiftmask* AND *endmask*

In summary, the result is generated by

dest = ((*source* // *rotate*) & CM) OR (*dest* & ~CM)

Example 1 See rotate right **rotr** example: rotate r7 right 5 bits = r9

```
rotr:   sr.dz   5,32,r7,r9      ; Rotate r7 right 5 bits = r9
                                ; ( r7 // 5 ) = r9
```

Example 2 See shift register right arithmetic **sra** example: sign(r7>>5) = r9

```
sra:    sr.es   5,32,r7,r9      ; Shift r7 right arithmetic 5 bits = r9
                                ; Sign_Extend( r7 >> 5 ) = r9
```

Example 3 See shift register right logical **srl** example: zero(r7>>5) = r9

```
srl:    sr.ez   5,32,r7,r9      ; Shift r7 right logical 5 bits = r9
                                ; Zero_Extend( r7 >> 5 ) = r9
```

Example 4 See extract signed field **exts** example: sign(r7 bits 5–12) = LS r9

```
exts:   sr.ds   5,8,r7,r9   ; Signed 8-bit field extract of r7 bits 5–12
                            ; Sign_Extend( (r7 // 5) & 0xff ) = r9
```

Example 5 See extract unsigned field **extu** example: zero(r7 bits 5–12) = LS r9

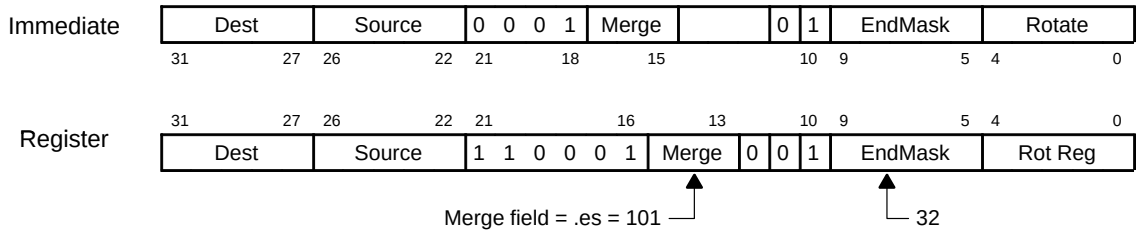
```
extu:  sr.dz  5,8,r7,r9   ; Unsigned 8-bit field extract of r7 bits 5–12
                          ; Zero_Extend( (r7 // 5) & 0xff ) = r9
```

Syntax **sra** *rotate*,**32**,*source*,*dest*

Operation SIGN_EXTEND(*source* >> *rotate*) → *dest*

Operands
I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding



Description The data in the *source* register is shifted right by the *rotate* amount, and the shifted result is written into the *dest* register with left sign extension. Note that 32 is used for *endmask*. The assembler uses the **sr.es** instruction for the shift right arithmetic sra mnemonic.

- The shiftmask option uses enable (e) (see **sr.**{**d**|**e**|**i**}{...})
- The endmask option uses sign (s) (see **sr.**{...}{**m**|**s**|**z**})

Data in register *source* is rotated right by:

- **immediate form**—the value in the *rotate* field
- **register form**—the five LSBs of data in register *rotate*

A shiftmask is generated from the twos complement of the *rotate* value as found in Table 10–24, *Mask Values as* 2^{Code-1} *in Shift Operations*.

A bitwise AND of the rotated data in the *source* register and the shiftmask is written into the *dest* register with left sign extension. In summary, the shift right arithmetic operation sra is accomplished by

dest = SIGN_EXTEND((*source* // *rotate*) & *shiftmask*)

Example 1 Shift r7 right arithmetic 5 bits, write left sign extended into r9

```
sra     5,32,r7,r9        ; Shift r7 bits 5–31 to r9 bits 0–26
                          ; Sign_Extend( r7 >> 5 ) = r9

 31                              5     0
 ◄──────────────────────────────►|       ; Input LSB = 5
 abcdefghijklmnopqrstuvwxyaABCDEF  ; r7 input register

 BCDEFabcdefghijklmnopqrstuvwxyzA  ; r7 // 5 (LSB = 5)
      ◄──────────────────────────►|  ; Rotate r7 right 5 bits

 00000111111111111111111111111111  ; Shiftmask = Table(–Rotate)
                                   ; where Rotate = 5

 aaaaaabcdefghijklmnopqrstuvwxyzA  ; (r7//5) & 0x7fffff
      ◄──────────────────────────►|  ; Sign_Extend(r7 >> 5) = r9
 31   26                       0

                                   ; 0x01234567 >> 5 = 0x00091A2B
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 5 (rotate amount)
- r5 = 32 (width field value)
- r7 = 0x123 4567 (original value)
- r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0x0009 1A2B:

```
sra   5,32,r7,r9  ; Immediate form
```

or

```
sra   r4,32,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0xFFF6 E5D4:

```
sra   5,32,r9,r7  ; Immediate form
```

or

```
sra   r4,32,r9,r7 ; Register form
```

Syntax **sri.**{**d**|**e**|**i**}{**m**|**s**|**z**} *rotate,mask,source,dest*

Operation ((*source* // *rotate*) & CM) OR (*dest* & ~CM) → *dest*

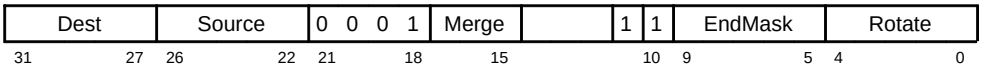
where the composite mask CM = *shiftmask* & *~endmask*

Operands

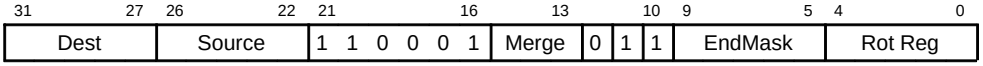
I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding

Immediate



Register



Description

The sri.xx instruction performs in the same way as the sl.xx instruction performs, except that the rotate amount is used as a right rotation. If the shiftmask is enabled, the negated (twos complement) rotate amount is used to generate the shiftmask from Table 10–24, *Mask Values as* 2^{Code}*–1 in Shift Operations*. In addition, the sri.xx instruction inverts (ones complement) the endmask.

The composite mask (CM) is formed as:

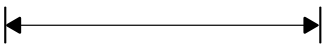
CM = *shiftmask* AND (*~endmask*)

In summary, the result is generated by:

dest = ((*source // rotate*) & CM) OR (*dest & ~CM*)

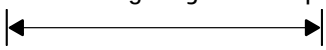
Example 1 r7 bits 16–31 replace r9 bits 12–26 (compare this to shift register right logical **srl** example)

```
sri.em   5,11,r7,r9                    ; r7 bits 16–31 replace r9
                                       ; bits 12–26

 31              16                    0
```



```
abcdefghijklmnopqrstuvwxyaABCDEF ; r7 input register

BCDEFabcdefghijklmnopqrstuvwxyzA ; r7 // 5 (26 = MSB)
```

```
00000111111111111111111111111111 ; Shiftmask = Table(–Rotate)
                                  ; where Rotate = 5

11111111111111111111100000000000 ; Inverted endmask=Table(width)
00000111111111111111100000000000 ; CM = shiftmask & endmask
98765.................PQRSTUVWXYZ ; r9 input

98765abcdefghijklmnopPQRSTUVWXYZ ; ((r7//5) & CM) OR (r9 & ~CM)
                                  ; = r9 rotated field merge
 31   26               11          0
                                  ; r7 = 0x01234567
                                  ; r9 = 0xFEDCBA98 (original)
                                  ; r9 = 0xF8091A98 (output)
```



Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- ❑ r4 = 5 (rotate amount)
- ❑ r5 = 11 (width field value)
- ❑ r7 = 0x123 4567 (original value)
- ❑ r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0xF809 1A98:

```
sri.em   5,11,r7,r9  ; Immediate form
```

or

```
sri.em   r4,31,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0x07F6 E567:

```
sri.em   5,11,r9,r7  ; Immediate form
```

or

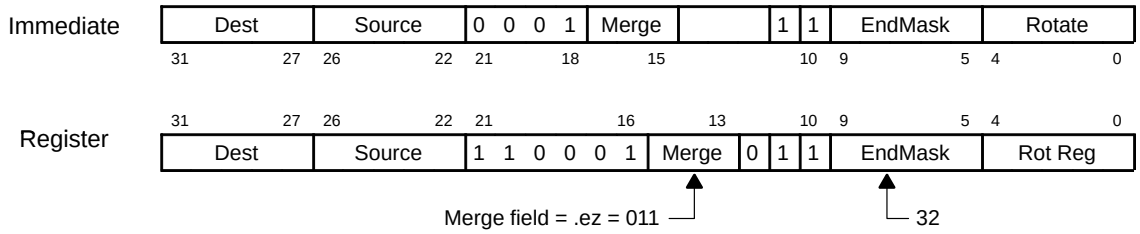
```
sri.em   r4,31,r9,r7 ; Register form
```

Syntax **srl** *rotate*,32,source,*dest*

Operation (*source* >> *rotate*) → *dest*

Operands
I,I,R,R Immediate Form
R,I,R,R Register Form

Encoding



Description The data in the *source* register is shifted right by the *rotate* amount, and the shifted result is written into the *dest* register. Note that 32 is used for *endmask*. The assembler uses the **sr.ez** instruction for the shift right logic sri mnemonic.

- The shiftmask option uses enable (e) (see **sr.**{**d**|**e**|**i**}{...})
- The endmask option uses zero (z) (see **sr.**{...}{**m**|**s**|**z**})

Data in register *source* is rotated right by:

- **immediate form**—the value in the *rotate* field
- **register form**—the five LSBs of data in register *rotate*

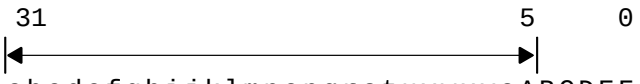
A shiftmask is generated from the twos complement of the *rotate* value as found in Table 10–24, *Mask Values as* 2^{Code-1} *in Shift Operations*.

A bitwise AND of the rotated data in the *source* register and the shiftmask is written into the *dest* register. In summary, the shift right arithmetic operation srl is implemented by

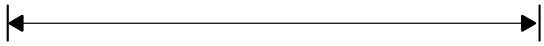
dest = (*source* // *rotate*) & *shiftmask*

Example 1 Shift r7 bits 5–31 right logical, write zero extended to r9 bits 0–26

```
sra     5,32,r7,r9        ; Shift r7 bits 5–31 to r9 bits 0–26
                          ; Sign_Extend( r7 >> 5 ) = r9
```




```
abcdefghijklmnopqrstuvwxyaABCDEF   ; r7 input register

BCDEFabcdefghijklmnopqrstuvwxyzA   ; r7 // 5 (LSB = 5)
```



```
                                   ; Rotate r7 right 5 bits

00000111111111111111111111111111  ; Shiftmask = Table(–Rotate)
                                   ; where Rotate = 5

aaaaaabcdefghijklmnopqrstuvwxyzA   ; (r7//5) & 0x7fffff
```

```
                                   ; Sign_Extend(r7 >> 5) = r9
```


```
                                   ; 0x01234567 >> 5 = 0x00091A2B
```

Example 2 Assume that the sample data for this example is the same as that for Example 1. Also assume that the registers contain the following data upon entry:

- r4 = 5 (rotate amount)
- r5 = 32 (width field value)
- r7 = 0x123 4567 (original value)
- r9 = 0xFEDC BA98 (original value)

Either of these instructions returns r9 = 0x0009 1A2B:

```
srl   5,32,r7,r9  ; Immediate form
```

or

```
srl   r4,32,r7,r9 ; Register form
```

Also, either of these instructions returns r7 = 0x07F6 E5D4:

```
srl   5,32,r9,r7  ; Immediate form
```

or

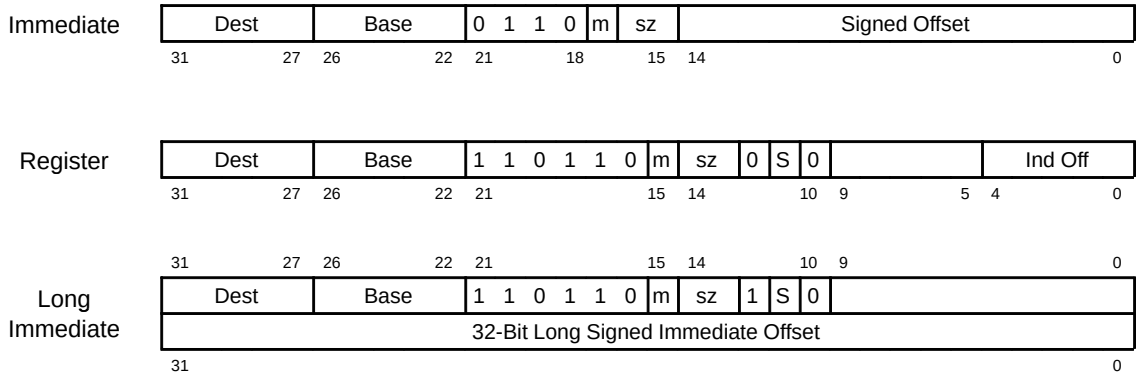
```
srl   r4,32,r9,r7 ; Register form
```

Syntax **st**[{**.b**|**.h**|**.d**}] *offset*[**:s**] (*base*[**:m**]),*source*

Operation
IF *offset*, then form (*offset* + *base*) → *addr*
IF *offset***:s**, then scale by data size as (*offset*<<*sz*) + *base* → *addr*
IF *base***:m**, then update *base* as addr → *base*
source → MEM(*addr*)

Operands
I(R),R Immediate Form (does not support **:s** scaling)
R(R),R Register Form
L(R),R Long-Immediate Form

Encoding



Notes: 1) m = Modify base address; S = scale offset by data size; sz = data size.
2) sz can have a value of 0, 1, 2, or 3. 0 indicates an 8-bit byte, 1 indicates a 16-bit halfword, 2 indicates a 32-bit word (default), and 3 indicates a 64-bit doubleword.

Description If you specify offset scaling by using the **:s** suffix, then the data in the indirect *offset* register (or long immediate data) is shifted left 0, 1, 2, or 3 for byte, halfword, word, or doubleword sizes, respectively. The word size is the default, and you can specify a byte, halfword, or doubleword by using **.b**, **.h**, or **.d** in the command syntax. See Table 10–26.

Table 10–26. Data Size and Offset Values for st Instruction

Opcode Syntax	Data Size	offset:s left shift	sz
st.b	8-bit least significant byte	0	0
st.h	16-bit least significant halfword	1	1
st	32-bit word	2	2
st.d	64-bit doubleword	3	3

The optionally scaled offset is then added to the data in the *base* register to form a memory address *addr*. If you specify modify base address with the **:m** suffix, then the sum *addr* is written back into the *base* register as the memory access is issued.

If the data at memory address *addr* is not resident, a data-cache miss is issued to the TC. Once the data is resident in data cache, the data from the least significant part (for byte or halfword) of the *source* register is written into the memory address *addr*. If the data is a 64-bit doubleword, then the data originates from an (even,odd) register pair. The 32 LSBs are read from the even *source* register, and the 32 MSBs are read from the odd *source*+1 register. See subsection 10.9.2 for additional information about using r0 and r1.

Memory is access according to the data size alignment. Half-words ignore memory address LSB, words ignore two LSBs, doubleword ignore three LSBs.

Notes:

1) Double-precision loads and stores are independent of the external big- or little-endian memory organization. Endian ordering is transparent to the program when using 64-bit loads or stores—the most significant word is always in the odd (even + 1) register, and the least significant word is always in the even register (≥ 2).

2) If *offset, base,* and *source* registers are not available, execution stalls until they are available.

3) When accessing TC or VC on-chip registers with a store instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

4) If the *source* and *base* are the same register, the original source value is written into the preincremented memory location. For example, assume r13 = 8, then

```
st 4(r13:m),r13   ; MEM(0xC) = 8 (original source)
```

stores 8 (the original *source* value) into memory location 0xC (4 + 8), and 0xC is written back as the modified *base* value.

5) Writing a 1 to a bit in the FLTSTS register resets that bit; writing a 0 has no affect.

Example 1 Memory and registers before instructions:

```
020000C0:   00000000   00000000          r7     00000006
020000C8:   00000000   00000000          r8     12345678
020000D0:   00000000   00000000          r9     9ABCDEF0
020000D8:   00000000   00000000         r10     020000C0


                                          ;     r10 after
st.d    16(r10),r8                        ;      020000C0
st      8(r10),r9                         ;      020000C0
st.h    r7(r10),r8                        ;      020000C0
st.b    3(r10:m),r8                       ;      020000C3
```

Memory after instructions:

```
            Big Endian    Little Endian
020000C0:   00005678      78000000
020000C4:   00000078      56780000
020000C8:   00000000      9ABCDEF0
020000CC:   9ABCDEF0      00000000
020000D0:   12345678      12345678
020000D4:   9ABCDEF0      9ABCDEF0

            31     0      31     0 (bit)
```

Example 2

```
addu    1,r0,r14            ; r14 = 1
addu    0x20000c8,r0,r11    ; r12 = 0x0200 00C8
addu    0xc002,r0,r20       ; r20 = 0xC002
st.h    0x20000ca(r0),r20   ; 0x20000ca = 0x9ABC C002 (big endian)
                            ; 0x20000ca = 0xC002 DEF0 (little endian)
st.h    r14:s(r11),r20      ; same result as the previous instruction
```

Example 3 Reset bits in the TC's FLTSTS register:

```
addu    1,r0,r6             ; bit 0 = 1
st      0x0182000C(r0),r6   ; Resets bit m in the FLTSTS memory-mapped
                            ; register (MP packet request fault = 0)
```

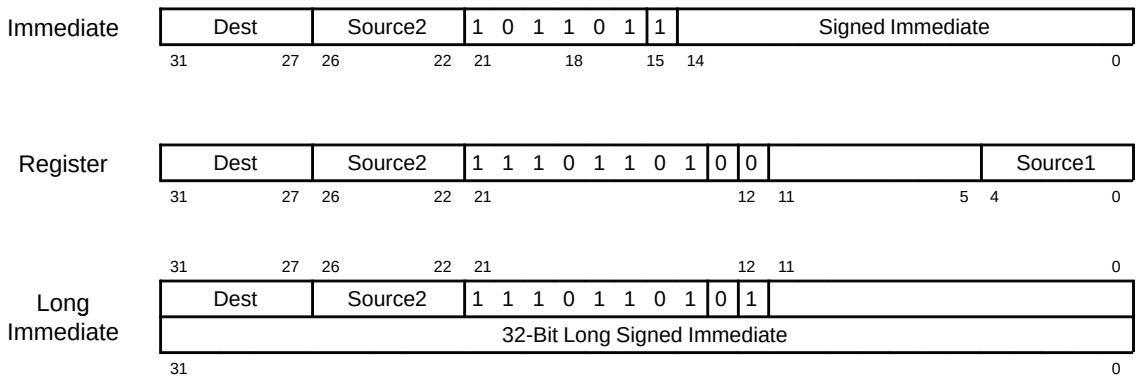
Syntax **sub** *source1,source2,dest*

Operation *source1 – source2 → dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description Data from the *source2* register is subtracted from data in the *source1* register (or the immediate data), and the result is written into the *dest* register. The order of the operands facilitates the subtraction (5 – Y). If (Y – 5) is needed, then use signed integer add –5,Y,dest. If the sub operation overflows, an integer overflow is signaled (INTPEN[[io]] is set to 1).

Example

```
                        ;     r7         r8         r9
                        ;   before     before     after
sub  r7,r8,r9           ;  12345678   12345677   00000001
sub  2,r8,r9            ;     –       00000001   00000001
sub  0x12345678,r8,r9   ;     –       12345677   00000001
sub  r7,r8,r9           ;  000000000  800000000  80000000   (overflows)
                        ;                                  INTPEN[[io]]=1
```

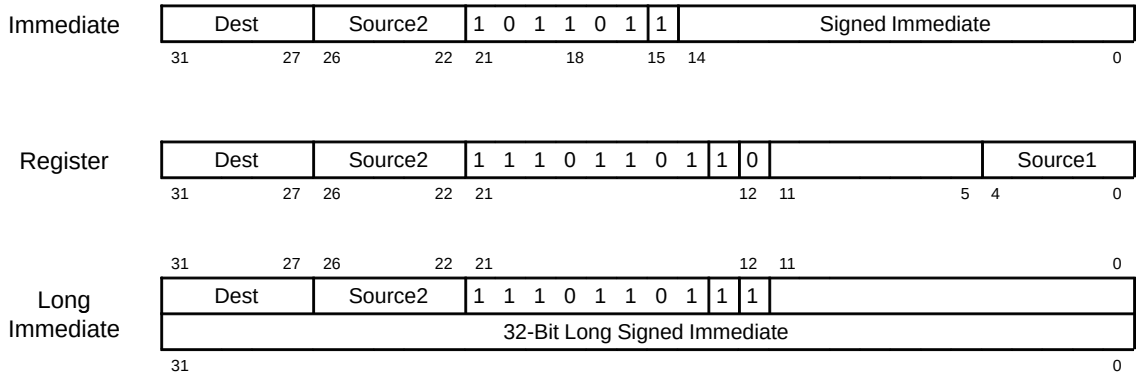
Syntax **subu** *source1,source2,dest*

Operation *source1 – source2 → dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description

Data from the *source2* register is subtracted from data in the *source1* register (or the immediate data), and the result is written into the *dest* register.

Using signed immediate data allows the instruction

subu –2,r8,r9

to be used to perform r9 = –2 – r8. Additionally,

addu 2,r8,r9

returns r9 = r8 – (–2) = r8 + 2. Thus a constant can be either operand of the "–" operator.

If the subu operation overflows, **no** integer overflow is signaled (INTPEN[io] remains unchanged).

Example

```
                        ;     r7         r8         r9
                        ;   before     before     after
subu  r7,r8,r9          ;  12345678   12345677   00000001
subu  2,r8,r9           ;     –       00000001   00000001
subu  0x12345678,r8,r9  ;     –       12345677   00000001
subu  r7,r8,r9          ; 000000000  800000000  80000000  (no overflow)
                        ;                                INTPEN[io] is
                        ;                                unchanged
```

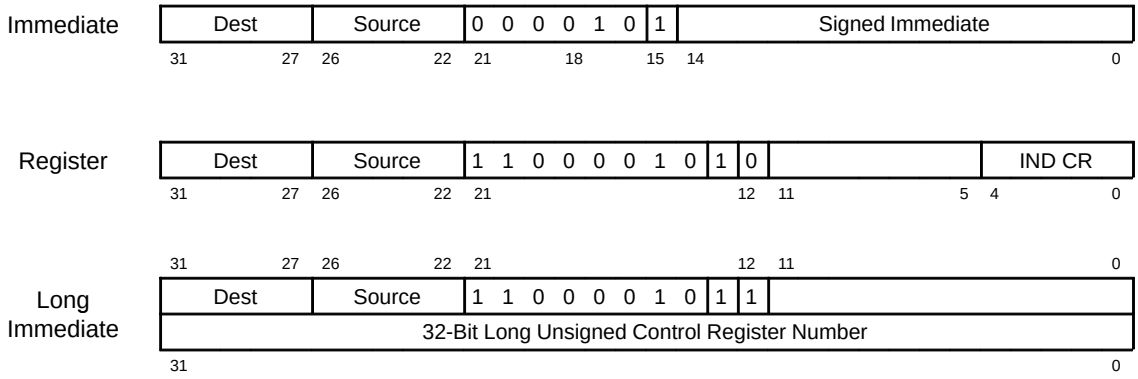
Syntax **swcr** *creg,source,dest*

Operation *creg* → *dest*
source→ *creg*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description The data from the *creg* register (or the immediate data) is used as an address to select a control register. The value in the control register is written into the *dest* register. The control register is then overwritten by the value in the *source* register. In the register form, the indirect control register number is held in the *creg* register.

This instruction can be used to change the floating-point options or interrupt enable. You can also use it as a write control register instruction (wrcr) by coding the *dest* register as r0 (writes to register r0 are discarded).

In user mode, if the control register number (*creg*) that you specify with the swcr instruction is less than 0x4000, the swcr instruction is treated as a nop instruction, and you do not receive an error signal.

If you use an undefined control register number, the operation is undefined, and you do not receive an error signal.

Notes:

1) If the floating-point pipeline is not empty, the swcr instruction stalls the MP's FEA pipeline until the floating-point pipeline is empty.

2) The MP must be in the supervisor mode to issue a swcr instruction for control registers < 0x4000.

3) For a list of control registers, see Table 2–1, *MP Control Register Numbers*.

Example 1

```
                        ;     IE          r8          r9          IE
                        ;   before      before      after       after

swcr  IE,r8,r9          ;  0000000F    12345678    0000000F    12345678

swcr  OUTP,r9,r9        ; Swap output pointer with r9
swcr  IN0P,r8,r8        ; Swap input 0 pointer with r8
swcr  IN1P,r7,r7        ; Swap input 1 pointer with r7
vldl.s r10              ; Load 32-bit r10 from MEM(IN1P)
                        ; and post-increment IN1P by 4 bytes
```

Example 2

```
addu  0x10,r0,r6        ; r6 = 0x10 (FLTOP)
swcr  r6,r0,r5          ; new FLTOP = 0 and r5 = old FLTOP
```

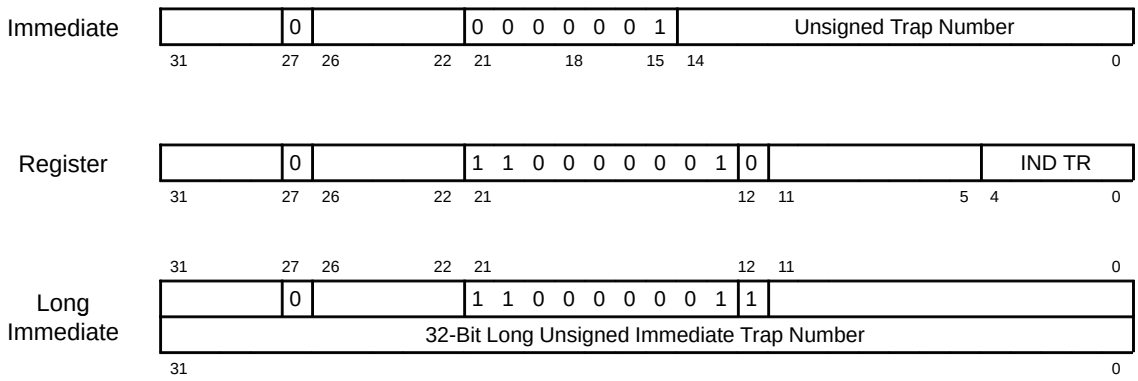
Syntax **trap** *trapnumber*

Operation MEM(0x1010180 + 4**trapnumber*) → PC

Operands

I	Immediate Form
R	Register Form
L	Long-Immediate Form

Encoding



Description The vector transfer address in the MP's parameter RAM is formed as:

vector transfer address = 0x01010180 + 4**trapnumber*

Program control is transferred to the trap service routine as:

MEM(*vector transfer address)* → *PC*

In the register form, the indirect unsigned trap number is held in the *trapnumber* register.

There is no branch delay slot following a trap instruction.

Traps are not affected by the IE register. A trap cannot be disabled, even if the corresponding interrupt is disabled.

Notes:

1) $0 \leq trapnumber \leq 415$ for transfer vectors in the MP's parameter RAM; however, $40 \leq trapnumber \leq 415$ uses much of the general-purpose MP parameter RAM and should be used judiciously.

2) *trapnumber* > 415 **must** address external RAM (≥ 0x0200 0000) to avoid getting an illegal memory address; therefore $4{,}177{,}824_{10} \leq trapnumber \leq 1{,}069{,}531{,}039_{10}$ and requires a cache access.

3) For a complete list of traps, see Figure 3–2, *MP Interrupt Registers—IE and INTPEN*, and Table 9–1, *Maskable Interrupt Priorities and Vector Addresses*.

4) Traps 40–71 are unavailable if externally initiated interrupts are enabled (CONFIG⟦X⟧ in Figure 3–7, *MP Configuration Register—CONFIG*). This is also true for VC frame timer-initiated interrupts (see FMEMCTL0/1⟦P⟧).

5) Traps are taken, even if IE⟦ie⟧ = 0.

Example The transfer vector address contains the interrupt/trap service entry address shown in Table 9–1, *Maskable Interrupt Priorities and Vector Addresses*, or Table 9–2, *Nonmaskable Trap Priorities and Vector Addresses*.

```
trap    36    ; PC =  MEM(0x1010210) =  Floating-point error (FP
              ;                         instruction used and
              ;                         interrupts were not
              ;                         enabled or the floating-
              ;                         point unit is stalled.)
trap    38    ; PC =  MEM(0x1010218) =  Illegal MP instruction
trap    79    ; PC =  MEM(0x10102BC) =  System or user trap
```

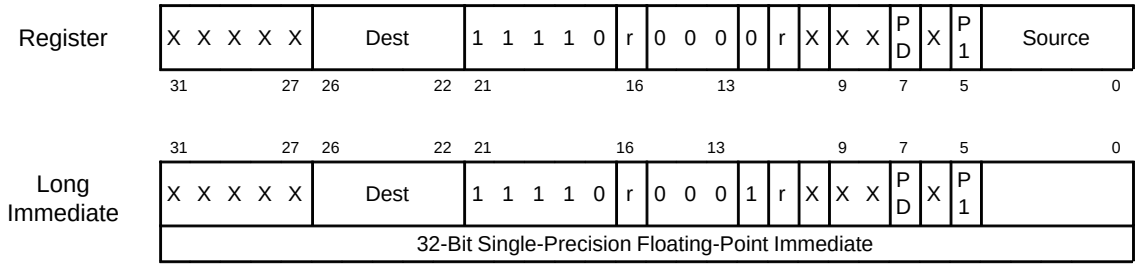
Syntax **vadd.**{**s**|**d**}{**s**|**d**} *source,dest,dest* [||≪vector load or store≫]

Operation *source* + *dest* → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; r = reserved; X = used to code vector load or store.

2) P1 and PD can each have a value of 0 or 1. For the vadd instruction, 0 indicates a single-precision floating-point value, and 1 indicates a double-precision floating-point value.

Description The vadd instruction performs a floating-point addition on the operands from the *source* register (or *immediate* data) and the *dest* register and writes the result (using the default rounding mode, set with the drm field of the FPST register) into the *dest* register.

An optional vector load or store (vld0, vld1, or vst) can be performed in parallel with the vadd instruction.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the vadd operation.

Precisions The following combinations of precisions are legal;

```
vadd.ss = SP + SP → SP
vadd.sd = SP + DP → DP
vadd.dd = DP + DP → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧=1 for the vadd instruction).

Example

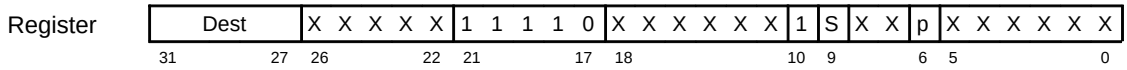
```
vadd.ss  r8,r9,r9              ; r8 + r9 = r9 (all in SP FP)
vadd.ss  r8,r9,r9  ||vld1.s r7 ; r8 + r9 = r9 (all in SP FP); also
                               ; load 32-bit r7 from MEM (IN1P) and
                               ; post-increment IN1P by 4 bytes.
vadd.dd  r6,r8,r8  ||vld1.d r6 ; r6 + r8 = r8 (all in DP FP); also
                               ; load 64-bit (r6,r7) from MEM (IN1P)
                               ; and postincrement IN1P by 8 bytes.
```

Syntax [≪vector operation≫||] **vld{0|1}.{s|d}** *dest*

Operation MEM(INxP) → *dest*
IF **vldx.s**, then INxP + 4 → INxP
IF **vldx.d**, then INxP + 8 → INxP

Operands R, implied input pointer INxP, where x = 0, 1

Encoding



Notes: 1) S = size; p = input pointer select; X = used to code vector floating-point operation.
2) S can have a value of 0 or 1. 0 indicates a size of 32 bits, and 1 indicates a size of 64 bits.
3) p can have a value of 0 or 1. 0 represents the IN0P register, and 1 represents the IN1P register.

Description Data at the memory location contained in control register INxP (x = 0 for vld0, or x = 1 for vld1 mnemonic) is loaded into the *dest* register. If the data is a 64-bit doubleword, then the data is loaded into an (even,odd) register pair. The 32 LSBs are loaded into the even *dest* register, and the 32 MSBs are loaded into the odd *dest*+1 register.

The selected INxP control register is then post incremented by

- 4 bytes for a 32-bit single word load,
- 8 bytes for a 64-bit doubleword load.

See subsection 10.9.2 for additional information about using r0 and r1.

Notes:

1) Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vld instruction).

2) Double-precision loads and stores are independent of the external big- or little-endian memory organization. Endian ordering is transparent to the program when using 64-bit loads or stores—the most significant word is always in the odd (even + 1) register, and the least significant word is always in the even register (≥2).

3) If the ≪vector operation≫ is omitted, vmpy.ss r0,r0,r0 is used as the vector nop for the vld instruction. Code should not rely on a particular pattern for the ≪vector no operation≫. (In this case, the || symbols are not required.)

4) Loads require one extra cycle before data is available for use. As the *dest* register is scoreboarded, a nop instruction is not needed; however, the program often can place a useful instruction in the scoreboard time slot.

5) If the *dest* register is not available, memory execution of the load stalls until it is available.

6) When accessing TC or VC on-chip registers with a load instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

Example

```
vld0.ss  r6                  ; vld without a vector operation,
                             ; load 32-bit r6 from MEM(IN0P),
                             ; and post-increment IN0P by 4 bytes
vadd.sd  r6,r8,r8  vld0.s r6 ; SP FP r6 is converted to DP FP,
                             ; r6 + r8 = r8 (all in DP FP),
                             ; load 32-bit r6 from MEM(IN0P),
                             ; and post-increment IN0P by 4 bytes
vadd.ss  r8,r9,r9  vld1.d r6 ; r8 + r9 = r9 (all in SP FP),
                             ; load 64-bit (r6,r7) from MEM(IN1P),
                             ; and post-increment IN1P by 8 bytes
```

Syntax **vmac.ss**{**s**|**d**} *source1,source2,*{*acc*|**0**},{*acc*|*dest*} [||≪vector load or store≫]

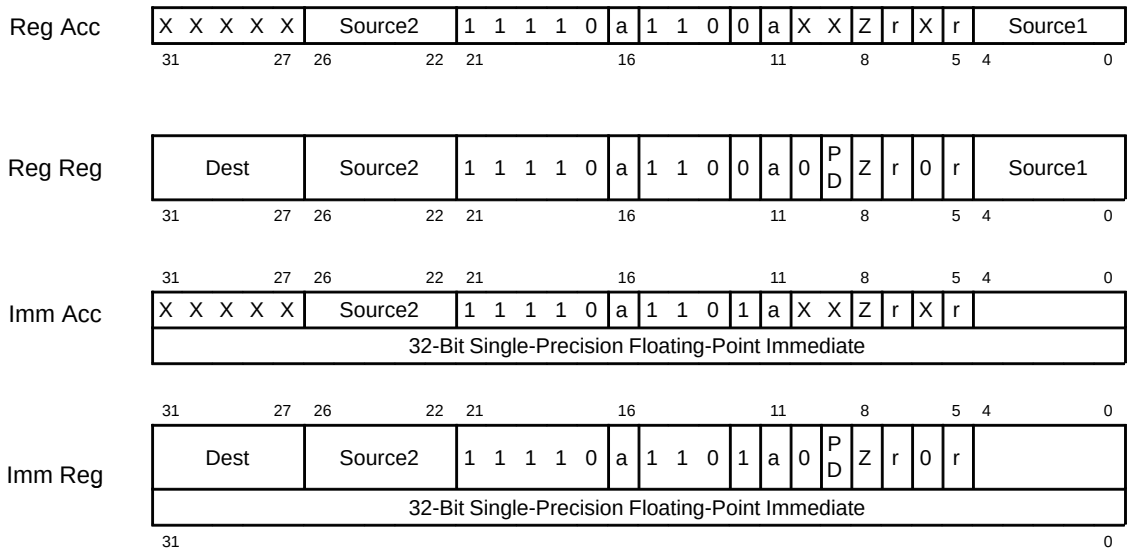
Operation

source1 × *source2* + 0.0 → *acc*
source1 × *source2* + 0.0 → *dest*
source1 × *source2* + *acc* → *acc*
source1 × *source2* + *acc* → *dest*

Operands

register	immediate
R,R,0,A	L,R,0,A
R,R,0,R	L,R,0,R
R,R,A,A	L,R,A,A
R,R,A,R	L,R,A,R

Encoding



Notes: 1) PD = *dest* precision; r = reserved; X = used to code vector load or store; Z = add to zero rather than double-precision floating-point accumulator; a = double-precision floating-point accumulator select, (MSB = 16, LSB = 11)

2) PD can have a value of 0 or 1. For the vmac instruction, 0 indicates a single-precision floating-point value, and 1 indicates a double-precision floating-point value.

Description Single-precision floating-point operands from the *source1* register (or the immediate data) and the *source2* register are multiplied to form a double-precision floating-point product. This double-precision floating-point product is added to double-precision 0 or the contents of double-precision floating-point accumulator (*acc*—a0, a1, a2, or a3) with the result (using the default rounding mode, set with the drm field of the FPST register) written into the *dest* register or accumulator. Note that vmac.sss uses round-to-zero mode only.

If a parallel vector load or store (vld0, vld1, or vst) is included, the result of the add must be a double-precision floating-point accumulator (*acc*). If no parallel vector load or store (vld0, vld1, or vst) is included, the result of the add can be written to a single-precision or double-precision floating-point *dest* register or to an accumulator. If the destination is single precision, then the result must be written to a *dest* register (not to an accumulator).

Latency

Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the vmac operation.

Precisions

The following combinations of precisions are legal (DP is double-precision, and SP is single-precision):

- with VLD or VST:

```
vmac.ssd = SP × SP + DP → DP (dest must be acc)
```

- no VLD or VST:

```
vmac.sss = SP × SP + DP → SP  (dest must be a
                               register ≠ r1)
vmac.ssd = SP × SP + DP → DP  (dest is acc or even
                               register ≠ r0)
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vmac instruction).

Example

```
vmac.sss  r8,r9,0,r6              ; r8 * r9 + 0.0 = r6
                                  ; SP * SP = DP FP product
                                  ; + DP 0.0 = DP FP sum which
                                  ; is converted to SP FP r6

vmac.ssd  r8,r9,a1,r6             ; r8 * r9 + al = r6
                                  ; SP * SP = DP FP product
                                  ; + DP FP accumulator a1 which
                                  ; results in DP FP sum in (r6,r7)

vmac.ssd  r8,r9,0,a2              ; r8 * r9 + 0.0 = a2
                                  ; SP * SP = DP FP product
                                  ; + DP 0.0 = DP FP sum in a2

vmac.ssd  r8,r9,a3,a3             ; r8 * r9 + a3 = a3
                                  ; SP * SP = DP FP product
                                  ; + DP FP accumulator a3 which
                                  ; results in DP FP sum in a3

vmac.ssd  r6,r8,a0,a0   ||vst.s r6 ; r6 * r8 + a0 = a0
                                  ; SP * SP = DP FP product
                                  ; + DP FP accumulator a0 which
                                  ; results in DP FP sum in a0,
                                  ; also store 32-bit r6 into MEM(OUTP)
                                  ; post-increment OUTP by 4 bytes
```

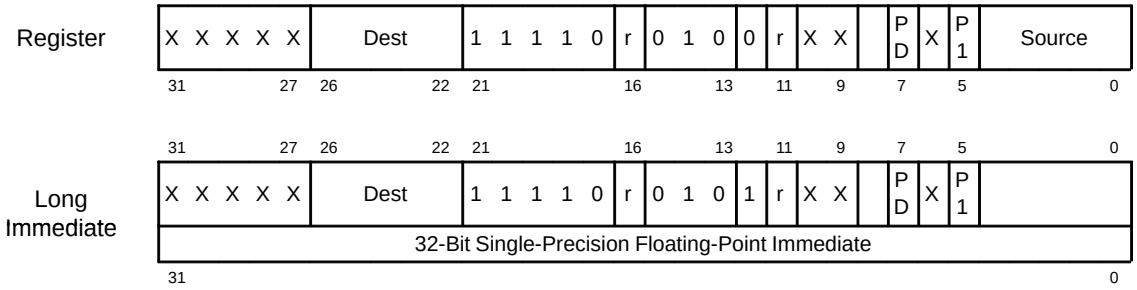
Syntax **vmpy.**{**s**|**d**}{**s**|**d**} *source,dest,dest* [||≪vector load or store≫]

Operation *source* × *dest* → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; r = reserved; X = used to code vector load or store.

2) P1 and PD can each have a value of 0 or 1. For the vmpy instruction, 0 indicates a single-precision floating-point value, and 1 indicates a double-precision floating-point value.

Description The operands in the *source* register (or the *immediate* data) and the *dest* register are multiplied as floating-point numbers, and the result (using the default rounding mode, set with the drm field of the FPST register) is written into the *dest* register.

An optional vector load or store (vld0, vld1, or vst) can be performed in parallel with the vmpy instruction.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the vmpy operation.

Precisions The following combinations of precisions are legal:

```
vmpy.ss = SP × SP → SP
vmpy.sd = SP × DP → DP
vmpy.dd = DP × DP → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vmpy instruction).

Example

```
vmpy.ss   r0,r0,r0                ; 0 * 0 = r0 (discarded)
                                  ; may be used as a vector ”nop”

vmpy.ss   r8,r9,r9                ; r8 * r9 = r9 (all in SP FP format)

vmpy.ss   r8,r9,r9     ||vld1.s r7 ; r8 * r9 = r9 (all in SP FP format),
                                  ; load 32-bit r7 from MEM(IN1P),
                                  ; post-increment IN1P by 4 bytes.

vmpy.sd   –0.75,r8,r8  ||vst.d r6  ; Converts SP –0.75 to DP FP, then
                                  ; –0.75 * r8 = r8 (all in DP FP),
                                  ; store 64-bit (r6,r7) into MEM(OUTP),
                                  ; post-increment OUTP by 8 bytes.
```

Syntax **vmsc.ss**{**s**|**d**} *source1*,*source2*,{*acc*|**0**},{*acc*|*dest*} [||≪vector load or store≫]

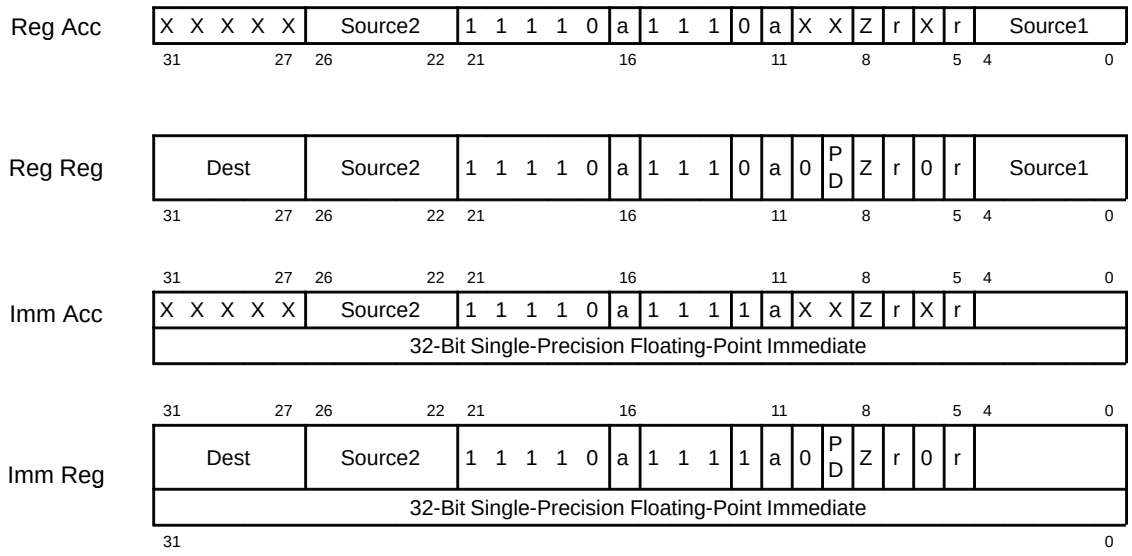
Operation

0.0 – *source1* × *source2* → *acc*
0.0 – *source1* × *source2* → *dest*
acc – *source1* × *source2* → *acc*
acc – *source1* × *source2* → *dest*

Operands

register	immediate
R,R,0,A	L,R,0,A
R,R,0,R	L,R,0,R
R,R,A,A	L,R,A,A
R,R,A,R	L,R,A,R

Encoding



Notes: 1) PD = *dest* precision; r = reserved; X = used to code vector load or store; Z = subtract from zero rather than double-precision floating-point accumulator; a = double-precision floating-point accumulator select, (MSB = 16, LSB = 11).

2) PD can have a value of 0 or 1. For the vmsc instruction, 0 indicates a single-precision floating-point value, and 1 indicates a double-precision floating-point value.

Description Single-precision floating-point operands from the *source1* register (or the immediate data) and the *source2* register are multiplied to form a double-precision product. This double-precision floating-point product is subtracted from double-precision 0 or the contents of double-precision floating-point accumulator (*acc*—a0, a1, a2, or a3) with the result (using the default rounding mode, set with the drm field of the FPST register) written into the *dest* register or accumulator. Note that vmsc.sss uses round-to-zero mode only.

If a parallel vector load or store (vld0, vld1, or vst) is included, the result of the subtract must be a double-precision floating-point accumulator (*acc*). If no parallel vector load or store (vld0, vld1, or vst) is included, the result of the subtract can be written to a single-precision or double-precision floating-point *dest* register or to an accumulator. If the destination is single precision, then the result must be written to a *dest* register (not to an accumulator).

Latency

Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the vmsc operation.

Precisions

The following combinations of precisions are legal (DP is double-precision, and SP is single-precision):

- with VLD or VST:

```
vmsc.ssd = DP – SP × SP → DP (dest must be acc)
```

- no VLD or VST:

```
vmsc.sss = DP – SP × SP → SP  (dest must be a
                               register ≠ r1)
vmsc.ssd = DP – SP × SP → DP  (dest is acc or even
                               register ≠  r0)
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vmsc instruction).

Example

```
vmsc.sss  r8,r9,0,r6              ; 0.0 – r8 * r9 = r6
                                  ; DP 0 – DP FP product where
                                  ; DP FP product = SP * SP
                                  ; to form DP FP difference
                                  ; and converted to SP FP r6

vmsc.ssd  r8,r9,a1,r6             ; a1 – r8 * r9 = r6
                                  ; DP FP a1 – DP FP product where
                                  ; DP FP product = SP * SP
                                  ; to form DP FP difference
                                  ; and store into DP FP (r6,r7)

vmsc.ssd  r8,r9,0,a2              ; 0.0 – r8 * r9 = a2
                                  ; DP 0 – DP FP product where
                                  ; DP FP product = SP * SP
                                  ; to form DP FP difference
                                  ; and store into DP FP a2

vmsc.ssd  r8,r9,a3,a3             ; a3 – r8 * r9 = a3
                                  ; DP FP a3 – DP FP product where
                                  ; DP FP product = SP * SP
                                  ; to form DP FP difference
                                  ; and store into DP FP a3

vmsc.ssd  r6,r8,a0,a0   ||vst.s r6  ; a0 – r6 * r8 = a0,
                                  ; DP FP a0 – DP FP product where
                                  ; DP FP product = SP * SP
                                  ; to form DP FP difference
                                  ; and store into DP FP a0,
                                  ; store 32-bit r6 into MEM(OUTP)
                                  ; post-increment OUTP by 4 bytes.
```

Syntax **vmsub.s**{**s**|**d**} *source*,{*acc*|**0**},{*acc*|*dest*} [||≪vector load or store≫]

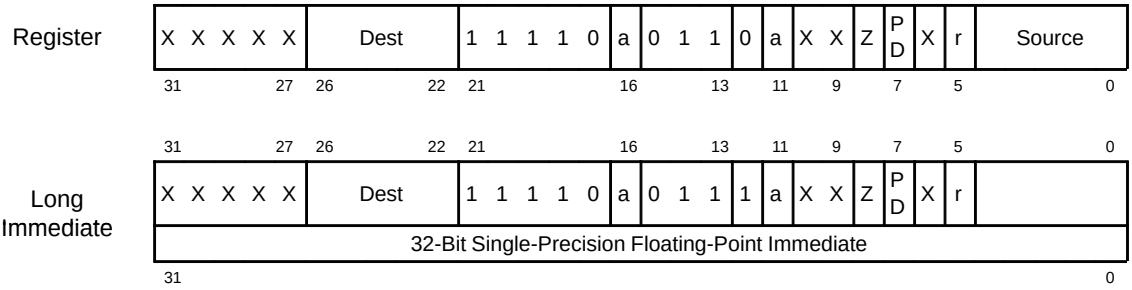
Operation

source × 1.0 – 0.0 → *acc*
source × 1.0 – 0.0 → *dest*
source × 1.0 – *acc* → *acc*
source × 1.0 – *acc* → *dest*

Operands

R,A,A	Register Form
L,A,A	Long-Immediate Form
R,A,R	Register Form
L,A,R	Long-Immediate Form

Encoding



Notes: 1) PD = *dest* precision; r = reserved; X = used to code vector load or store; Z = use zero rather than double-precision floating-point accumulator; a = double-precision floating-point accumulator select, (MSB = 16, LSB = 11).

2) PD can have a value of 0 or 1. For the vmsub instruction, 0 indicates a single-precision floating-point value, and 1 indicates a double-precision floating-point value.

Description

The single-precision floating-point data in the *source* register (or the immediate data) is multiplied by a single-precision 1 to form a double-precision floating-point result. A double-precision 0 or the contents of the double-precision floating-point accumulator (*acc*—a0, a1, a2, or a3) is subtracted from this double-precision floating-point product with the result (using the default rounding mode, set with the drm field of the FPST register) written into the *dest* register or accumulator. Note that vmsub.ss uses round-to-zero mode only.

You can perform an optional vector load or store (vld0, vld1, or vst) in parallel with the vmsub instruction. However, unlike vmac and vmsc, you can specify a *dest* register with a parallel load or store. Note that if the destination is single precision, then the result must be written to a *dest* register (not to an accumulator).

Latency

Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the vmsub operation.

Precisions

The following combinations of precisions are legal:

```
vmsub.ss = SP × 1.0 – DP → SP
vmsub.sd = SP × 1.0 – DP → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

> **Note:**
>
> Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vmsub instruction).

Example

```
vmsub.ss  r8,a0,r9                 ; r8 * 1.0 – a0 = r9
                                   ; SP * SP = DP FP product
                                   ; – DP FP a0 = DP FP difference
                                   ; and converted to SP FP r9.

vmsub.sd  –0.75,a0,r8  ||vst.s r6  ; –0.75 * 1.0 – a0 = r8
                                   ; SP * SP = DP FP product
                                   ; – DP FP a0 = DP FP difference
                                   ; and stored into DP FP (r8,r9),
                                   ; store 32-bit r6 into MEM(OUTP),
                                   ; post-increment OUTP by 4 bytes.

vmsub.sd  r7,a1,a1     ||vld1.d r8 ; r7 * 1.0 – a1 = a1
                                   ; SP * SP = DP FP product
                                   ; – DP FP a1 = DP FP difference
                                   ; and store into DP FP a1,
                                   ; load 64-bit (r8,r9) from MEM(IN1P),
                                   ; post-increment IN1P by 8 bytes.
```

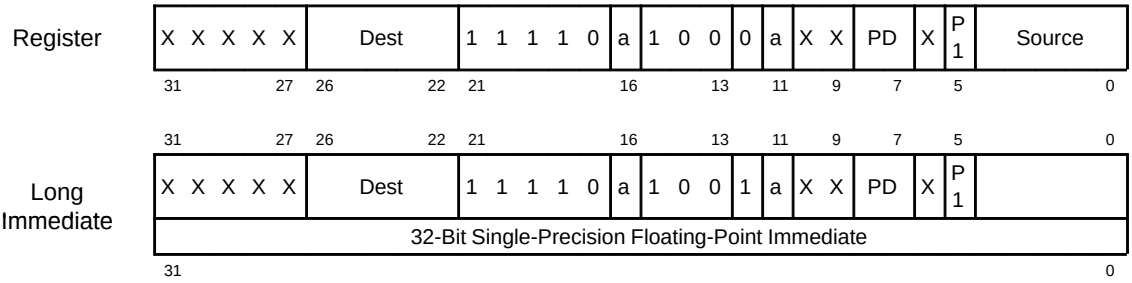
Syntax **vrnd.**{**s**|**d**}{**s**|**d**|**i**|**u**} *source*,{*acc*|*dest*} [||≪vector load or store≫]

Operation converted(*source*) → *dest*

Operands

R,R	Register Form
L,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; X = used to code vector load or store; a = double-precision floating-point accumulator select, (MSB = 16, LSB = 11).

2) P1 can have a value of 0 or 1; PD can have a value of 0, 1, 2, or 3. For the vrnd instruction, 0 indicates a single-precision floating-point value, 1 indicates a double-precision floating-point value, 2 indicates a 32-bit signed integer, and 3 indicates a 32-bit unsigned integer.

Description The floating-point data in the *source* register (or the immediate data) is converted to the precision requested and stored (using the default rounding mode, set with the drm field of the FPST register) into the *dest* register. The *dest* value in the mnemonic can specify either a double-precision floating-point accumulator (*acc*—a0, a1, a2, or a3) or a single-precision or double-precision floating-point register.

An optional vector load or store (vld0, vld1, or vst) can be performed in parallel with the vrnd instruction.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations*, for the latency of the vrnd operation.

Precisions

The following combinations of precisions are legal:

```
vrnd.si = SP → I    vrnd.di = DP → I  (dest = register)
vrnd.su = SP → U    vrnd.du = DP → U  (dest = register)
vrnd.ss = SP → SP   vrnd.ds = DP → SP (dest = register)
vrnd.sd = SP → DP   vrnd.dd = DP → DP (dest = acc, or
                                       even register ≠
                                       r0)
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

Note:

Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vrnd instruction).

Example

```
                                    ; Reload DP FP accumulator :
vrnd.dd   r8,a1                     ; DP FP (r8,r9) is rounded to DP FP
                                    ; and stored into DP FP a1.

vrnd.sd   –0.75,a0    ||vst.d r6    ; SP –0.75 is rounded to DP FP
                                    ; and stored into DP FP a0,
                                    ; store 64-bit (r6,r7) into
                                      MEM(OUTP),
                                    ; post-increment OUTP by 8 bytes.

vrnd.si   r8,r9       ||vld1.s r7   ; SP FP r8 is rounded to signed
                                    ; integer and stored into r9,
                                    ; load 32-bit r7 from MEM(IN1P),
                                    ; post-increment IN1P by 4 bytes.
```

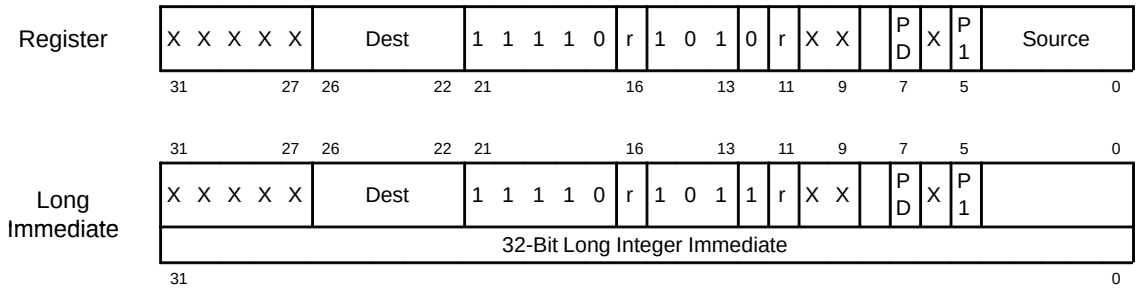
Syntax **vrnd.**{**i**|**u**}{**s**|**d**} *source,dest* [||≪vector load or store≫]

Operation converted(*source*) → *dest*

Operands

R,R	Register Form
L,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; X = used to code vector load or store; r = reserved.

2) P1 can have a value of 0 or 1; 0 indicates a 32-bit signed integer, and 1 indicates a 32-bit unsigned integer. PD can have a value of 0 or 1; 0 indicates a single-precision floating-point value, and 1 indicates a double-precision floating-point value.

Description Integer data in the *source* register (or the *immediate* data) is converted from signed or unsigned integer to the floating-point precision requested and written (using the default rounding mode, set with the drm field of the FPST register) into the *dest* register.

An optional vector load or store (vld0, vld1, or vst) can be performed in parallel with the vrnd instruction.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations* for the latency of the vrnd operation.

Precisions

The following combinations of precisions are legal:

```
vrnd.is = I → SP    vrnd.us = U → SP
vrnd.id = I → DP    vrnd.ud = U → DP
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

> **Note:**
>
> Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vrnd instruction).

Example

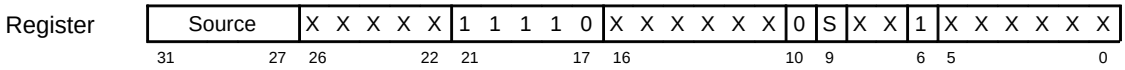
```
vrnd.id    123,r8     ||vst.d r6   ; DP round of signed integer 123
                                   ; and store DP FP into DP FP (r8,r9),
                                   ; store 64-bit (r6,r7) into MEM(OUTP),
                                   ; post-increment OUTP by 8 bytes.

vrnd.us    r8,r9      ||vld1.s r7  ; SP round of r8 unsigned integer
                                   ; and store SP FP into r9,
                                   ; load 32-bit r7 from MEM(IN1P),
                                   ; post-increment IN1P by 4 bytes.
```

Syntax [≪vector operation≫||] **vst.{s|d}** *source*

Operation *source* → MEM(OUTP)
IF **vst.s**, then OUTP + 4 → OUTP
IF **vst.d**, then OUTP + 8 → OUTP

Operands R, implied output pointer OUTP

Encoding



Notes: 1) S = size; X = used to code vector floating-point operation.
2) S can have a value of 0 or 1. 0 indicates a size of 32 bits, and 1 indicates a size of 64 bits.

Description Data in the *source* register is stored into the memory location contained in the control register OUTP. If the data is a 64-bit doubleword, then the data is read from an (even,odd) register pair. The 32 LSBs are read from the even *source* register, and the 32 MSBs are read from the odd *source*+1 register.

The OUTP control register is then post incremented by:

- 4 bytes for a 32-bit word store
- 8 bytes for a 64-bit doubleword store

See subsection 10.9.2 for additional information about using r0 and r1.

Notes:

1) Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vst instruction).

2) Double-precision loads and stores are independent of the external big- or little-endian memory organization. Endian ordering is transparent to the program when using 64-bit loads or stores—the most significant word is always in the odd (even + 1) register, and the least significant word is always in the even register (≥2).

3) If the ≪vector operation≫ is omitted, vmpy.ss r0,r0,r0 is used as the vector nop for the vst instruction. Code should not rely on a particular pattern for the ≪vector no-operation≫. (In this case, the || symbols are not required.)

4) If the *source* register is not available, memory execution of the store stalls until it is available.

5) When accessing TC or VC on-chip registers with a store instruction, you cannot use a 64-bit doubleword data size. For more information, see subsection 5.1.2, *Accessing TC and VC On-Chip Registers.*

Example

```
vst.s r6                          ; vst without a vector operation:
                                  ; store 32-bit r6 into MEM (OUTP),
                                  ; post-increment OUTP by 4 bytes.

vadd.sd   r6,r8,r8  ||vst.s r7    ; SP FP r6 is converted to DP FP,
                                  ; r6 + r8 = r8 (all in DP FP),
                                  ; store 32-bit r7 into MEM(OUTP),
                                  ; post-increment OUTP by 4 bytes.

vadd.ss   r8,r9,r9  ||vst.d r6    ; r8 + r9 = r9 (all in SP FP),
                                  ; store 64-bit (r6,r7) into MEM(OUTP),
                                  ; post-increment OUTP by 8 bytes.
```

Syntax **vsub.**{**s**|**d**}{**s**|**d**} *dest,source,dest* [||≪vector load or store≫]

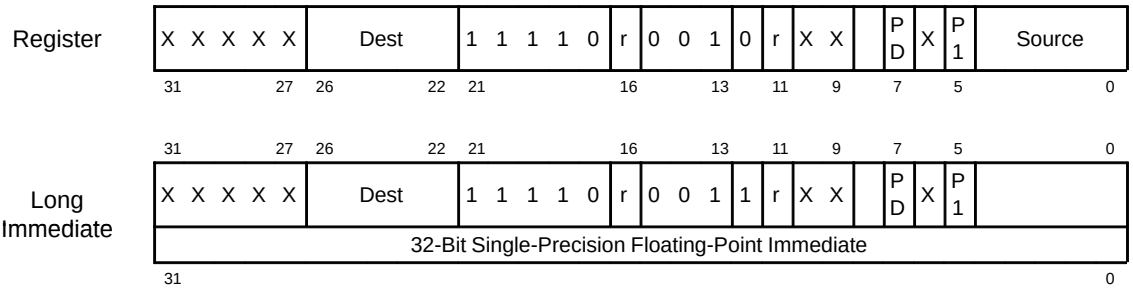
> **Note:**
>
> The operand order differs from other instructions—namely, *dest* is first, then *source* is next.

Operation *dest* – *source* → *dest*

Operands

R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Notes: 1) P1 = *source* precision; PD = *dest* precision; X = used to code vector load or store; r = reserved.

2) P1 and PD can each have a value of 0 or 1. For the vsub instruction, 0 indicates a single-precision floating-point value, and 1 indicates a double-precision floating-point value.

Description Data in the *source* register (or the immediate data) is subtracted from data in the *dest* register as floating-point numbers. The result (using the default rounding mode, set with the drm field of the FPST register) is written into the *dest* register.

An optional vector load or store (vld0, vld1, or vst) can be performed in parallel with the vsub instruction.

Latency Refer to Table 8–2, *Latencies of Floating-Point Operations* for the latency of the vsub operation.

Precisions The following combinations of precisions are legal:

```
vsub.ss = SP – SP → SP (dest = s, source = s)
vsub.sd = DP – SP → DP (dest = d, source = s)
vsub.dd = DP – DP → DP (dest = d, source = d)
```

Refer to subsections 10.9.2 and 10.9.3 for information about register use with floating-point operations.

> **Note:**
>
> Interrupts must be enabled to perform floating-point operations (IE⟦ie⟧ = 1 for the vsub instruction).

Example

```
vsub.dd   r6,r8,r6               ; r6 – r8 = r6 (all in DP FP)

vsub.sd   r8,0.75,r8  ||vst.s r6 ; Convert SP 0.75 to DP FP,
                                 ; r8 – 0.75 = r8 (all in DP FP),
                                 ; store 32-bit r6 into MEM(OUTP),
                                 ; post-increment OUTP by 4 bytes.

vsub.ss   r7,r6,r7    ||vld1.d r8 ; r7 – r6 = r7 (all in SP FP),
                                 ; load 64-bit (r8,r9) from MEM(IN1P),
                                 ; post-increment IN1P by 8 bytes.
```

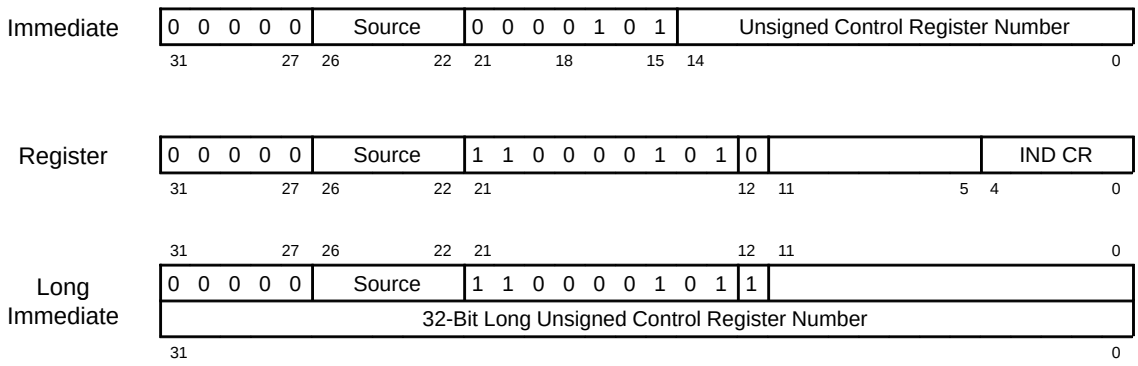
Syntax **wrcr** *creg,source*

Operation *source* → *creg*

Operands

I,R	Immediate Form
R,R	Register Form
L,R	Long-Immediate Form

Encoding



Description

The data from the *creg* register (or the *immediate* data) is used as an address to select a control register. The control register is then overwritten by the value in the *source* register. In the register form, the indirect control register number is held in the *creg* register. This instruction is a variation of **swcr** *creg,source*,r0.

This instruction can be used to change the floating-point options (FPST) or interrupt enable (IE). Additionally, it is used to turn off bits in the interrupt pending (INTPEN) register by writing a 1 to the selected bit(s); writing 0s has no affect.

In user mode, if the control register number that you specify with the wrcr instruction is less than 0x4000, the wrcr instruction is treated as a nop instruction, and you do not receive an error signal.

If you use an undefined control register number, the operation is undefined, and you do not receive an error signal.

Notes:

1) If the floating-point pipeline is not empty, the wrcr instruction stalls the MP's FEA pipeline until the floating-point pipeline is empty.
2) For a list of control registers, see Table 2–1, *MP Control Register Numbers*.
3) The MP must be in the supervisor mode to issue the wrcr instruction for control registers < 0x4000.
4) If *creg* = INTPEN, then writing a 1 to a bit from *source* will clear that bit in the INTPEN register; writing a 0 leaves the INTPEN bit unchanged.

Example 1

```
                                      IE        r8        IE
                                ;    before    before    after

wrcr  IE,r8                     ;   0000000F  12345678  12345678

wrcr  IN0P,r7                   ;   Input 0 Pointer = address in r7
wrcr  IN1P,r8                   ;   Input 1 Pointer = address in r8
wrcr  OUTP,r9                   ;   Output Pointer = address in r9
vst.s r10                       ;   Store 32-bit r10 into MEM(OUTP)
                                ;   and post-increment OUTP by 4 bytes.
```

Example 2

```
addu  0x20,r0,r7                ;   Set r7 = 0x20
wrcr  INTPEN,r7                 ;   Clears bit 5 in INTPEN (floating-point
                                ;   overflow)
```

Example 3

```
addu  0xF,r0,r6                 ;   r6 = 0xF (TSCALE)
addu  500,r0,r5                 ;   r5 = 500
wrcr  r6,r5                     ;   TSCALE = 500
```

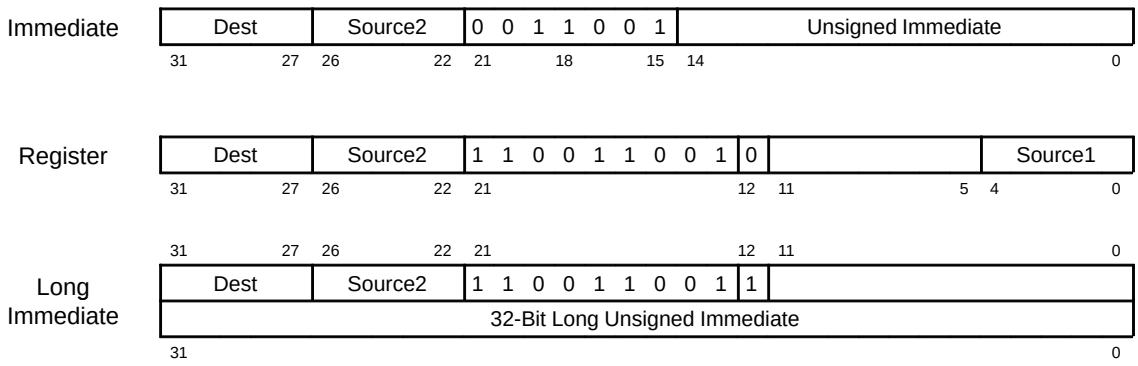
Syntax **xnor** *source1,source2,dest*

Operation *source1* XNOR *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description The bitwise exclusive-NOR of data from the *source1* register (or the immediate data) and data from the *source2* register is written into the *dest* register. In the register form, the indirect unsigned *source1* operand is held in the *source1* register.

Example

```
                                        ;       r7          r8          r9
                                        ;    before      before       after

xnor    r7,r8,r9                        ;   0000000F    12345678    EDCBA988

xnor    0x7FFF,r8,r9                    ;       -       12345678    EDCBD678

xnor    0xFF0000FF,r8,r9                ;       -       12345678    12CBA978
```

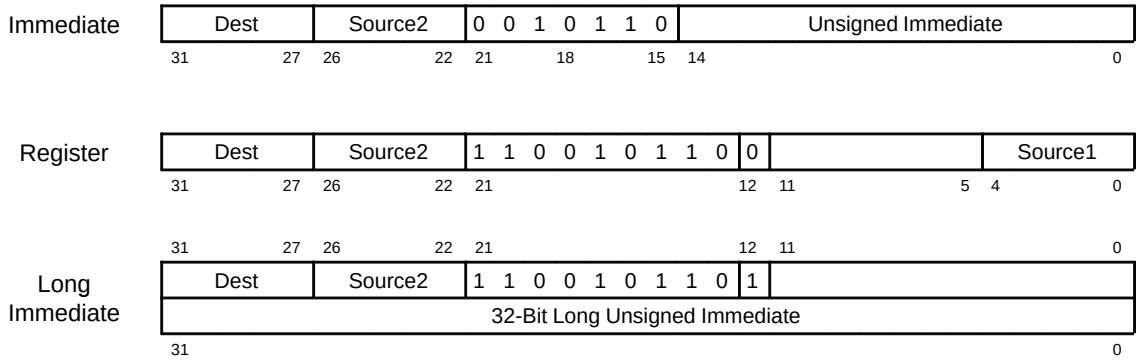
Syntax **xor** *source1,source2,dest*

Operation *source1* XOR *source2* → *dest*

Operands

I,R,R	Immediate Form
R,R,R	Register Form
L,R,R	Long-Immediate Form

Encoding



Description The bitwise exclusive-OR of data from the *source1* register (or the immediate data) and data from the *source2* register is written into the *dest* register. In the register form, the indirect unsigned *source1* operand is held in the *source1* register.

Example

```
                                    ;       r7          r8          r9
                                    ;    before      before       after

xor     r7,r8,r9                    ;  0000000F    12345678    12345677

xor     0x7FFF,r8,r9                ;       -      12345678    12342987

xor     0xFF0000FF,r8,r9            ;       -      12345678    ED345687
```

Chapter 11

Master Processor Applications

The sample applications in this chapter illustrate various MP operations. All timings listed assume that there are no cache misses, data-bank contentions, round-robin contentions, input or output exceptions, NaNs (not-a-numbers), denormal values, infinities, traps, interrupts, transfer controller waits, or instruction-cache loads required.

Topics

11.1 Call and Return From a Subroutine

Example 11–1 illustrates how to call the SUBR subroutine with a single-precision floating-point argument Z. Note that R31 is a link and R1 is the stack pointer.

Example 11–1. Call and Return From a Subroutine

```
Main:
       frndn.is  –75,r2          ; Z = –75 is passed in r2
       bsr.a     SUBR,r31        ; Label SUBR, r31 = link, and nop is
                                 ; executed in the branch delay slot
       st        Ans(r0),r2      ; SUBR returns to this st
. . .
SUBR:
       st.d      –2*4(r1:m),r30  ; push 64-bit r30 onto stack (r1–8),
                                 ; then update r1 = r1–8 bytes. This
                                 ; saves the return link.
       .
       .
       .
       addu      Answer,r0,r2    ; returns answer in r2
       ld.d      0(r1),r30       ; pop 64-bit r30 from the stack. This
                                 ; restores the return link
       jsr       r31(r0),r0      ; return to calling program,
       addu      2*4,r1,r1       ; restore stack pointer in the branch
                                 ; delay slot
```

Notes: 1) Interrupts must be enabled before any floating-point operations are issued (IE⟦ie⟧ = 1).

2) The stack pointer (R1) is on an 8-byte boundary by software convention.

11.2 Conversions

Conversions from signed and unsigned integers to or from single-precision or double-precision floating point are often useful. The MP supports most (but not all) possible combinations. Some examples of conversions are shown in Example 11–2. For more information about the individual instructions, see Section 10.12, *Alphabetical Instruction Reference*.

Example 11–2. Conversions—frnd, vrnd

```
frndn.is  –75,r25          ; signed integer –75 to single-precision
                           ; floating-point in r25
frndz.ud  0x87654321,r24   ; unsigned integer 0x8765 4321 to double-
                           ; precision floating-point r24
frndp.ds  r24,r26          ; round double-precision floating-point r24 to
                           ; +infinity and convert to single-precision
                           ; floating-point in r26
vrnd.dd   r24,a1           ; restore double-precision floating-point a1
                           ; from double-precision floating-point r24
vrnd.ds   r24,r9           ; round double-precision floating-point r24
                           ; using FPST[drm] and convert to single-
                           ; precision floating-point in r9
```

Notes: 1) Interrupts must be enabled before any floating-point operations are issued (IE[ie] = 1).

2) Integer-overflow detection is shown in Example 11–3.

11.3 Integer Multiply, Divide, and Remainder

The MP uses floating-point divide and conversions to affect integer division and remainder operations (note how integer multiply instructions fmpy.iii and fmpy.uuu are used). The signed integer and unsigned integer cases are shown in Example 11–4. Notice that the conversions use **d** (not **s**) as the double-precision floating-point mantissa, which is 52 bits (not 24 bits, as found in single-precision floating point), because the integer has 32 bits of significance.

To determine if an integer multiply has overflowed, consider Example 11–3. This also includes tests for floating-point conversion to integer overflow tests.

Integer divide and remainder are shown in Example 11–4.

Example 11–3. Test Integer Overflow Sample Code

(a) Integer-Multiply Overflow (FPST⟦mo⟧ = 1)

```
fmpy.iii  r5,r6,r7    ; r5 * r6 –> r7 signed integer multiply
add       r7,r0,r0    ; wait for multiply to complete
rdcr      FPST,r4     ; floating-point unit completed multiply
bbo.a     INTOV,r4,5  ; FPST bit 5 = 1 is integer multiply overflow
fmpy.uuu  r8,r9,r10   ; r8 * r9 –> r10 unsigned integer multiply
add       r10,r0,r0   ; wait for multiply to complete
rdcr      FPST,r4     ; floating-point unit completed multiply
bbo.a     INTOV,r4,5  ; FPST bit 5 = 1 is integer multiply overflow
```

(b) Floating-Point Conversion to Integer Overflow (FPST⟦i⟧ = 1)

```
frndn.di  r6,r8       ; double-precision floating-point r6 –> signed
                      ; integer r8
add       r8,r0,r0    ; wait for conversion to complete
rdcr      FPST,r4     ; floating-point unit completed conversion
bbo.a     INTOV,r4,4  ; FPST bit 4 = 1 is invalid conversion
frndn.su  r9,r10      ; single-precision floating-point r9 –> unsigned
                      ; integer r10
add       r10,r0,r0   ; wait for conversion to complete
rdcr      FPST,r4     ; floating-point unit completed conversion
bbo.a     INTOV,r4,4  ; FPST bit 4 = 1 is invalid conversion
```

Notes: 1) Interrupts must be enabled before any floating-point operations are issued (IE⟦ie⟧ = 1).

2) Integer multiply uses the fmpy instruction.

Example 11–4. Integer Multiply, Divide, and Remainder

(a) Signed Integer Multiply

```
fmpy.iii  12345,r5,r6     ; signed integer multiply as 32 LSBs of
                          ; (12345 * r5) = r6
```

(b) Unsigned Integer Multiply

```
fmpy.uuu  r7,r8,r9        ; unsigned integer multiply as 32 LSBs of
                          ; (r7 * r8) = r9
```

(c) Signed Integer Divide

```
frndz.id  Num,Fnum        ; numerator integer to double-precision
                          ; floating-point
frndz.id  Div,Fdiv        ; divisor integer to double-precision
                          ; floating-point
fdiv.ddd  Fnum,Fdiv,Fquot ; quotient in double-precision floating-point
frndz.di  Fquot,Quot      ; double-precision floating-point to integer
                          ; quotient
```

(d) Unsigned Integer Divide

```
frndz.ud  Num,Fnum        ; unsigned numerator to double-precision
                          ; floating-point
frndz.ud  Div,Fdiv        ; unsigned divisor to double-precision
                          ; floating-point
fdiv.ddd  Fnum,Fdiv,Fquot ; quotient in double-precision floating-point
frndz.du  Fquot,Quot      ; double-precision floating-point to unsigned
                          ; quotient
```

(e) Signed Integer Remainder

```
frndz.id  Num,Fnum        ; numerator integer to double-precision
                          ; floating-point
frndz.id  Div,Fdiv        ; divisor integer to double-precision
                          ; floating-point
fdiv.ddd  Fnum,Fdiv,Fquot ; quotient in double-precision floating-point
frndz.di  Fquot,Quot      ; double-precision floating-point to integer
                          ; quotient
fmpy.iii  Div,Quot,IDxQ   ; product of integer divisor and quotient
sub       Num,IDxQ,Rem    ; signed integer remainder = Num – Div * Quot
```

Notes: 1) Other rounding modes can be used (such as frndn for nearest), rather than the truncation in the frndz instructions.

2) Interrupts must be enabled before any floating-point operations are issued (IE⟦ie⟧ = 1).

3) FPST⟦z⟧ is the divide-by-zero signal.

4) FPST⟦i⟧ is the output floating-point-to-integer conversion invalid signal.

5) Divide and remainder do not check for divide-by-zero or output conversion errors.

6) If IE⟦fx⟧ = 1 and if the answer (quotient or remainder) of the floating-point-to-integer conversion is not an integral value, the INTPEN⟦fx⟧ inexact bit is set to 1. The FPST⟦x⟧ inexact bit is set to 1, independent of the IE⟦fx⟧ value for floating-point-to-integer conversion of nonintegral values.

Example 11–4. Integer Multiply, Divide, and Remainder (Continued)

(f) Unsigned Integer Remainder

```
frndz.ud  Num,Fnum          ; unsigned numerator to double-precision
                            ; floating-point
frndz.ud  Div,Fdiv          ; unsigned divisor to double-precision
                            ; floating-point
fdiv.ddd  Fnum,Fdiv,Fquot   ; quotient in double-precision floating-point
frndz.du  Fquot,Quot        ; double-precision floating-point to unsigned
                            ; quotient
fmpy.uuu  Div,Quot,UDxQ     ; product of unsigned integer divisor and
                            ; quotient
subu      Num,UDxQ,Rem      ; unsigned integer remainder = Num – Div*Quot
```

Notes: 1) Other rounding modes can be used (such as frndn for nearest), rather than the truncation in the frndz instructions.

2) Interrupts must be enabled before any floating-point operations are issued (IE⟦ie⟧ = 1).

3) FPST⟦z⟧ is the divide-by-zero signal.

4) FPST⟦i⟧ is the output floating-point-to-integer conversion invalid signal.

5) Divide and remainder do not check for divide-by-zero or output conversion errors.

6) If IE⟦fx⟧ = 1 and if the answer (quotient or remainder) of the floating-point-to-integer conversion is not an integral value, the INTPEN⟦fx⟧ inexact bit is set to 1. The FPST⟦x⟧ inexact bit is set to 1, independent of the IE⟦fx⟧ value for floating-point-to-integer conversion of nonintegral values.

11.4 ArcTAN Floating-Point Example

Consider the series expansion for ArcTAN(Z) as the following polynomial:

$$ArcTan(Z)=C_0+C_1*X+C_3*X^3+C_5*X^5+C_7*X^7+C_9*X^9+C_{11}*X^{11}+C_{13}*X^{13}$$

where X = (Z–1)/(Z+1) when Z is nonnegative.

This can be rewritten as:

$$ArcTan(Z)=C_0+(C_1+C_5*X^4+C_9*X^8+C_{13}*X^{12)}*X+(C_3+C_7*X^4+C_{11}*X^8)*X^3$$

or

$$ArcTan(Z)=C_0+Y_0*X+Y_1*X^3$$

where

$$Y_0=C_1+C_5*X^4+C_9*X^8+C_{13}*X^{12}$$

and

$$Y_1=C_3+C_7*X^4+C_{11}*X^8$$

Note that Y_0 and Y_1 do not depend on each other and can be calculated independently in parallel within the floating-point unit pipeline. This eliminates some of the holes in the floating-point unit pipeline. Example 11–5 illustrates nonvector, single-precision, floating-point code, and the pipeline is shown in Example 11–6.

This same program can be written to use the vector floating-point operations and improve efficiency, as illustrated in Example 11–7; the pipeline is shown in Example 11–8. In summary, the ArcTAN program requires 65, 47, or 41 cycles for the straightforward code (not shown), scalar single-precision floating point (Example 11–5), or vector single-precision floating point (Example 11–7), respectively.

Example 11–5. ArcTAN Single-Precision Scalar Floating-Point Sample

```
ArcTAN:                         ; Z is nonnegative

    fadd.sss  One,Z,Zp1         ; Zp1 = Z + 1
    fsub.sss  Z,One,Zm1         ; Zm1 = Z – 1
    fdiv.sss  Zm1,Zp1,X         ; X = (Z – 1) / (Z + 1)
    fmpy.sss  X,X,Xsq           ; Xsq = X * X
    fmpy.sss  Xsq,Xsq,X4th      ; X4th = Xsq * Xsq
    fmpy.sss  X,Xsq,X3rd        ; X3rd = X * Xsq
    fmpy.sss  C13,X4th,Y0       ; Y0 = C13 * X4th
    fmpy.sss  C11,X4th,Y1       ; Y1 = C11 * X4th
    fadd.sss  C9,Y0,Y0          ; Y0 = C9 + Y0
    fadd.sss  C7,Y1,Y1          ; Y1 = C7 + Y1
    fmpy.sss  Y0,X4th,Y0        ; Y0 = Y0 * X4th
    fmpy.sss  Y1,X4th,Y1        ; Y1 = Y1 * X4th
    fadd.sss  C5,Y0,Y0          ; Y0 = C5 + Y0
    fadd.sss  C3,Y1,Y1          ; Y1 = C3 + Y1
    fmpy.sss  Y0,X4th,Y0        ; Y0 = Y0 * X4th
    fmpy.sss  Y1,X3rd,Y1        ; Y1 = Y1 * X3rd
    fadd.sss  C1,Y0,Y0          ; Y0 = C1 + Y0
    fadd.sss  C0,Y1,Y1          ; Y1 = C0 + Y1
    fmpy.sss  Y0,X,Y0           ; Y0 = Y0 * X
    jsr       r31(r0),r0        ; subroutine return
    fadd.sss  Y0,Y1,Answer      ; Answer = Y0 + Y1
                                ; in branch delay slot
```

Notes: 1) X = (Z–1)/(Z+1) when Z is nonnegative.

2) Cn constants (including one) are preloaded into registers.

3) Interrupts must be enabled before any floating-point operations are issued (IE⟦ie⟧ = 1).

Example 11–6. ArcTAN Single-Precision Floating-Point Scalar Pipeline

	Unpack	Floating-Point Multiply Unit		Floating-Point Add Unit			
Cycle	Execute	Multiply	Normalize	Align	Add	Normalize	
1	fadd 1	— — —	— — —		— — —	— — —	
2	fsub Z	— — —	— — —	fadd 1	— — —	— — —	
3	— — —	— — —	— — —	fsub Z	fadd 1	— — —	
4	— — —	— — —	— — —	— — —	fsub Z	fadd 1	→ Zp1
5	— — —	— — —	— — —	— — —	— — —	fsub Z	→ Zm1
6	fdiv X†	— — —	— — —	— — —	— — —	— — —	
7	— — —	fdiv X	— — —	— — —	— — —	— — —	
8	— — —	fdiv X	— — —	— — —	— — —	— — —	
9	— — —	fdiv X	— — —	— — —	— — —	— — —	
10	— — —	fdiv X	— — —	— — —	— — —	— — —	
11	— — —	fdiv X	— — —	— — —	— — —	— — —	
12	— — —	fdiv X	— — —	— — —	— — —	— — —	
13	— — —	— — —	fdiv X	— — —	— — —	— — —	→ X
14	fmpy X	— — —	— — —	— — —	— — —	— — —	
15	— — —	fmpy X	— — —	— — —	— — —	— — —	
16	— — —	— — —	fmpy X	— — —	— — —	— — —	→ Xsq
17	fmpy X4	— — —	— — —	— — —	— — —	— — —	
18	fmpy X3	fmpy X4	— — —	— — —	— — —	— — —	
19	— — —	fmpy X3	fmpy X4	— — —	— — —	— — —	→ X4th
20	fmpy Y0	— — —	fmpy X3	— — —	— — —	— — —	→ X3rd
21	fmpy Y1	fmpy Y0	— — —	— — —	— — —	— — —	
22	— — —	fmpy Y1	fmpy Y0	— — —	— — —	— — —	→ Y0
23	fadd C9	— — —	fmpy Y1	— — —	— — —	— — —	→ Y1
24	fadd C7	— — —	— — —	fadd C9	— — —	— — —	
25	— — —	— — —	— — —	fadd C7	fadd C9	— — —	
26	— — —	— — —	— — —	— — —	fadd C7	fadd C9	→ Y0
27	fmpy Y0	— — —	— — —	— — —	— — —	fadd C7	→ Y1
28	fmpy Y1	fmpy Y0	— — —	— — —	— — —	— — —	
29	— — —	fmpy Y1	fmpy Y0	— — —	— — —	— — —	→ Y0
30	fadd C5	— — —	fmpy Y1	— — —	— — —	— — —	→ Y1
31	fadd C3	— — —	— — —	fadd C5	— — —	— — —	
32	— — —	— — —	— — —	fadd C3	fadd C5	— — —	
33	— — —	— — —	— — —	— — —	fadd C3	fadd C5	→ Y0
34	fmpy Y0	— — —	— — —	— — —	— — —	fadd C3	→ Y1

†fdiv waits for the scoreboard of the Zm1 register in the previous fsub instruction.

Example 11–6. ArcTAN Single-Precision Floating-Point Scalar Pipeline (Continued)

	Unpack	Floating-Point Multiply Unit		Floating-Point Add Unit			
Cycle	Execute	Multiply	Normalize	Align	Add	Normalize	
35	fmpy Y1	fmpy Y0	— — —	— — —	— — —	— — —	
36	— — —	fmpy Y1	fmpy Y0	— — —	— — —	— — —	→ Y0
37	fadd C1	— — —	fmpy Y1	— — —	— — —	— — —	→ Y1
38	fadd C0	— — —	— — —	fadd C1	— — —	— — —	
39	— — —	— — —	— — —	fadd C0	fadd C1	— — —	
40	— — —	— — —	— — —	— — —	fadd C0	fadd C1	→ Y0
41	fmpy Y0	— — —	— — —	— — —	— — —	fadd C0	→ Y1
42	(jsr)	fmpy Y0	— — —	— — —	— — —	— — —	
43	— — —	— — —	fmpy Y0	— — —	— — —	— — —	→ Y0
44	fadd Y0	— — —	— — —	— — —	— — —	— — —	
45	— — —	— — —	— — —	fadd Y0	— — —	— — —	
46	— — —	— — —	— — —	— — —	fadd Y0	— — —	
47	— — —	— — —	— — —	— — —	— — —	fadd Y0	→ Answer

Example 11–7. ArcTAN Single-Precision Vector Floating-Point Sample

```
ArcTAN:                          ; Z is nonnegative

    fadd.sss  One,Z,Zp1          ; Zp1 = Z + 1
    fsub.sss  Z,One,Zm1          ; Zm1 = Z – 1
    vmac.ssd  C1,One,0,a0        ; 0 + C1*1.0 = a0
    vmac.ssd  C3,One,0,a1        ; 0 + C3*1.0 = a1
    vmac.ssd  C0,One,0,a2        ; 0 + C0*1.0 = a2
    fdiv.sss  Zm1,Zp1,X          ; X = (Z – 1) / (Z + 1)
    fmpy.sss  X,X,Xsq            ; Xsq = X * X
    fmpy.sss  Xsq,Xsq,X4th       ; X4th = Xsq * Xsq
    fmpy.sss  X,Xsq,X3rd         ; X3rd = X * Xsq
    vmac.ssd  C5,X4th,a0,a0      ; a0 + C5*X4th = a0
    vmac.ssd  C7,X4th,a1,a1      ; a1 + C7*X4th = a1
    fmpy.sss  X4th,X4th,X8th     ; X8th = X4th * X4th
    vmac.ssd  C9,X8th,a0,a0      ; a0 + C9*X8th = a0
    fmpy.sss  X4th,X8th,X12th    ; X12th = X4th * X8th
    vmac.sss  C11,X8th,a1,Y1     ; a1 + C11*X8th = Y1
    vmac.sss  C13,X12th,a0,Y0    ; a0 + C13*X12th = Y0
    vmac.ssd  Y1,X3rd,a2,a2      ; a2 + Y1*X3rd = a2
    jsr       r31(r0),r0         ; subroutine return
    vmac.sss  Y0,X,a2,Answer     ; Answer = a2 + Y0*X
                                 ; in branch delay slot
```

Notes: 1) X = (Z–1)/(Z+1) when Z is nonnegative.

2) Cn constants (including one) are preloaded into registers.

3) Interrupts must be enabled before any floating-point operations are issued (IE⟦ie⟧ = 1).

Example 11–8. ArcTAN Single-Precision Floating-Point Vector Pipeline

	Unpack	Floating-Point Multiply Unit		Floating-Point Add Unit			
Cycle	Execute	Multiply	Normalize	Align	Add	Normalize	
1	fadd 1	— — —	— — —		— — —	— — —	
2	fsub Z	— — —	— — —	fadd 1	— — —	— — —	
3	vmac C1	— — —	— — —	fsub Z	fadd 1	— — —	
4	vmac C3	vmac C1	— — —	— — —	fsub Z	fadd 1	→ Zp1
5	vmac C0	vmac C3	vmac C1	— — —	— — —	fsub Z	→ Zm1
6	fdiv X	vmac C0	vmac C3	vmac C1	— — —	— — —	
7	— — —	fdiv X	vmac C0	vmac C3	vmac C1	— — —	
8	— — —	fdiv X	— — —	vmac C0	vmac C3	vmac C1	→ a0
9	— — —	fdiv X	— — —	— — —	vmac C0	vmac C3	→ a1
10	— — —	fdiv X	— — —	— — —	— — —	vmac C0	→ a2
11	— — —	fdiv X	— — —	— — —	— — —	— — —	
12	— — —	fdiv X	— — —	— — —	— — —	— — —	
13	— — —	— — —	fdiv X	— — —	— — —	— — —	→ X
14	fmpy X	— — —	— — —	— — —	— — —	— — —	
15	— — —	fmpy X	— — —	— — —	— — —	— — —	
16	— — —	— — —	fmpy X	— — —	— — —	— — —	→ Xsq
17	fmpy X4	— — —	— — —	— — —	— — —	— — —	
18	fmpy X3	fmpy X4	— — —	— — —	— — —	— — —	
19	— — —	fmpy X3	fmpy X4	— — —	— — —	— — —	→ X4th
20	vmac C5	— — —	fmpy X3	— — —	— — —	— — —	→ X3rd
21	vmac C7	vmac C5	— — —	— — —	— — —	— — —	
22	fmpy X4	vmac C7	vmac C5	— — —	— — —	— — —	
23	— — —	fmpy X4	vmac C7	vmac C5	— — —	— — —	
24	— — —	— — —	fmpy X4	vmac C7	vmac C5	— — —	→ X8th
25	vmac C9	— — —	— — —	— — —	vmac C7	vmac C5	→ a0
26	fmpy X4	vmac C9	— — —	— — —	— — —	vmac C7	→ a1
27	vmac C11	fmpy X4	vmac C9	— — —	— — —	— — —	
28	— — —	vmac C11	fmpy X4	vmac C9	— — —	— — —	→ X12th
29	vmac C13	— — —	vmac C11	— — —	vmac C9	— — —	
30	— — —	vmac C13	— — —	vmac C11	— — —	vmac C9	→ a0
31	— — —	— — —	vmac C13	— — —	vmac C11	— — —	
32	— — —	— — —	— — —	vmac C13	— — —	vmac C11	→ Y1
33	vmac Y1	— — —	— — —	— — —	vmac C13	— — —	
34	(jsr)	vmac Y1	— — —	— — —	— — —	vmac C13	→ Y0

Example 11–8. ArcTAN Single-Precision Floating-Point Vector Pipeline (Continued)

	Unpack	Floating-Point Multiply Unit		Floating-Point Add Unit			
Cycle	Execute	Multiply	Normalize	Align	Add	Normalize	
35	— — —	— — —	vmac Y1	— — —	— — —	— — —	
36	vmac Y0	— — —	— — —	vmac Y1	— — —	— — —	
37	— — —	vmac Y0	— — —	— — —	vmac Y1	— — —	
38	— — —	— — —	vmac Y0	— — —	— — —	vmac Y1	→ a2
39	— — —	— — —	— — —	vmac Y0	— — —		
40	— — —	— — —	— — —	— — —	vmac Y0	— — —	
41	— — —	— — —	— — —	— — —	— — —	vmac Y0	→ Answer

11.5 C Call Interface Example

Consider the following main C program:

Example 11–9. Vector-Matrix Multiply Main C Program

```
/* vector*matrix product */
void VecMatMpy4() ;
      float InVec[5][4] = {
            { 11, 12, 13, 14},
            { 21, 22, 23, 24},
            { 31, 32, 33, 34},
            { 41, 42, 43, 44},
            { 51, 52, 53, 54} };
      float    Matrix[4][4] = {
            {1.1,1.2,1.3,1.4},
            {2.1,2.2,2.3,2.4},
            {3.1,3.2,3.3,3.4},
            {4.1,4.2,4.3,4.4} };
      float OutVec[5][4];
      int      n=5;     /* number of vectors = 5 */
   main()
{
   VecMatMpy4(InVec, Matrix, OutVec, n);
                  /* calls InVec * Matrix = OutVec */
      return 0 ;
}
```

The standard C calling conventions used in Example 11–9 include the global declarations of functions in the calling program:

```
void VecMatMpy4() ;
```

The main program calls the subroutine VecMatMpy4 (integer *number of vectors* > 0) as:

```
VecMatMpy4(In_Vector, Matrix, Out_Vector, numvec) ;
```

where

```
float In_Vector[numvec][4]    ; Input vector as numvec × 4
float Matrix[4][4]            ; Input static matrix as 4× 4
float Out_Vector[numvec][4]   ; Output vector as numvec × 4
int numvec                    ; Number of vectors > 0
```

This becomes the standard MP assembly calling conventions as:

- Interrupt enable bit 0 must be set (IE⟦ie⟧ = 1) for floating-point operations
- R2 = In_Vector address (64-bit boundary)—argument 1
- R4 = Matrix input address (64-bit boundary)—argument 2
- R6 = Out_Vector address (64-bit boundary)—argument 3
- R8 = integer numvec > 0 as argument 4
- jsr.a _VecMatMpy4(R0),R31
- Any registers used in the range R20–R30 are saved on the stack and restored (four 64-bit stack doublewords are used in Example 11–10)

11.6 Matrix Multiply Using Vector Floating-Point Instructions

Using the C program in Example 11–9, the vector-matrix multiply (VecMatMpy4) is examined. The input arguments are passed in registers R2–R8 (label **Pointers** in Example 11–10). Label **Push** illustrates saving registers on the stack; label **Pop** shows the registers being restored.

The register allocation is shown in Example 11–11.

- The dynamic 1×4 input vector uses registers R4–R7.
- The dynamic 1×4 output vector uses registers R24–R27.
- The static 4×4 matrix uses registers R8–R23.

This assumes big-endian addressing. Notice the extensive use of load/store doubles in Example 11–10, which reduces the number of I/Os by a factor of 2. This requires all addresses to be on 64-bit boundaries.

The matrix load, vector load, and vector store are shown in Example 11–10 as labels **LoadMatrix**, **LoadInVec**, and **StoreOutVec**, respectively. The vector-matrix multiply code is not shown is this figure.

Example 11–10. Load/Store Doubles in [1x4]*[4x4] VecMatMpy in Big Endian

```
              .global  _VecMatMpy4     ; declare VecMatMpy4 to be a
                                       ;   global label

              .align   16*4            ; instruction cache boundary

_VecMatMpy4:                           ; Entry––uses 4 double precision
                                       ;   stack words

Push:         st.d     –4*8(r1:m),r20  ; push 64–bit r20 to (r1–32
                                       ;  bytes) then modify r1 = (r1 –
                                       ;  32 bytes)
              st.d     1*8(r1),r22     ; push 64–bit r22 to stack
              st.d     2*8(r1),r24     ; push 64–bit r24 to stack
              st.d     3*8(r1),r26     ; push 64–bit r26 to stack

Pointers:     wrcr     IN0P, r2        ; Input vector address (arg1)
              wrcr     IN1P, r4        ; Input matrix address (arg2)
              wrcr     OUTP, r6        ; Output vector address (arg3)
              add      8*4,r2,r3       ; r3 = 32 + r2 (r2 = input vector
                                       ;   address) load–look–ahead of
                                       ;   vectors
              add      –1,r8,r2        ; r2 = r8 – 1 [r8 = no. of
                                       ;   vectors] (arg4)
                                       ;                              Matrix
LoadMatrix:   vld1.d   r8              ; 64–bit r8  = memory(IN1P) 12, 11
              vld1.d   r10             ; 64–bit r10 = memory(IN1P) 14, 13
              vld1.d   r12             ; 64–bit r12 = memory(IN1P) 22, 21
              vld1.d   r14             ; 64–bit r14 = memory(IN1P) 24, 23
              vld1.d   r16             ; 64–bit r16 = memory(IN1P) 32, 31
              vld1.d   r18             ; 64–bit r18 = memory(IN1P) 34, 33
              vld1.d   r20             ; 64–bit r20 = memory(IN1P) 42, 41
              vld1.d   r22             ; 64–bit r22 = memory(IN1P) 44, 43
                                       ;                               InVec
LoadInVec:    vld0.d   r4              ; 64–bit r4  = memory(IN0P)  2,  1
              vld0.d   r6              ; 64–bit r6  = memory(IN0P)  4,  3

; Code for vector–matrix multiply goes here.
                                       ;                              OutVec
StoreOutVec:  vst.d    r24             ; memory(OUTP) = 64–bit r24  2,  1
              vst.d    r26             ; memory(OUTP) = 64–bit r26  4,  3

Pop:          ld.d     0*8(r1),r20     ; pop 64–bit r20 from stack
              ld.d     1*8(r1),r22     ; pop 64–bit r22 from stack
              ld.d     2*8(r1),r24     ; pop 64–bit r24 from stack
              ld.d     3*8(r1),r26     ; pop 64–bit r26 from stack

Exit:         jsr      r31(r0),r0      ; subroutine return
              addu     4*8,r1,r1       ; restore stack pointer in branch
                                       ;   delay slot
```

Note: Interrupts must be enabled (IE⟦ie⟧ = 1) before any floating-point or vector instructions are issued, including vld and vst.

Example 11–11. [1x4]*[4x4] = [1x4] Vector-Matrix Multiply Kernel in Big Endian

Single-Precision Floating-Point Registers

r5	r4	r7	r6

×

r9	r8	r11	r10
r13	r12	r15	r14
r17	r16	r19	r18
r21	r20	r23	r22

=

r25	r24	r27	r26

```
VM_mpy: vmac.ssd   r9,r5,0,a0          ;  0 +  r9 * r5 → a0
        vmac.ssd   r8,r5,0,a1          ;  0 +  r8 * r5 → a1
        vmac.ssd   r11,r5,0,a2         ;  0 + r11 * r5 → a2
        vmac.ssd   r10,r5,0,a3         ;  0 + r10 * r5 → a3

        vmac.ssd   r13,r4,a0,a0        ; a0 + r13 * r4 → a0
        vmac.ssd   r12,r4,a1,a1        ; a1 + r12 * r4 → a1
        vmac.ssd   r15,r4,a2,a2        ; a2 + r15 * r4 → a2
        vmac.ssd   r14,r4,a3,a3        ; a3 + r14 * r4 → a3

        vmac.ssd   r17,r7,a0,a0        ; a0 + r17 * r7 → a0
        vmac.ssd   r16,r7,a1,a1        ; a1 + r16 * r7 → a1
        vmac.ssd   r19,r7,a2,a2        ; a2 + r19 * r7 → a2
        vmac.ssd   r18,r7,a3,a3        ; a3 + r18 * r7 → a3

        vmac.sss   r21,r6,a0,r25       ; a0 + r21 * r6 → r25 in SP FP
        vmac.sss   r20,r6,a1,r24       ; a1 + r20 * r6 → r24 in SP FP
        vmac.sss   r23,r6,a2,r27       ; a2 + r23 * r6 → r27 in SP FP
        vmac.sss   r22,r6,a3,r26       ; a3 + r22 * r6 → r26 in SP FP
```

Notes: 1) Sample vector-matrix multiply with data resident in registers.

2) Interrupts must be enabled (IE⟦ie⟧ = 1) before any floating-point or vector instructions are issued.

3) **SP** = single precision
FP = floating point

Once the data has been loaded into the registers, the vector-matrix multiply code is shown in Example 11–11. However, you can overlap some of the computations with the data loads, as illustrated in Example 11–12. This saves eight individual vld instructions.

Notice that the load of register R6 is between the loads of registers R20 and R22 in Example 11–12. This load of register R6 assures that the data in register R7 is valid for the first vmac that uses register R7. The last vld of register R4 is the anticipatory load of the first two elements of the next vector (if needed). The vst instructions for the output vector are not shown in Example 11–12 but are deferred to Example 11–13.

Note:

Care should be taken when intermixing load/store double with load/store single instructions. While ld.d, vld.d, st.d, and vst.d instructions are endian-independent, endian order dependency still applies to ld, vld.s, st, and vst.s operations.

Example 11–12. First Vector-Matrix Multiply, Register Load in Big Endian

```
              .global  _VecMatMpy4      ; declare VecMatMpy4 to be a
                                        ; global label

              .align   16*4             ; instruction cache boundary

_VecMatMpy4:                            ; Entry uses four double-precision
                                        ; stack words

Push:         st.d     –4*8(r1:m),r20   ; push 64–bit r20 to (r1–32
                                        ; bytes), then modify r1 =
                                        ; (r1 – 32 bytes)
              st.d     1*8(r1),r22      ; push 64–bit r22 to stack
              st.d     2*8(r1),r24      ; push 64–bit r24 to stack
              st.d     3*8(r1),r26      ; push 64–bit r26 to stack
```

(continued on next page)

Example 11–12. First Vector-Matrix Multiply, Register Load in Big Endian (Continued)

```
Pointers:     wrcr     IN0P, r2        ; Input vector address (arg1)
              wrcr     IN1P, r4        ; Input matrix address (arg2)

              add      4*4,r2,r3       ; r3 = 16 + r2 (r2 = input vector
                                       ; address)
                                       ;   load–look–ahead of vectors
              add      –1,r8,r2        ; r2 = r8 – 1 [r8 = no of vectors]
                                       ;  (arg4)
              v1d1.d   r8              ; 64–bit r8 = Memory(IN1P) 12, 11
              v1d0.d   r4              ; 64–bit r4 = Memory(IN0P)  2,  1
              wrcr     OUTP, r6        ; Output Vector Address (arg3)

VM_mpy1:                               ; 1st vector-matrix multiply code
                                       ;   input matrix
     vmac.ssd r9,r5,0,a0     || vld1.d r10  ;  0 +  r9 * r5 →a0; 14, 13
     vmac.ssd r8,r5,0,a1     || vld1.d r12  ;  0 +  r8 * r5 →a1; 22, 21
     vmac.ssd r11,r5,0,a2    || vld1.d r14  ;  0 + r11 * r5 →a2; 24, 23
     vmac.ssd r10,r5,0,a3    || vld1.d r16  ;  0 + r10 * r5 →a3; 32, 31

     vmac.ssd r13,r4,a0,a0   || vld1.d r18  ; a0 + r13 * r4 →a0; 34, 33
     vmac.ssd r12,r4,a1,a1   || vld1.d r20  ; a1 + r12 * r4 →a1; 42, 41
     vmac.ssd r15,r4,a2,a2   || vld0.d r6   ; a2 + r15 * r4 →a2
                                            ; in input vector 4,3
     vmac.ssd r14,r4,a3,a3   || vld1.d r22  ; a3 + r14 * r4 →a3; 44, 43

     vmac.ssd r17,r7,a0,a0   || vld0.d r4   ; a0 + r17 * r7 →a0
                                            ; load input vector 2,1

; Optional Load-Look-Ahead instruction goes here as: ld 4*4(r3:m),r0.

     vmac.ssd r16,r7,a1,a1                  ; a1 + r16 * r7 →a1
     vmac.ssd r19,r7,a2,a2                  ; a2 + r19 * r7 →a2
     vmac.ssd r18,r7,a3,a3                  ; a3 + r18 * r7 →a3

     vmac.sss r21,r6,a0,a25               ; a0 + r21 * r6 →r25 in SP FP
     vmac.sss r20,r6,a1,r24               ; a1 + r20 * r6 →r24 in SP FP
     vmac.sss r23,r6,a2,r27               ; a2 + r23 * r6 →r27 in SP FP
     vmac.sss r22,r6,a3,r26               ; a3 + r22 * r6 →r26 in SP FP

     bcnd     Pop,r2,le0.w                ; exit if no. vectors is – or 0
     add         –1,r2,r2                 ; decrement no. vector count in
                                          ; r2
```

Note: Interrupts must be enabled (IE⟦ie⟧ = 1) before any floating-point or vector instructions are issued, including vld and vst.

The output vectors are stored, as shown in Example 11–13, as vst.d registers R24 and R26. The optional ld 4*4(r3),r0 reads one vector ahead of the vld0.d R4 instruction in an effort to hide most of a potential data cache miss (if needed).

Example 11–13. Nth Vector-Matrix Multiply, Register Load/Store in Big Endian

```
VM_mpyN:          ; N-th vector-matrix multiply
          vmac.ssd  r9,r5,0,a0              ;  0 +  r9 * r5 →a0
          vmac.ssd  r8,r5,0,a1              ;  0 +  r8 * r5 →a1
          vmac.ssd  r11,r5,0,a2             ;  0 + r11 * r5 →a2
          vmac.ssd  r10,r5,0,a3             ;  0 + r10 * r5 →a3

          vmac.ssd  r13,r4,a0,a0            ; a0 + r13 * r4 →a0
          vmac.ssd  r12,r4,a1,a1||vld0.d r6 ; a1 + r12 * r4 →a1
                                            ; load input vector 4,3
          vmac.ssd  r15,r4,a2,a2||vst.d r24 ; a2 + r15 * r4 →a2
                                            ; write 2, 1
          vmac.ssd  r14,r4,a3,a3||vst.d r26 ; a3 + r14 * r4 →a3
                                            ; write 4, 3
          vmac.ssd  r17,r7,a0,a0||vld0.d r4 ; a0 + r17 * r7 →a0
                                            ; load new input vector 2,1

; Optional Load-Look-Ahead instruction goes here as: ld 4*4(r3:m),r0.

          vmac.ssd  r16,r7,a1,a1            ; a1 + r16 * r7 →a1
          vmac.ssd  r19,r7,a2,a2            ; a2 + r19 * r7 →a2
          vmac.ssd  r18,r7,a3,a3            ; a3 + r18 * r7 →a3

          vmac.sss  r21,r6,a0,r25           ; a0 + r21 * r6 →r25 in SP
                                            ; FP
          vmac.sss  r20,r6,a1,r24           ; a1 + r20 * r6 →r24 in SP
                                            ; FP
          vmac.sss  r23,r6,a2,r27           ; a2 + r23 * r6 →r27 in SP
                                            ; FP
          vmac.sss  r22,r6,a3,r26           ; a3 + r22 * r6 →r26 in SP
                                            ; FP

          bcnd      VM_mpyN,r7,gt0.w        ; loop if no. vectors is > 0
          add       –1,r2,r2                ; decrement no. vector count
                                            ; in r2

Pop:      ld.d      0*8(r1),r20             ; pop 64-bit r20 from stack
          ld.d      1*8(r1),r22             ; pop 64-bit r22 from stack
          vst.d     r24                     ; write OutVector 2,1
          ld.d      2*8(r1),r24             ; pop 64-bit r24 from stack
          vst.d     r26                     ; write OutVector 4,3
          ld.d      3*8(r1),r26             ; pop 64-bit r26 from stack

Exit:     jsr       r31(r0),r0              ; subroutine return
          addu      4*8,r1,r1               ; restore stack pointer
                                            ;       in branch delay slot
```

Note: Interrupts must be enabled (IE⟦ie⟧ = 1) before any floating-point or vector instructions are issued, including vld and vst.

In summary, the full VecMatMpy4 subroutine is the code shown in Example 11–12, followed by the code shown in Example 11–13.

- The first section of VecMatMpy4 shown in Example 11–12 pushes the registers R20–R27 onto the stack, making room for the input data.
- Next, the input and output addresses are loaded into control registers.
- Then the first values of input vector and matrix are loaded so that the vector operations can begin (there is one-cycle latency between vld instruction and data being available).
- Assuming that numvec is greater than 1, an anticipatory load from the next data region is calculated in register R3 for future use in an effort to minimize cache miss times.

In Example 11–12, the first output 1×4 vector is calculated using all four accumulators. Also, the remaining elements of the matrix and the first input vector are loaded. The optional ld instruction is reading an element that is two vectors ahead of the one being used in the vector-matrix-multiply, so that if a cache miss is taken, part of it is hidden by the computations. The final values of the output vector are indicated by the dest register in the last four vmac.sss instructions. The bcnd and add instructions are the loop controls to determine when the last vector has been computed. Notice that there are no stores in this example; the answers normally take six cycles from their respective vmac instructions.

In Example 11–13, the remaining output 1×4 vectors are calculated using all four accumulators. Also, the next elements of the input vector are loaded. The previous output vector is stored using the two vst.d instructions. The optional ld instruction is reading an element two vectors ahead of the one being used in the vector-matrix-multiply, so that if a cache miss is taken, part of it is hidden by the computations. The final values of the output vector are indicated by the dest register in the last four vmac.sss instructions. The bcnd and add instructions are the loop controls to determine when the last vector has been computed. The answers normally take six cycles from their respective vmac instructions.

When the loop is completed, the stack information is restored to the original registers and the stack pointer reset. The final 1×4 output vector is stored (two vst.d instructions), and the subroutine exits.

11.6.1 Notes on VecMatMpy Data

Since the MP has a data cache size of 4096 bytes or 1024 32-bit words (256 1×4 vectors), you can improve the data-cache hit rate by setting the In_Vector address to the Out_Vector address (250 1×4 vectors). Otherwise, 125 1×4 input and output vectors will fill the data cache. Matrix requires 16 32-bit words.

If there are a large number of vectors to be processed, most of the data cache times can be hidden by executing a look-ahead-read (one subblock is 16 32-bit words = 4 1×4 vectors). This means every four vectors can cause a read data cache miss (or 0.25 cache misses per 1×4 vector).

If the data cache is full, then writeback of data cache is required if the least recently used (LRU) block has been modified. Only after the data-cache block (1 block = 4 subblocks) has been written can a new subblock be read from external memory.

If the program is not resident in instruction cache, then four instruction-cache reads are required to load this program.

11.7 Performance of the Matrix Multiply

The following assumptions are used in this section:

- 50 MHz is the MVP internal clock rate.
- Cache subblock is 16 32-bit words (data or instructions). External DRAM is 2 cycles—read/write cache subblock is 22 cycles minimum.
- The transfer controller is available immediately for all cache services.
- When you are performing the look-ahead-read, the wait time for data load is ≈ 8 cycles net per cache miss; the remainder of the 22 cycles is hidden by computations.
- All data and programs begin on a cache subblock boundary (multiple of 16 32-bit words).
- numvec values shown provide the best performance; however, available memory determines the practical numvec limit.
- Performance assumes no floating-point exceptions, denormals, NaNs, infinities, or contention.

11.7.1 Absolute Peak

The absolute peak performance is 16 cycles/vector, based on no cache misses, no look-ahead-read waits, and no loop logic with program unrolled to hide pipeline latencies. This maps to 100 MFLOPS and 3.1 million 1×4 vectors/second. This assumes that the program and the entire input/output data are resident in caches. If input is equal to the output address, numvec is less than 251. If input is not equal to the output address, numvec is less than 126.

11.7.2 Looping With No Cache Misses

The absolute peak performance is 19 cycles/vector, based on no cache misses, look-ahead-read hidden, and the addition of the overhead of looping. This is 84.2 MFLOPS and 2.6 million 1×4 vectors/second. This assumes that the program and the entire input/output data are resident in caches. If input equals the output address, numvec is less than 251. If input is not equal to the output address, numvec is less than 126.

Note:

This is the performance when the MP jointly shares PP data RAM banks (provided the PP does not overwrite the MP's data).

11.7.3 Looping With One Cache Miss for Four Vectors

If there is one data-cache miss for every four vectors, then the absolute peak performance is:

```
19 cycles * 4 vectors + 1 miss * 8 net cycles = 84 cycles
```

or

```
84 cycles / 4 vectors = 21 cycles/vector average
```

This provides 76.2 MFLOPS and 2.4 million 1×4 vectors/second. This assumes that the program is resident and either the input data or the output data caused the cache miss. If input is equal to the output address, numvec is less than 251. If input is not equal to the output address, numvec is less than 126.

11.7.4 Looping With Two Cache Misses for Four Vectors

If the number of data-cache misses doubles to two for every four vectors (write followed by read = 22 + 8 = 30 cycles), then the absolute peak performance is:

```
19 cycles * 4 vectors + 1 write/read * 30 cycles = 106
cycles
```

or

```
106 cycles / 4 vectors = 26.5 cycles/vector average
```

This provides 60.4 MFLOPS and 1.9 million 1×4 vectors/second. This assumes that the number of vectors exceeds the number that can be held in data cache. If input is equal to the output address, numvec is greater than 250. If input is not equal to the output address, numvec is greater than 125.

11.7.5 Looping With Three Cache Misses for Four Vectors

If three data-cache miss accesses are required for four vectors (read input, write output, read output = 8 + 22 + 22 = 52 cycles), then the absolute peak performance is:

```
19 cycles * 4 vectors + 1 read/write/read * 52 cycles =
128 cycles
```

or

```
128 cycles / 4 vectors = 32 cycles/vector average
```

This produces 50 MFLOPS and 1.6 million 1×4 vectors/second. If input is equal to the output address, numvec is greater than 250. If input is not equal to the output address, numvec is greater than 125.

11.7.6 Instructions Not Resident

If four instruction-cache reads are required, the numbers above must also include the 4 * 22 = 88 cycles in the totals at program start-up. This is amortized by the total number of output vectors generated.

11.7.7 TC Round Robin

If other processors require the transfer controller at the same time, the absolute peak performance will decrease because of the round-robin memory request services.

11.7.8 Summary

Table 11–1 summarizes the preliminary performance estimates for [1 × 4]*[4 × 4] vector-matrix multiply. Should the data size exceed available data cache, then the performance will approach the three cache-miss numbers.

Table 11–1. VecMatMpy4 Performance Estimates

	MVP = 50 MHz, 2-Cycle DRAM, No Contention or Exceptions				
				Number of Vectors	
Cycles per Vector	MFLOPS Peak	Million 1 × 4 Vectors per Second	Number of Cache Misses per Subblock	Input = Output Address	Input ≠ Output Address
16	100.0	3.1	0	< 251	< 126
19	84.2	2.6	0	< 251	< 126
21	76.2	2.1	1	< 251	< 126
26.5	60.4	1.9	2	> 250	> 125
32	50.0	1.6	3	> 250	> 125

11.8 Combined Sine and Cosine Using Vector Floating Point

Using the vector floating-point unit pipelines, the expansion for sine and cosine (in radians) is formed simultaneously. Once the two values are returned, then all six of the trigonometry values are optionally formed. These include (where $-1 \leq X \leq +1$):

- $\text{Sine}(X*\pi/2) = C_1*X + C_3*X^3 + C_5*X^5 + C_7*X^7 + C_9*X^9$
- $\text{Cosine}(X*\pi/2) = \text{Sine}(\pi/2 - |X|*\pi/2)$
- $\text{Tangent}(X*\pi/2) = \text{Sine}(X*\pi/2) / \text{Cosine}(X*\pi/2)$
- $\text{Cotangent}(X*\pi/2) = \text{Cosine}(X*\pi/2) / \text{Sine}(X*\pi/2)$
- $\text{Secant}(X*\pi/2) = 1.0 / \text{Cosine}(X*\pi/2)$
- $\text{Cosecant}(X*\pi/2) = 1.0 / \text{Sine}(X*\pi/2)$

The assumption here is that single-precision floating-point argument X has been scaled to have magnitude ≤ 1, which includes the removal of the $\pi/2$ factor as well. The sine equation is rewritten as:

$$\text{Sine}(X*\pi/2) = (C_1 + C_5*X^4 + C_9*X^8)*X^1 + (C_3 + C_7*X^4)*X^3$$

or

$$\text{Sine}(X*\pi/2) = [(C_1 + C_5*X^4 + C_9*X^8) + (C_3 + C_7*X^4)*X^2]*X^1$$

A sample vector floating-point program is shown in Example 11–14. The floating-point pipeline is shown in Example 11–15. This demonstrates that both sine and cosine can be formed in approximately the time required for sine only or cosine only.

Example 11–14. Vector Single-Precision Floating Point of Six Forward Trigonometry Values

```
VecTrig:                        ; Sin, Cos, Tan, Cot, Sec, and Csc

   and         AbsMsk,X,Y       ; Y = 0x7fffffff & X for cosine
   fsub.sss    One,Y,Y          ; Y = 1 – |X| = cosine argument

   fmpy.sss    X,X,Xsq          ; Xsq = X*X for sine
   fmpy.sss    Y,Y,Ysq          ; Ysq = Y*Y for cosine
   fmpy.sss    Xsq,Xsq,X4th     ; X4th = Xsq*Xsq for sine
   fmpy.sss    Ysq,Ysq,Y4th     ; Y4th = Ysq*Ysq for cosine
   fmpy.sss    X4th,X4th,X8th   ; X8th = X4th*X4th for sine
   fmpy.sss    Y4th,Y4th,Y8th   ; Y8th = Y4th*Y4th for cosine

   vmac.ssd    C9,X8th,0,a0     ; 0 + C9*X8th = a0 for sine
   vmac.ssd    C9,Y8th,0,a2     ; 0 + C9*Y8th = a2 for cosine
   vmac.ssd    C7,X4th,0,a1     ; 0 + C7*X4th = a1 for sine
   vmac.ssd    C7,Y4th,0,a3     ; 0 + C7*Y4th = a3 for cosine

   vmac.ssd    C5,X4th,a0,a0    ; a0 + C5*X4th = a0 for sine
   vmac.ssd    C5,Y4th,a2,a2    ; a2 + C5*Y4th = a2 for cosine
   vmac.sss    One,C3,a1,SIN    ; a1 + 1*C3 = SIN for sine
   vmac.sss    One,C3,a3,COS    ; a3 + 1*C3 = COS for cosine

   vmac.ssd    One,C1,a0,a0     ; a0 + 1*C1 = a0 for sine
   vmac.ssd    One,C1,a2,a2     ; a2 + 1*C1 = a2 for cosine
   vmac.sss    SIN,Xsq,a0,SIN   ; a0 + SIN*Xsq = SIN for sine
   vmac.sss    COS,Ysq,a2,COS   ; a2 + COS*Xsq = COS for cosine

   fmpy.sss    SIN,X,SIN        ; SIN*X = SIN (answer for sine)
   fmpy.sss    COS,Y,COS        ; COS*Y = COS (answer for cosine)

   fdiv.sss    One,SIN,CSC      ; 1/SIN = CSC (answer for cosecant)
   fdiv.sss    One,COS,SEC      ; 1/COS = SEC (answer for secant)

   fmpy.sss    COS,CSC,COT      ; COS*CSC = COT (answer for cotangent)
   fmpy.sss    SIN,SEC,TAN      ; SIN*SEC = TAN (answer for tangent)
```

Notes: 1) The factor of $\pi/2$ has been removed upon entry.

2) The divide-by-zero check is not shown in the code above.

Example 11–15. Vector Single-Precision Floating Point of Six Forward Trigonometry Values Pipeline

	Unpack	Floating-Point Multiply Unit		Floating-Point Add Unit			
Cycle	Execute	Multiply	Normalize	Align	Add	Normalize	
1	(and)	— — —	— — —	— — —	— — —	— — —	→ Y
2	fsub 1	— — —	— — —	— — —	— — —	— — —	
3	fmpy X	— — —	— — —	fsub 1	— — —	— — —	
4	— — —	fmpy X	— — —	— — —	fsub 1	— — —	
5	— — —	— — —	fmpy X	— — —	— — —	fsub 1	→ Y
6	fmpy Y	— — —	fmpy X	— — —	— — —	— — —	→ Xsq
7	fmpy Xs	fmpy Y	— — —	— — —	— — —	— — —	
8	— — —	fmpy Xs	fmpy Y	— — —	— — —	— — —	→ Ysq
9	fmpy Ys	— — —	fmpy Xs	— — —	— — —	— — —	→ X4th
10	fmpy X4	fmpy Ys	— — —	— — —	— — —	— — —	
11	— — —	fmpy X4	fmpy Ys	— — —	— — —	— — —	→ Y4th
12	fmpy Y4	— — —	fmpy X4	— — —	— — —	— — —	→ X8th
13	vmac X8	fmpy Y4	— — —	— — —	— — —	— — —	
14	— — —	vmac X8	fmpy Y4	— — —	— — —	— — —	→ Y8th
15	vmac Y8	— — —	vmac X8	— — —	— — —	— — —	
16	vmac X4	vmac Y8	— — —	vmac X8	— — —	— — —	
17	vmac Y4	vmac X4	vmac Y8	— — —	vmac X8	— — —	
18	vmac X4	vmac Y4	vmac X4	vmac Y8	— — —	vmac X8	→ a0
19	vmac Y4	vmac X4	vmac Y4	vmac X4	vmac Y8	— — —	
20	vmac C3	vmac Y4	vmac X4	vmac Y4	vmac X4	vmac Y8	→ a2
21	vmac 1	vmac C3	vmac Y4	vmac X4	vmac Y4	vmac X4	→ a1
22	vmac C1	vmac 1	vmac C3	vmac Y4	vmac X4	vmac Y4	→ a3
23	vmac 1	vmac C1	vmac 1	vmac C3	vmac Y4	vmac X4	→ a0
24	— — —	vmac 1	vmac C1	vmac 1	vmac C3	vmac Y4	→ a2
25	— — —	— — —	vmac 1	vmac C1	vmac 1	vmac C3	→ Si(ne)
26	vmac Si	— — —	— — —	vmac 1	vmac C1	vmac 1	→ Co(sine)
27	vmac Co	vmac Si	— — —	— — —	vmac 1	vmac C1	→ a0
28	— — —	vmac Co	vmac Si	— — —	— — —	vmac 1	→ a2
29	— — —	— — —	vmac Co	vmac Si	— — —	— — —	
30	— — —	— — —	— — —	vmac Co	vmac Si	— — —	
31	— — —	— — —	— — —	— — —	vmac Co	vmac Si	→ Si(ne)
32	fmpy Si	— — —	— — —	— — —	— — —	vmac Co	→ Co(sine)

Example 11–15. Vector Single-Precision Floating Point of Six Forward Trigonometry Values Pipeline (Continued)

	Unpack	Floating-Point Multiply Unit		Floating-Point Add Unit			
Cycle	Execute	Multiply	Normalize	Align	Add	Normalize	
33	fmpy Co	fmpy Si	— — —	— — —	— — —	— — —	
34	— — —	fmpy Co	fmpy Si	— — —	— — —	— — —	→ Si(ne)
35	fdiv Si	— — —	fmpy Co	— — —	— — —	— — —	→ Co(sine)
36	— — —	fdiv Si	— — —	— — —	— — —		
37	— — —	fdiv Si	— — —	— — —	— — —	— — —	
38	— — —	fdiv Si	— — —	— — —	— — —	— — —	
39	— — —	fdiv Si	— — —	— — —	— — —	— — —	
40	— — —	fdiv Si	— — —	— — —	— — —	— — —	
41	fdiv Co	fdiv Si	— — —	— — —	— — —	— — —	
42	— — —	fdiv Co	fdiv Si	— — —	— — —	— — —	→ Cosecant
43	— — —	fdiv Co	— — —	— — —	— — —	— — —	
44	— — —	fdiv Co	— — —	— — —	— — —	— — —	
45	— — —	fdiv Co	— — —	— — —	— — —	— — —	
46	— — —	fdiv Co	— — —	— — —	— — —	— — —	
47	fmpy Co	fdiv Co	— — —	— — —	— — —	— — —	
48	— — —	fmpy Co	fdiv Co	— — —	— — —	— — —	→ Secant
49	fmpy Si	— — —	fmpy Co	— — —	— — —	— — —	→ Cotangent
50	— — —	fmpy Si	— — —	— — —	— — —	— — —	
51	— — —	— — —	fmpy Si	— — —	— — —	— — —	→ Tangent

11.9 Pack/Unpack for Load/Store Double

In order to be endian-independent, 32-bit integer or single-precision floating-point numbers can be packed into a 64-bit doubleword so that load-double and store-double can reduce the number of memory references. Example 11–16 illustrates sample MP assembler code. This same technique can be applied to pack four halfwords or eight bytes into a 64-bit doubleword.

Example 11–16. Pack/Unpack for Load/Store Double

(a) MP code to pack two 32-bit registers into one 64-bit double register (for example, (Reg1, Reg2) → Reg3).

```
addu  0,r2,MSwd      ; Reg1 → MSword for Reg3
jsr   r31(r0),r0     ; optional return
addu  0,r4,LSwd      ; Reg2 → LSword for Reg3
```

(b) MP code to unpack two 32-bit registers from one 64-bit double register, (for example, Reg3 → (Reg1,Reg2)).

```
addu  0,MSwd,r2      ; MSword in Reg3 → Reg1
jsr   r31(r0),r0     ; optional return
addu  0,LSwd,r4      ; LSword in Reg3 → Reg2
```

(c) MP code for packing two 32-bit words into one 64-bit doubleword using addresses (for example, (Arg1,Arg2) → Arg3).

```
ld    Arg1(r0),MSwd  ; Memory(Arg1) → MSword for arg3
ld    Arg2(r0),LSwd  ; Memory(Arg2) → LSword for arg3
jsr   r31(r0),r0     ; optional return
st.d  Arg3(r0),LSwd  ; 64-bit r6 → Memory(Arg3)
```

(d) MP code for unpacking two 32-bit words from one 64-bit doubleword using addresses (for example, Arg3 → (Arg1,Arg2)).

```
ld.d  Arg3(r0),LSwd  ; Memory(Arg3) → 64-bit r6
st    Arg1(r0),MSwd  ; MSword in Arg1 → Memory(Arg1)
jsr   r31(r0),r0     ; optional return
st    Arg2(r0),LSwd  ; LSword in Arg1 → Memory(Arg2)
```

Notes: 1) The jsr instruction is shown for subroutine return. It is not required if used in macro code.

11.10 Clean Data Cache Using the dcachec Instruction

Example 11–17 illustrates a program that writes back modified data cache subblocks to external memory to preserve the data cache coherency. Note that the cmnd instruction's DCR bit resets flags but does **not** execute a writeback of data.

Refer to Figure 3–5, *MP Cache LRU Register*, and Figure 3–6, *MP Cache Tag Registers*, for additional details.

Example 11–17. Data-Cache Clean Sample Code

```
        .global _CleanDCache

crNum   .set    r2                      ; control register number for
                                        ;  DTAGn, n=0,1, ... ,15
setNum  .set    r3                      ; set number 0, 1, 2, or 3
blkNum  .set    r4                      ; block number 0, 1, 2, or 3
tag     .set    r5                      ; tag register value for current
                                        ;  block
dirty   .set    r6                      ; dirty bits for current block

_CleanDCache:
        or      0x400,r0,crNum          ; CR number for DTAG0
        or      r0,r0,setNum            ; initial set number = 0
SetLoop:
        or      r0,r0,blkNum            ; initial block number = 0
BlockLoop:
        rdcr    crNum,tag               ; read next DTAG register
        and     0x154,tag,dirty         ; isolate dirty bits
        bcnd    BlockClean,dirty,eq0.w  ; jump if block is entirely
                                        ;  clean
        addu    1,blkNum,blkNum         ; increment block number
CleanSub0:
        bbz     CleanSub1,dirty,2       ; jump if subblock 0 is clean
        sl.im   8,10,setNum,tag         ; add set to tag address
        dcachec 0x00(tag)               ; clean dirty subblock 0
CleanSub1:
        bbz.a   CleanSub2,dirty,4       ; jump if subblock 1 is clean
        dcachec 0x40(tag)               ; clean dirty subblock 1
CleanSub2:
        bbz.a   CleanSub3,dirty,6       ; jump if subblock 2 is clean
        dcachec 0x80(tag)               ; clean dirty subblock 2
CleanSub3:
        bbz.a   BlockClean,dirty,8      ; jump if subblock 3 is clean
        dcachec 0xc0(tag)               ; clean dirty subblock 3
BlockClean:
        bbz     BlockLoop,blkNum,2      ; loop again if block < 4
        addu    1,crNum,crNum           ; point to next DTAG register

        bbz     SetLoop,setNum,2        ; loop again if set < 4
        addu    1,setNum,setNum         ; increment set number

        jsr.a   r31(r0),r0              ; return from subroutine
```

11.11 Floating-Point Exception Interrupt Handler Routines

Subsection 11.11.1 describes the recommended floating-point exception interrupt procedure. The following must be true to execute floating-point exception interrupt handler routines:

- The floating-point unit must have interrupts enabled (IE⟦ie⟧ = 1) before any floating-point instruction can be submitted.
- A floating-point instruction is considered complete when that instruction writes status information into the FPST register. Every floating-point instruction writes to FPST once and only once.
- Floating-point exception interrupt handler routines are not called unless a floating-point instruction produces an exception and its corresponding bit is 1 in register IE.
- Floating-point exceptions do not set bits in the INTPEN register unless that same bit is 1 (enables that interrupt) in register IE.
- Floating-point operations are executed in one of these modes:
  - Pipelined (parallel) mode uses FPST⟦sm⟧ = 0 for overlapped parallel floating-point operations.
    - When floating-point interrupts are disabled, floating-point instructions that generate exceptions do not take an interrupt.
    - When floating-point interrupts are enabled, a floating-point instruction that produces an exception takes a floating-point interrupt if that corresponding bit is 1 in register IE. However, if that bit is 0 in register IE, then no interrupt is taken for that floating-point exception.

- Sequential (serial) mode uses FPST⟦sm⟧ = 1 for debugging code that produces exceptions.
  - When floating-point interrupts are disabled, floating-point instructions that generate exceptions do not take an interrupt.
  - When floating-point interrupts are enabled, a floating-point instruction that produces an exception takes a floating-point interrupt if that corresponding bit is 1 in register IE. However, if that bit is 0 in register IE, then no interrupt is taken for that floating-point exception.

    When an input exception occurs (for example, divide by zero, signaling NaNs) and that bit is 1 in register IE, then the write to register destination is inhibited so that floating-point exception interrupt handler routines can examine input operands to determined the reason for the exception. Results are written to register destination in all other cases, including output exceptions (for example, **frndn.di r8,r6** when the output integer overflows the 32-bit destination register).

11.11.1 Recommended Floating-Point Exception Interrupt Procedure

When an interrupt service routine is entered, the routine should save the context of the interrupted program. The context of the interrupted program should be restored before unstalling the floating-point unit. The reason is that the floating-point pipeline, once unstalled, writes the results to the destination registers, overwriting their previous contents. If this is done in a context other than the one in which the floating-point operations were initiated, the loss of the contents of these registers could cause the context to be irretrievably corrupted.

The recommended procedure for floating-point exception interrupt handling routines is as follows:

- ❑ Clear the current interrupt bit in the INTPEN register (set this interrupt bit to 0 by using the wrcr instruction to write a 1 to that bit.)
- ❑ Save the context of the interrupted program. This includes the register file and the control registers (especially the EPC, EIP, IE, and FPST registers.)
- ❑ Execute the main body of the floating-point exception interrupt handling routine.

At the end of the floating-point exception interrupt handling routine:

- ❑ Clear other pending floating-point exceptions. This is optional according to the needs of your application.
- ❑ IF other exceptions are cleared or there are no other exceptions:
  - ■ Restore the context of the interrupted program.
  - ■ Use the swcr or wrcr instruction to reset the FPST⟦fs⟧ bit to 0.
  - ■ Determine if sequential mode is enabled:
    - ■ Sequential mode is enabled (FPST⟦sm⟧ = 1): return branch EPC, followed by a nop instruction in the branch delay slot (use the trap return sequence).
    - ■ Sequential mode is not enabled (FPST⟦sm⟧ = 0): use the interrupt return sequence—return branch EIP, followed by return branch EPC.

- ELSE other exceptions are not cleared or there are no other exceptions:
  - Do not change FPST[[fs]].
  - Restore the context of the interrupted program.
  - Return branch EIP.
  - Return branch EPC.

Note:

The MP must be in the supervisor mode to swap (swcr) or write (wrcr) control register numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

11.11.2 Normal Pipelined Mode—Not in Floating-Point Sequential Mode

Pipelined (parallel) mode uses FPST[[sm]] = 0 for overlapped floating-point unit pipelined operations.

- When floating-point interrupts are disabled, floating-point instructions that generate exceptions do not take an interrupt.
- When floating-point interrupts are enabled, a floating-point instruction that produces an exception takes a floating-point interrupt if that corresponding bit is 1 in register IE. However, if that bit is 0 in register IE, then no interrupt is taken for that floating-point exception.
- Standard floating-point exception interrupt handling routine pipelined mode exit includes:

```
brcr  EIP   ; branch to EIP
brcr  EPC   ; restores PC
```

- The addresses in EIP, EPC can be multiple cycles away from the instruction that caused the exception (for example, fdiv.ddd r4,r6,r8 output exception such as inexact). If an intervening branch or jump instruction has been taken, then EIP, EPC can be quite different than the location of the floating-point instruction that produced the exception.

Part of a sample floating-point exception interrupt handling routine code is shown in subsection 11.11.1.

11.11.3 Sequential Mode—Serial Floating-Point Operation

Sequential (serial) mode uses FPST⟦sm⟧ = 1 for debugging code that produces exceptions. This mode of operation allows only one floating-point operation to be in the floating-point unit at a time. Thus, either 1) each floating-point operation completes with no exceptions, or 2) an exception is produced, and an interrupt is taken if that bit is 1 in register IE; no interrupt is taken if that bit is 0 in register IE. Register file write enable is inhibited so that input arguments are available for input exception interrupt handling routines.

- When floating-point interrupts are disabled, floating-point instructions that generate exceptions do not take an interrupt.
- When floating-point interrupts are enabled, a floating-point instruction that produces an exception takes a floating-point interrupt if that corresponding bit is 1 in register IE. However, if that bit is 0 in register IE, then no interrupt is taken for that floating-point exception.

  When an input exception occurs (for example, divide-by-zero, signaling NaNs) and that bit is 1 in register IE, then the write to the register destination is inhibited so that floating-point exception interrupt handling routines can examine input operands to determine the reason for the exception. Results are written to register destination in all other cases, including output exceptions (for example, **frndn.di r8,r6** when the output integer overflows the 32-bit destination register).
- Standard floating-point exception interrupt handling routine sequential mode exit includes:

```
brcr  EPC    ; branch to EPC
nop          ; branch delay slot
```

- The address in EIP should point to the instruction that caused the exception (for example, fdiv.ddd r4,r0,r8 sets divide-by-zero interrupt—INTPEN⟦fz⟧ if IE⟦fz⟧ = 1).

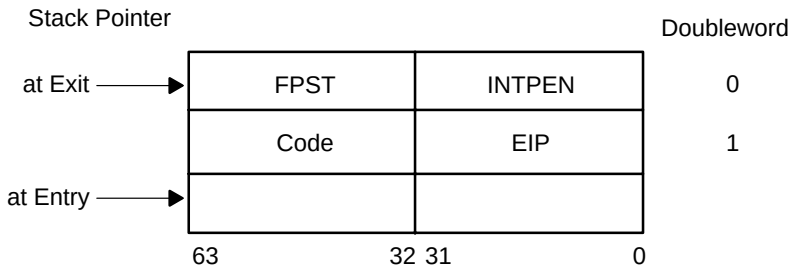
Part of a sample floating-point exception interrupt handling routine code is shown in Example 11–18.

Note:

If parallel loads coded with vector instructions (vld) are used, input variables can be overwritten by these load instructions.

Example 11–18. Floating-Point Exception Interrupt Handler Routine—Sample Code

(a) Stack information returned by this floating-point exception interrupt handler routine:



(b) Sample code segment (without context save/restore or main body):

```
FP_INEXACT:                     ; Floating-Point Inexact Interrupt

         st.d    –3*8(r1:m),r4  ; push 64-bit r4 onto stack and
                                ;     sp – 3*8 → sp
         rdcr    EIP,r4         ; EIP → r4
         or      0x7,r0,r5      ; code = 7 for this routine
         st.d    2*8(r1),r4     ; push (EIP,code) → stack

         rdcr    INTPEN,r4      ; INTPEN → r4
         or      0x80,r0,r5     ; reset bit fx = 0 in INTPEN
         wrcr    INTPEN,r5      ; INTPEN & (~r5) → INTPEN

         rdcr    FPST,r5        ; FPST → r5
         st.d    1*8(r1),r4     ; 64-bit (INTPEN,FPST) → stack

         and.ft  0x100000,r5,r5 ; reset bit fs = 0 in FPST
         wrcr    FPST,r5        ; r5 → FPST
         bbo     SeqINEX,r5,21  ; FPST sequential mode bit 21 = 0 ?
         ld.d    0(r1),r4       ; pop 64-bit r4 from stack
         addu    1*8,r1,r1      ; return sp + 1*8 → sp and
                                ;       save 4 values pushed
         brcr    EIP            ; 0 = pipelined mode standard exit
         brcr    EPC            ; restores PC

SeqINEX                         ; Sequential mode Inexact exit
         addu    1*8,r1,r1      ; return sp + 1*8 → sp
         brcr    EPC            ; 1 = sequential mode standard exit
         nop                    ; branch delay slot instruction
```

Notes: 1) If the four values saved on the stack are not needed, then replace the **addu 1*8,r1,r1** instructions with **addu 3*8,r1,r1**.

2) The choice of the code = 7 returned by this floating-point exception interrupt handling routine in the **or 0x7,r0,r5** instruction is for illustration only. The system can choose other values, including their order and number.

3) The MP must be in the supervisor mode to swap swcr or write wrcr control registers numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

11.12 Using Floating-Point Sequential Mode

This section describes the possible use of the floating-point sequential mode to aid in locating the instructions/operands causing floating-point exceptions. Initially, register FPST is cleared to 0 for the individual and accumulated floating-point faults; refer to Figure 3–2, *MP Interrupt Registers—IE and INTPEN*, and Figure 3–3, *MP Floating-Point Status Register—FPST*, for additional details.

The first step is to reset all floating-point bits in these three registers. This is shown in the following code in Example 11–19.

The program is then executed until it reaches a check point. Next, the accumulated bits in register FPST are examined to see if any errors occurred (individual floating-point interrupts are disabled) as shown in Example 11–20.

Example 11–19. Reset Floating-Point Exceptions in FPST, IE, and INTPEN

```
rdcr   FPST,r5          ; FPST → r5
and    0x000e0000,r5,r5; save fm and drm fields only
wrcr   FPST,r5          ; r5 → FPST
or     0xEC,r0,r6       ; reset fx,fu,fo,fz,fi bits
wrcr   INTPEN,r6        ; INTPEN & (~r6) → INTPEN
rdcr   IE,r7            ; IE → r7
and.ft 0xEC,r7,r7       ; reset fx,fu,fo,fz,fi bits
or     1,r7,r7          ; set bit ie = 1 (enable
                        ;  interrupts)
wrcr   IE,r7            ; r7 → IE
```

Note: The MP must be in supervisor mode to write wrcr control register numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

Example 11–20. Test Accumulated Floating-Point Error Bits in FPST

```
          rdcr   FPST,r5          ; FPST → r5
          and    0x07c00000,r5,r5 ; examine ai,az,ao,au,ax bits
          bcnd.a OK,r5,eq0.w      ; zero means no errors.
FPerror: ...                      ; nonzero = errors.
```

The program is then restarted as before, but the sequential mode is enabled, and the individual floating-point interrupts are enabled, as shown in Example 11–21.

Then when a floating-point exception occurs, the floating-point unit stalls, and for input exceptions only, **no** result is written into register destination. Bit FPST⟦fs⟧ is set to 1, along with more information about the floating-point instruction (destination, opcode, rm, and fault bits for this exception). The address of that instruction is contained in register EIP. The bit that corresponds to this interrupt is set in register INTPEN, and the program transfers to that interrupt service routine if that floating-point exception bit is enabled in the IE register.

Example 11–21. Reset Floating-Point Exceptions, Enable Sequential Mode

```
rdcr FPST,r5          ; FPST → r5
and  0x000e0000,r5,r5; save fm and drm fields only
or   0x00200000,r5,r5; set sm = 1
wrcr FPST,r5          ; r5 → FPST
or   0xEC,r0,r6       ; reset fx,fu,fo,fz,fi bits
wrcr INTPEN,r6        ; INTPEN & (~r6) → INTPEN
rdcr IE,r7            ; IE → r7
or   0xED,r7,r7       ; set fx,fu,fo,fz,fi bits and
                      ; set bit ie = 1 (enable
                      ; interrupts)
wrcr IE,r7            ; r7 → IE
```

Note: The MP must be in supervisor mode to write wrcr control register numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

The service routine examines the bits to see what corrective action (if any) can be taken. If the problem can be fixed, corrective action is taken, this interrupt's bit is reset in register INTPEN, and the program execution resumes. If there are no other exceptions (see subsection 11.11.1, *Recommended Floating-Point Exception Interrupt Procedure*), then bit fs is reset to 0, as shown in Example 11–22.

To run in the fast mode (for example, all denormal values are reset to 0), clear FPST[[fm]] to 0, as shown in Example 11–23.

For programs that require the default rounding mode to be rounding-toward-minus-infinity, Example 11–24 shows how to set register FPST field drm = 3.

Example 11–22. Reset Bit fs in Register FPST

```
rdcr    FPST,r5           ; FPST → r5
and.ft  0x00100000,r5,r5  ; reset bit fs
wrcr    FPST,r5           ; r5 → FPST
```

Note: The MP must be in **supervisor mode** to write **wrcr** control register numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

Example 11–23. Setting Fast Mode in Register FPST

```
rdcr    FPST,r5           ; FPST → r5
and.ft  0x00080000,r5,r5  ; reset bit fm = 0 (fast)
wrcr    FPST,r5           ; r5 → FPST
```

Notes: 1) The MP must be in supervisor mode to write wrcr control register numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

2) Fast mode (fm = 0) flushes denormal numbers to zero; IEEE mode (fm = 1) returns denormal numbers.

Example 11–24. Setting Default Round Mode in Register FPST

```
rdcr    FPST,r5           ; FPST → r5
or      0x00060000,r5,r5  ; set field drm = 3
wrcr    FPST,r5           ; r5 → FPST
```

Note: The MP must be in supervisor mode to write wrcr control register numbers < 0x4000 (see Table 2–1, *MP Control Register Numbers*).

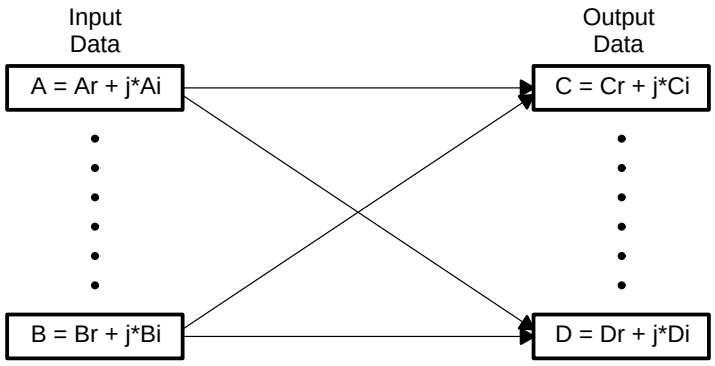
11.13 Complex FFT Butterfly

This example describes the not-in-place radix$_2$ FFT butterfly kernel for single-precision floating-point complex numbers. Radix$_2$ butterfly is used as a function call in computing a single precision, not-in-place, decimation-in-frequency Fast Fourier transform (FFT). A not-in-place algorithm is chosen because it requires two input pointers and one output pointer. The MP has the capability of performing a floating-point operation in parallel with either a load or a store, using two input pointers (IN0P or IN1P) and one output pointer (OUTP).

The function computes two butterflies in one loop to increase processor efficiency. An index table is used to look up the coefficients (cosine and sine). The sample kernel code is illustrated in Example 11–25 and the associated floating-point pipeline is shown in Example 11–26. The equations for two input complex numbers and the resulting two output numbers are given next (where j = square_root(–1) as shown in Figure 11–1):

- Input value A = Ar + j*Ai, where Ar = real, Ai = imaginary
- Input value B = Br + j*Bi, where Br = real, Bi = imaginary
- Phase shift W = Wr + j*Wi, where Wr = real (cosine), Wi = imaginary (sine)
- Output value C = Cr + j*Ci, where Cr = Ar + Br, and Ci = Ai + Bi
- Output value D = Dr + j*Di, where:
  - real Dr = (Ar – Br)*cosine + (Ai – Bi)*sine
  - imaginary Di = (Ai – Bi)*cosine – (Ar – Br)*sine

Figure 11–1. Sample Butterfly Diagram



Notes: 1) Ar, Br, Cr, and Dr are the real parts of data A, B, C, and D, respectively.

2) Ai, Bi, Ci, and Di are the imaginary parts of the A, B, C, and D, respectively.

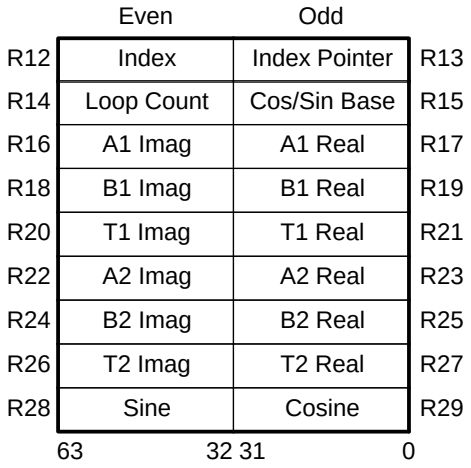
3) W = Cosine(2π*f*t) + j*sine(2π*f*t) where f is frequency and t is time.

4) j = square_root(–1).

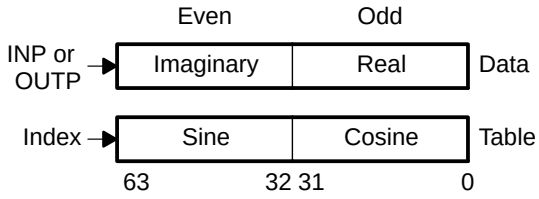
These two butterflies require 23 cycles (assuming no contentions, cache misses, exceptions, NaNs, infinities, or denormal numbers) for an average time of $\approx$11.5 cycles per butterfly. The register and memory allocations are shown in Figure 11–2. The outer loops and the bit-reversal addressing used to reorder the output data are not included here.

Figure 11–2. Two Complex Butterfly Registers (Memory Assignments)

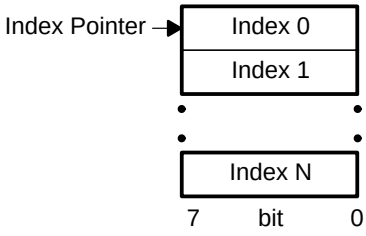
(a) Register Allocation



(b) Memory Data and Table Formats—Big Endian



(c) Index Table Required for Sine/Cosine Lookup for N < 256.



Example 11–25. Two Complex FFT Butterflies—Sample Code

```
Radix2Loop:                                ; Radix 2 loop label
  vmac.ssd  r26,r29,0,a0  || vld0.d  r16   ; 0 + Ti2*cos = a0
                                           ; 64-bit Ail = Memory(IN1P++)
  vmac.ssd  r27,r29,0,a1  || vld1.d  r18   ; 0 + Tr2*cos = a1
                                           ; 64-bit Bil = Memory(IN0P++)
  ld.b      1(r13:m),r12                   ; Index = Memory(IndexPtr++)
  vmsc.sss  r27,r28,a0,r24                 ; a0 – Tr2*sin = Bi2 output
  vmac.sss  r26,r28,a1,r25                 ; a1 + Ti2*sin = Br2 output
  ld.d      r12:s(r15),r28                 ; 64-bit cos = Memory(WPtr+8*r12)
                                         ; 1-cycle floating-point scoreboard
  fsub.sss  r16,r18,r20                    ; Ail – Bil = Til
  fsub.sss  r17,r19,r21                    ; Arl – Brl = Trl
  vadd.ss   r18,r16,r16   || vst.d   r22   ; Bil + Ail = Ail output
                                           ; Memory(OUTP++) = 64-bit Ai2
  vadd.ss   r19,r17,r17   || vst.d   r24   ; Brl + Arl = Arl output
                                           ; Memory(OUTP++) = 64-bit Bi2
  vmac.ssd  r20,r29,0,a2  || vld0.d  r22   ; 0 + Til*cos = a2
                                           ; 64-bit Ai2 = Memory (IN0P++)
  vmac.ssd  r21,r29,0,a3  || vld1.d  r24   ; 0 + Trl*cos = a3
                                           ; 64-bit Bi2 = Memory (IN1P++)
                                         ; 1-cycle floating-point scoreboard
  vmsc.sss  r21,r28,a2,r18                 ; a2 – Trl*sin = Bil
  vmac.sss  r20,r28,a3,r19                 ; a3 + Til*sin = Brl
  addu      –1,r14,r14                     ; decrement Loop Count
                                         ; 1-cycle floating-point scoreboard
  fsub.sss  r22,r24,r26                    ; Ai2 – Bi2 = Ti2
  fsub.sss  r23,r25,r27                    ; Ar2 – Br2 = Tr2
  vadd.ss   r24,r22,r22   || vst.d   r16   ; Bi2 + Ai2 = Ai2 output
                                           ; Memory(OUT++) = 64-bit Ail
  bcnd      Radix2Loop,r14,gt0.w           ; Loop if > 0
  vadd.ss   r25,r23,r23   || vst.d   r18   ; Br2 + Ar2 = Ar2 output
                                           ; Memory(OUTP++) = 64-bit Bil
```

Note: Interrupts must be enabled (IE⟦ie⟧ = 1) for all floating-point and vector instructions, including vld and vst.

Example 11–26. Floating-Point Pipeline of Two Complex FFT Butterflies

	Unpack	Floating-Point Multiply Unit		Floating-Point Add Unit			
Cycle	Execute	Multiply	Normalize	Align	Add	Normalize	
1	vmac a0	– — —	— — —	— — —	— — —	— — —	
2	vmac a1	vmac a0	— — —	— — —	— — —	— — —	
3	(ld.b)	vmac a1	vmac a0	— — —	— — —	— — —	
4	vmsc a0	— — —	vmac a1	vmac a0	— — —	— — —	
5	vmac a1	vmsc a0	— — —	vmac a1	vmac a0	— — —	
6	(ld.d)	vmac a1	vmsc a0	— — —	vmac a1	vmac a0	→ a0
7	— — —	— — —	vmac a1	vmsc a0	— — —	vmac a1	→ a1
8	fsub il	— — —	— — —	vmac a1	vmsc a0	— — —	
9	fsub rl	— — —	— — —	fsub il	vmac al	vmsc a0	→ Bi2
10	vadd il	— — —	— — —	fsub rl	fsub il	vmac a1	→ Br2
11	vadd rl	— — —	— — —	vadd il	fsub rl	fsub il	→ Ti1
12	vmac a2	— — —	— — —	vadd rl	vadd il	fsub rl	→ Tr1
13	vmac a3	vmac a2	— — —	— — —	vadd rl	vadd il	→ Ai1
14	— — —	vmac a3	vmac a2	— — —	— — —	vadd r1	→ Ar1
15	vmsc a2	— — —	vmac a3	vmac a2	— — —	— — —	
16	vmac a3	vmsc a2	— — —	vmac a2	vmac a2	— — —	
17	(addu)	vmac a3	vmsc a2	— — —	vmac a3	vmac a2	→ a2
18	— — —	— — —	vmac a3	vmsc a2	— — —	vmac a3	→ a3
19	fsub i2	— — —	— — —	vmac a3	vmsc a2	— — —	
20	fsub r2	— — —	— — —	fsub i2	vmac a3	vmsc a2	→ Bi1
21	vadd i2	— — —	— — —	fsub r2	fsub i2	vmac a3	→ Br1
22	(bcnd)	— — —	— — —	vadd i2	fsub r2	fsub i2	→ Ti2
23	vadd r2	— — —	— — —	— — —	vadd i2	fsub r2	→ Tr2
1	vmac a0	— — —	— — —	vadd r2	— — —	vadd i2	→ Ai2
2	vmac a1	vmac a0	— — —	— — —	vadd r2	— — —	
3	(ld.b)	vmac a1	vmac a0	— — —	— — —	vadd r2	→ Ar2

11.14 Using the Double-Buffer Transfer Model

In general, you should double-buffer a sequence of packet transfers so that data transfers between external memory and the MVP's on-chip RAMs can occur in parallel with data processing performed by the processor, without causing excessive crossbar contention.

Double-buffering begins when you assign one of the on-chip RAMs to the TC's output/input process and another on-chip RAM (or RAMs) to the processor's compute process. While the result of the previous compute process is output to external memory, and the next block of data in external memory is input to the on-chip RAM assigned to the TC, the processor concurrently processes the current data in other RAM(s).

At the conclusion of the process, the RAM assignments swap so that the RAM containing processed data is assigned to the TC, and the RAM containing the newly acquired input data is assigned to that processor. An example of the double-buffer transfer model is shown in Figure 11–4.

By transferring entire or partial rows of pixels, you can efficiently manage image-processing algorithms that operate on small blocks (or **kernels**) of pixels. This section illustrates how to do this for the Roberts Edge Detection algorithm, which uses a 2×2 block.

Since Roberts 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 on-chip RAMs. While data from two rows are processed by the MP, the TC, in parallel with the MP, first outputs a processed row of data, then inputs a new row of data. This process is summarized as follows:

1) Initially, two entire rows (or partial rows, if the row width is greater than 1K bytes) are brought into the on-chip RAMs.

2) While the first edge-enhanced output row is computed, the third row in the image is input to the on-chip RAMs.

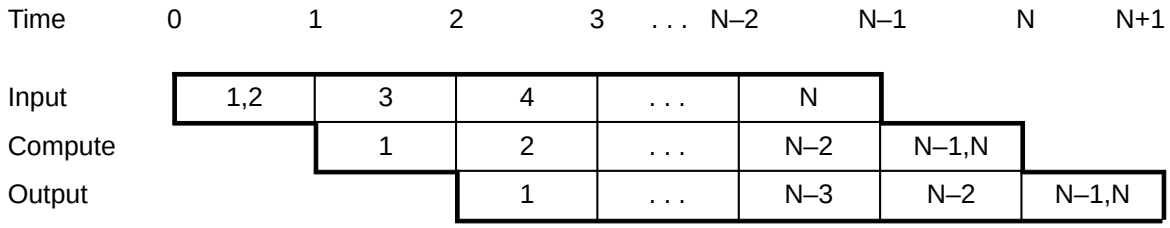
3) Next, while the second output row is computed, the first edge-enhanced row of pixels is transferred to external memory and the fourth row in the image is brought into the on-chip RAMs.

This represents a pipelining of three stages to achieve maximum performance: transferring a row of data from external memory into the on-chip RAMs, processing on the row of data, and transferring an output row of data from the on-chip RAMs to external memory. These three stages are referred to as input, compute, and output in Figure 11–3.

For this example, while row N is being completed, the output packet transfer for row N–1 and the input packet transfer for row N+2 can be linked together. Once the output packet transfer is completed, the input packet transfer request is submitted automatically.

Memory is allocated as shown in Figure 11–4 to eliminate the possibility of contention between TC crossbar accesses and processor crossbar accesses. Note that one of the three local data RAMs is always dedicated exclusively to the TC output/input.

Figure 11–3. Input, Compute, Output Pipeline Stages

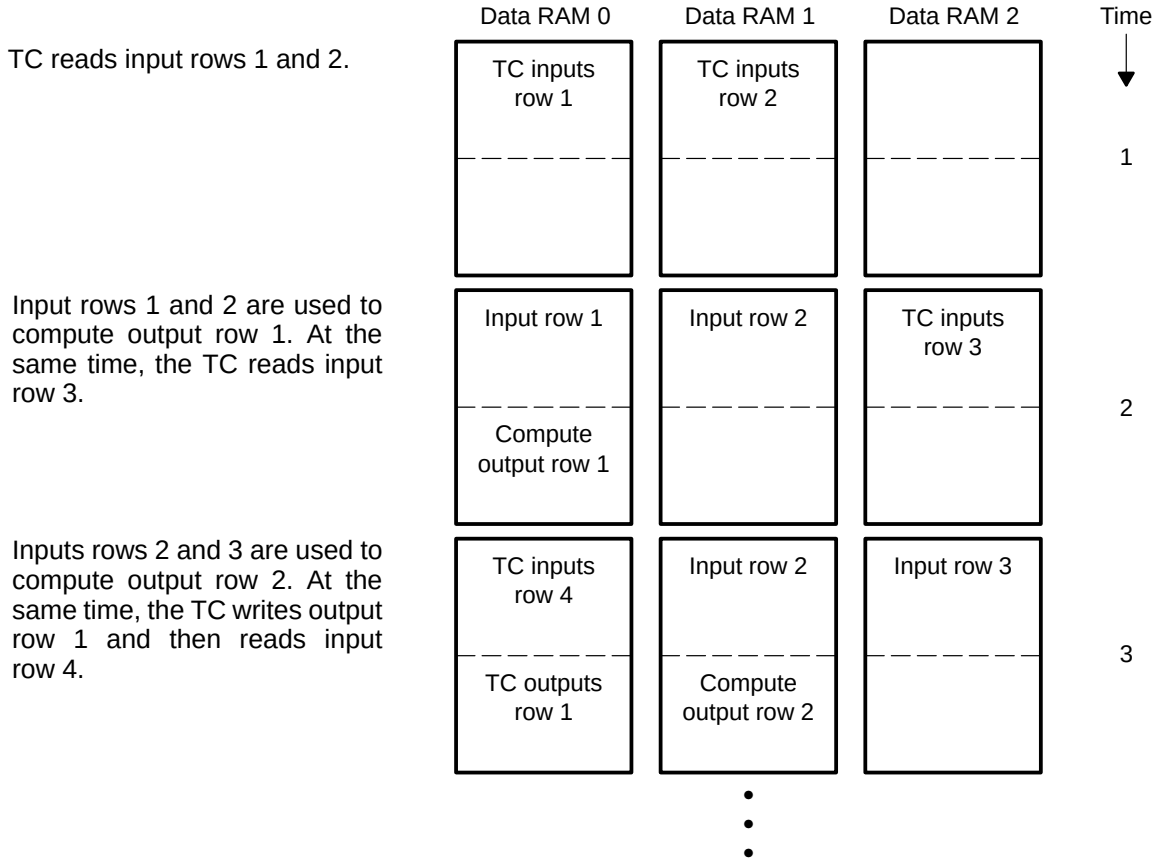


The data being transmitted by the linked output and input packet transfers are read into or written from the upper and lower halves, respectively, of the same RAM module. Since the two packet transfers are sequential (they do not overlap in time), they will not contend with each other. Note that the MP 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; thus, no contention occurs.

For kernels requiring more rows of data (for example, three rows for Sobel Edge Detection), the number of input buffers required increases, and the above method of data management becomes increasingly more difficult. At some point, depending on the row size of the image, it becomes more practical to deal with individual blocks of data instead of rows.

The description above could just as easily apply to columns of data instead of rows. However, it is more efficient to transfer rows of data when possible, because they frequently reside in the same page in external memory, while a column of data is generally not contained in a single page.

Figure 11–4. Sample Memory Allocation (Double Buffering)



11.15 Communications Between the MP and PP: A Sample Program

This sample program demonstrates an approach where the MP calls a PP program; that PP program is executed and then sends a message to the MP; then the MP uses the results of the PP's program. The steps include:

Step 1: The MP resets PP0 (and waits for PP0 to set the halt bit in the PPERROR register)

Step 2: The MP initializes the PP task interrupt address (0x0100 01B8 = PP0_START)

Step 3: The MP sends a task command to PP0 (unhalts PP0 to begin program execution)

Step 4: PP0 sends a message interrupt to the MP (indicates PP0 program completion)

Step 5: The MP uses the results of the PP0 program

You can assemble and link this program, and then use the parallel debug manager and C source debuggers to simulate the sample program (see subsections 11.15.6 and 11.15.7).

11.15.1 Step 1: The MP Resets PP0 and Waits for PP0 to Halt

The following section of the mainMP.asm MP code illustrates how the MP resets PP0 and then waits until PP0 has halted (when bit 16 in the PPERROR register is set to 1 at label wait1).

```
; mainMP.asm     MP resets, calls, and waits for PP0 message
;               MP then calls the MPcall subroutine and ends.

        .global  _main             ; MP main entry
        .global  wait1             ; wait 1 for PP0 to halt after reset
        .global  wait2             ; wait 2 for PP0 to send message to
                                   ;  MP
        .global  END               ; end of the MP main program
        .global  MPcall            ; main calls MPcall subroutine
        .global  PP0_START         ; PP0 program start address

_main:                             ; MP main entry point
        wrcr     IE,r0             ; disable interrupts
        addu     0x80000001,r0,r3  ; Reset PP0 command word
        cmnd     r3                ; issue Reset PP0 command
        addu     –1,r0,r2          ; r2 = –1
        wrcr     INTPEN,r2         ; clears all pending interrupts

wait1:  rdcr     PPERROR,r4        ; r4 = PPERROR control register
        bbz.a    wait1,r4,16       ; spin if bit 16 = 0 (PP0 has not
                                   ;  halted)
```

11.15.2 Step 2: The MP Initializes the PP Task Interrupt Address and Unhalts PP0

The following section of mainMP.asm MP code demonstrates the MP store of the PP0 entry address PP0_START into PP0's parameter RAM task interrupt location 0x0100 01B8. Then, the MP unhalts PP0 by issuing a cmnd instruction so that the PP0 program execution can begin.

```
addu  PP0_START,r0,r5    ; PP0 program entry address
st    0x010001b8(r0),r5  ; PP0 task interrupt location

addu  0x20000001,r0,r6   ; Unhalt PP0 command word
cmnd  r6                 ; issue Unhalt PP0 command
```

11.15.3 Step 3: The MP Waits for a PP0 Message and Then Calls MP Subroutine

The following section of mainMP.asm MP code illustrates the MP waiting for PP0's message interrupt to the MP (bit 16 is 1 in INTPEN). Once this is received, the MP can issue a subroutine call to MPcall. When subroutine MPcall execution is complete, that program returns to END. This concludes the mainMP.asm programming example.

```
wait2:  rdcr    INTPEN,r7     ; r7 = INTPEN register
        bbz.a   wait2,r7,16   ; test bit 16 (0=no PP0 message to MP)

        jsr.a   MPcall,r31    ; call MP subroutine MPcall

wait3:  rdcr    PPERROR,r4    ; r4 = PPERROR control register
        bbz.a   wait3,r4,16   ; wait until PP0 has halted

END:    br.a    END           ; spin loop
        nop                   ; branch delay slot as legal MP instruction
```

11.15.4 Step 4: PP0 Sends a Message Interrupt to the MP

The following section of PP0run.s PP code is an example of a program that stores some numbers in PP0's data RAM0 location 0x100 and then issues a message interrupt to the MP (sets PP0 message bit 16 in the INTPEN register). After a few more PP instructions, PP0 issues a halt command to itself and halts (sets PP0 halt bit 16 in PPERROR register and **no** interrupt is signaled) before the first instruction after the cmnd instruction is executed.

```
; PP0run.s      PP0 stores numbers into 0x100 and then sends a message
;               to the MP. PP0 also halts itself.

        .global  PP0_START     ; PP0 entry address

out:    .set     0x100         ; PP0 Data RAM0 location

PP0_START:                     ; PP0 program entry point
        d1 = 0x1111
        d2 = 0x2222
        d3 = 0x3333
        d4 = 0x4444
        d5 = 0x5555
        d6 = 0x2100            ; PP0 message to MP
        d7 = 0x40002001        ; Halt message to PP0
        *(out+[0]) = d1        ; Mem(0x100) = d1
        *(out+[1]) = d2        ; Mem(0x104) = d2
        *(out+[2]) = d3        ; Mem(0x108) = d3
        *(out+[3]) = d4        ; Mem(0x10c) = d4
        *(out+[4]) = d5        ; Mem(0x110) = d5
        cmnd = d6              ; issue PP0 message to MP command
        cmnd = d7              ; issue PP0 halt message command

; PP0 halts at this point. If it is unhalted without task signal, then
;  program execution begins with the next instruction. If a task signal
;  is used, then program starts at the location found in 0x0100 #1B8
;  (task interrupt location) where # is the PP number.

        nop                    ; first delay slot
        nop                    ; second delay slot
        *(out+[5]) = d6        ; Mem(0x114) = d6
        *(out+[6]) = d7        ; Mem(0x118) = d7
```

11.15.5 Step 5: The MP Uses the Results of the PP0 Program

The following section of MP subroutine MPcall.asm code illustrates how the MP program loads some numbers from PP0's data RAM0 location 0x100 and then returns to the mainMP program that called it.

```
; MPcall.asm      MP subroutine that loads registers from PP0's data
;                 RAM0 region (see PPrun.s)

         .global  MPcall           ; MP subroutine entry

out:     .set     0x100            ; PP0 Data RAM0 location

MPcall:                           ; Subroutine entry label

         ld       out+0(r0),r10    ; r10 = 0x1111
         ld       out+4(r0),r11    ; r11 = 0x2222
         ld       out+8(r0),r12    ; r12 = 0x3333
         ld       out+12(r0),r13   ; r13 = 0x4444
         ld       out+16(r0),r14   ; r14 = 0x5555
         ld       out+20(r0),r15   ; r15 = Mem(0x114)
         ld       out+24(r0),r16   ; r16 = Mem(0x118)
         ld       out+28(r0),r17   ; r17 = Mem(0x11c)
         jsr.a    r31(r0),r0       ; subroutine routine
         nop                       ; branch delay slot
```

11.15.6 Assemble and Link the MP and PP Sample Programs

You can assemble the three assembly language programs described in the previous subsections. From a command shell, enter:

```
mpasm mainMP.asm ↵          ; this creates mainMP.obj
mpasm MPcall.asm ↵          ; this creates MPcall.obj
ppasm PP0run.s ↵            ; this creates PP0run.obj
```

Next, link the .obj files to form MPPP.out:

```
mvplnk –o MPPP.out mainMP.obj MPcall.obj PP0run.obj –e main ↵
```

11.15.7 Simulate MP and PP Sample Programs Using the PDM

You can use the parallel debug manager (PDM) with the MP and PP debuggers to simulate the MP and PP sample programs. Use the init.pdm file that you created in the advanced tutorial (see the *Create a batch file for repetitive tasks* discussion on page NO TAG of the *MVP C Source Debugger User's Guide*). Follow these steps to test the MPPP.out program:

1) To start the PDM session, enter the following from a command shell:

```
pdm ⏎
```

2) In the COMMAND window of the PP0 debugger, enter:

```
load MPPP.out ⏎
ba PP0_START ⏎
mem 0x100 ⏎
```

This loads the MPPP.out object file into the PP debugger and sets a breakpoint at PP0_START.

3) In the COMMAND window of the MP debugger, enter:

```
load MPPP.out ⏎
ba MPcall ⏎
ba wait1 ⏎
ba wait2 ⏎
ba END ⏎
mem 0x010001b8 ⏎
```

This loads the MPPP.out object file into the MP debugger and sets various breakpoints.

4) In the COMMAND window of the MP debugger, run the program:

```
run ⏎
```

When the program halts at the breakpoint set at wait1, run the program again:

```
run ⏎
```

Wait until the program halts at the breakpoint set at wait2.

5) In the COMMAND window of the PP0 debugger, enter:

```
step ⏎
```

This causes PP0 to step to PP0_START.

Repeat the STEP command until PP0 reaches the second cmnd instruction:

```
step ⏎
```

6) In the COMMAND window of the MP debugger, run the program again:

   run ⏎

   When the program halts at the breakpoint set at MPcall, run the program again:

   run ⏎

   Wait until the program halts at the breakpoint set at END.

7) Look at the CPU window of the MP debugger. Check the contents of the following registers:

```
r10 = 0x1111
r11 = 0x2222
r12 = 0x3333
r13 = 0x4444
r14 = 0x5555
r15,r16,r17 values are not specified
```

8) Close the parallel debug manager and the debuggers by entering the following from the command line of the parallel debug manager:

   quit ⏎

Appendix A

Understanding the Floating-Point Numbering System

This appendix briefly describes the floating-point numbering system. For a complete description, refer to the IEEE-754 standard. The number formats and types for the MP are single-precision (32-bit) and double-precision (64-bit).

Topics

A.1 Formats for Floating-Point Numbers

A floating-point number is composed of three parts: the sign, the mantissa, and the exponent. The value of the number is presented by the following:

$$-1^{sign} \times mantissa \times radix^{exponent}$$

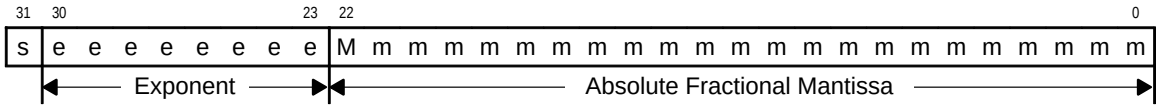
The IEEE-754 standard uses a binary radix of 2 and a sign magnitude form for the mantissa. Single- and double-precision formats are shown in Figure A–1 and Figure A–2, respectively.

If the exponent is nonzero, the mantissa value is 1 (the 1 is referred to as the **hidden** bit) plus the fractional mantissa, where the single-precision exponent bias is 127 and the double-precision exponent bias is 1023.

There are several number categories within each precision. Some categories represent ranges of numbers, and others are abstract types. This chapter describes the following categories:

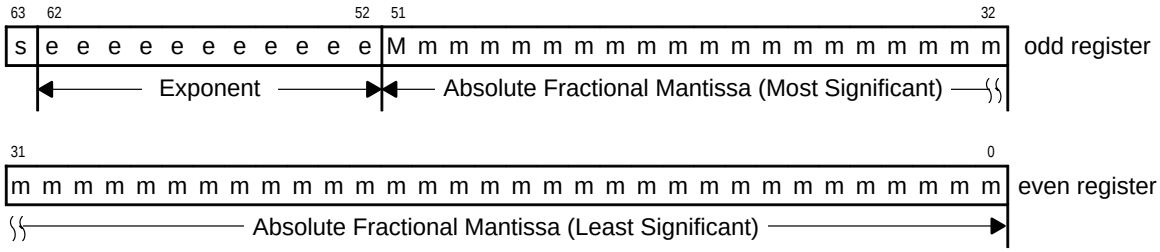
- Normal numbers
- Denormal numbers
- Infinities
- Not-a-numbers (NaNs)
- Wrapped numbers

Figure A–1. Single-Precision Format Example—Packed Notation



Note: s = sign e = exponent
m = mantissa M = most significant fractional mantissa bit

Figure A–2. Double-Precision Format Example—Packed Notation



Note: s = sign e = exponent
m = mantissa M = most significant fractional mantissa bit

A.2 Normal Numbers

Normal numbers are numbers within the range of the given precision. The term normalized comes from the fact that the mantissa of a normalized number ranges from 1 to 2 ($1 \leq mantissa < 2$). This fact is used in the multiply and divide algorithms; since the value is between 1 and 2, the MSB of the mantissa is always 1. This fact is exploited to avoid explicitly encoding the MSB—it is called the implied or hidden bit (HB). When a number is written and the HB not explicitly shown, the number is said to be in a packed format. Conversely, if the HB is explicitly shown, the number is said to be in an unpacked format.

The IEEE standard uses a biased exponent—an exponent with a constant offset or bias. The result of using a biased exponent is that all of the exponents are positive numbers. The single-precision bias is 127, and the double-precision bias is 1023.

Example A–1 and Example A–2 illustrate normal numbers in single-precision and double-precision format, respectively. For a bit representation of single-precision and double-precision formats, see Figure A–1 and Figure A–2, respectively.

Example A–1. Normal Numbers in Single-Precision Format (Packed Notation)

0x3F80 0000 = +1.00
0xBF40 0000 = –0.75

Note: s = 0, 1
0<e<0xFF
M and m are don't cares
HB = 1, bias = 127
The values of the floating-point number = $-1^{s} \times 2^{(e-127)} \times 1.Mm$

Example A–2. Normal Numbers in Double-Precision Format (Packed Notation)

0x3FF0 0000 0000 0000 = +1.00
0xBFE8 0000 0000 0000 = –0.75

Note: s = 0, 1
0<e<0x7FF
M and m are don't cares
HB = 1, bias = 1023
The values of the floating-point number = $-1^{s} \times 2^{(e-1023)} \times 1.Mm$

A.3 Denormal Numbers

Denormal numbers fall between the minimum normal number and zero. They are identified in the packed format by an exponent of all 0s and a non-0 mantissa. For denormal numbers, the HB is 0. The real value of the biased exponent is 1, but the exponent is stored in the register file in packed format as a 0 so that denormal numbers can be readily distinguished from normal numbers.

Example A–3 and Example A–4 illustrate denormal numbers in single-precision and double-precision format, respectively. Example A–5 and Example A–6 illustrate 0s in single-precision and double-precision format, respectively. For a bit representation of single-precision and double-precision formats, see Figure A–1 and Figure A–2, respectively.

Example A–3. Denormal Numbers in Single-Precision Format (Packed Notation)

$0x0000\ 0001 = +2^{-149}$
$0x807F\ FFFF = -(2^{-126}-2^{-149})$

Note: s = 0, 1
e = 0
Mm $\neq$ 0
HB = 0
The values of the floating-point number=$-1^{s} \times 2^{-126} \times 0.Mm$

Example A–4. Denormal Numbers in Double-Precision Format (Packed Notation)

$0x0000\ 0000\ 0000\ 0001 = +2^{-1074}$
$0x800F\ FFFF\ FFFF\ FFFF = -(2^{-1022}-2^{-1074})$

Note: s = 0, 1
e = 0
Mm $\neq$ 0
HB = 0
The values of the floating-point number = $-1^{s} \times 2^{-1022} \times 0.Mm$

Example A–5. Zeros in Single-Precision Format (Packed Notation)

$0x8000\ 0000 = -0.0$
$0x0000\ 0000 = +0.0$

Note: s = 0, 1
e = 0
Mm = 0
HB = 0

Example A–6. Zeros in Double-Precision Format (Packed Notation)

$0x8000\ 0000\ 0000\ 0000 = -0.0$
$0x0000\ 0000\ 0000\ 0000 = +0.0$

Note: s = 0, 1
e = 0
Mm = 0
HB = 0

A.4 Infinities

An infinity value is a type of floating-point number with special features that correspond to the mathematical concept of infinity. An infinity is a number whose exponent is all 1s and whose packed mantissa is 0. It can have either sign, and the HB is 0.

Example A–7 and Example A–8 illustrate infinity numbers in single-precision and double-precision format, respectively. For a bit representation of single-precision and double-precision formats, see Figure A–1 and Figure A–2, respectively.

Example A–7. Infinity Numbers in Single-Precision Format (Packed Notation)

0x7F80 0000 = $+\infty$
0xFF80 0000 = $-\infty$

Note: s = 0, 1
e = 0xFF
M = m = 0
HB = 0

Example A–8. Infinity Numbers in Double-Precision Format (Packed Notation)

0x7FF0 0000 0000 0000 = $+\infty$
0xFFF0 0000 0000 0000 = $-\infty$

Note: s = 0, 1
e = 0x7FF
M = m = 0
HB = 0

A.5 Not-a-Number (NaN)

The NaN (not-a-number) has no numerical value but does have special meanings. There are two type of NaNs: signaling NaN (SNaN) and quiet NaN (QNaN).

The definition of a SNaN is not specified by the IEEE-754 standard, but the standard does require that a system support both signaling and quiet NaNs. In general, NaNs are defined as a number with all 1s in the exponent and a non-0 mantissa; the MSB of the packed mantissa (bit M) differentiates between SNaN (M = 0) and QNaN (M = 1).

- If the SNaN is an operand to a floating-point operation, then the invalid operation exception is signaled, and a QNaN is returned.
- If a QNaN is an operand to a floating-point operation (and the second operand is not SNaN), then the invalid operation exception is **not** signaled, and a QNaN is returned.
- If two QNaNs are operands to a floating-point instruction, then the source1 QNaN is returned. For mixed precisions, single-precision $\rightarrow$ double-precision right-fills the mantissa with 0s, while double-precision $\rightarrow$ single-precision truncates the mantissa; the input sign is copied, and the output exponent is all 1s. The output of all invalid operations is a QNaN, and to simplify design, it will always be the same QNaN (that QNaN is all 1s).

The NaN sign is a don't care. The NaN exponent is all 1s. The NaN hidden bit (HB) is 1. The packed MSB of a signaling NaN = 0, and the mantissa $\neq$ 0. The packed MSB of a quiet NaN = 1, and the mantissa is a don't care.

Example A–9 and Example A–10 illustrate signaling not-a-numbers in single-precision and double-precision format, respectively. Example A–11 and Example A–12 illustrate quiet not-a-numbers in single-precision and double-precision format, respectively. For a bit representation of single-precision and double-precision formats, see Figure A–1 and Figure A–2, respectively.

Example A–9. Signaling NaNs in Single-Precision Format (Packed Notation)

0x7F80 0001 = no value
0xFFBF FFFF = no value

Note: s = 0, 1
e = 0xFF
M = 0
m $\neq$ 0
HB = 1

Example A–10. Signaling NaNs in Double-Precision Format (Packed Notation)

0x7FF0 0000 0000 0001 = no value
0xFFF7 FFFF FFFF FFFF = no value

Note: s = 0, 1
e = 0x7FF
M = 0
m ≠ 0
HB = 1

Example A–11. Quiet NaNs in Single-Precision Format (Packed Notation)

0x7FC0 0000 = no value
0xFFFF FFFF = no value

Note: s = 0, 1
e = 0xFF
M = 1
m = don’t care
HB = 1

Example A–12. Quiet NaNs in Double-Precision Format (Packed Notation)

0x7FF8 0000 0000 0000 = no value
0xFFFF FFFF FFFF FFFF = no value

Note: s = 0, 1
e = 0x7FF
M = 1
m = don’t care
HB = 1

A.6 Wrapped Numbers

The wrapped format represents numbers that have exceeded the exponent range for a particular format (single-precision or double-precision). For example, when a single-precision denormalized number is normalized, the exponent becomes too small (<0x01) for the standard single-precision exponent to represent. The exponent underflow is then represented in a wrapped format, meaning that the exponent has gone below a biased value of 1.

The biased single-precision exponent first becomes the value 0, and then it becomes negative (0xFF). In the single-precision wrapped format, a biased value of 0x00 is equal to an unbiased value of –127, and a biased value of wrapped 0xFF is equal to an unbiased value of –128. The exponent wraps around 0 and starts over from the maximum exponent (0xFF).

When numbers become too large to represent, exponent overflow wraps around the largest exponent (0xFF in single-precision) and starts over with the smallest exponent (0x00). Associated with that operand is a signal (not visible to user) that indicates to the floating-point unit that the number has wrapped.

Example A–13 illustrates exponent wrapping for single-precision exponent overflow and underflow, respectively. Single precision uses an 8-bit biased exponent:

- The largest biased exponent that has not overflowed is 0xFE.
- The smallest biased exponent that has not underflowed is 0x01.

If a number has not overflowed or underflowed, then the exponent values 0x00 and 0xFF have special meanings (denormal and NaN/infinity, respectively).

Example A–13. Wrapped Numbers in Single-Precision Format

(a) Large exponents and their values

Exponent	No Overflow Biased	No Overflow Unbiased	Overflow Biased	Overflow Unbiased	
0xFD	253	126	253†	126†	not wrapped
0xFE	254	127	254†	127†	not wrapped
0xFF	special	special	255	128	1st wrapped
0x00	special	special	256	129	2nd wrapped
0x01	1	–126	257	130	3rd wrapped
0x02	2	–125	258	131	4th wrapped

(b) Small exponents and their values

Exponent	No Underflow Biased	No Underflow Unbiased	Underflow Biased	Underflow Unbiased	
0x02	2	–125	2†	–125†	not wrapped
0x01	1	–126	1†	–126†	not wrapped
0x00	special	special	0	–127	1st wrapped
0xFF	special	special	–1	–128	2nd wrapped
0xFE	254	127	–2	–129	3rd wrapped
0xFD	253	126	–3	–130	4th wrapped

† These exponents have not overflowed/underflowed.

For information on results with interrupts enabled, see subsection C.1.2, *Output Exceptions With Interrupts Enabled*.

Example A–14 illustrates exponent wrapping for double-precision exponent overflow and underflow, respectively. Double precision uses an 11-bit biased exponent:

- The largest biased exponent that has not overflowed is 0x7FE.
- The smallest biased exponent that has not underflowed is 0x001.

If a number has not overflowed or underflowed, then the exponent values 0x000 and 0x7FF have special meanings (denormal and NaN/infinity, respectively).

Example A–14. Wrapped Numbers in Double-Precision Format

(a) Large exponents and their values

Exponent	No Overflow Biased	No Overflow Unbiased	Overflow Biased	Overflow Unbiased	
0x7FD	2045	1022	2045†	1022†	not wrapped
0x7FE	2046	1023	2046†	1023†	not wrapped
0x7FF	special	special	2047	1024	1st wrapped
0x000	special	special	2048	1025	2nd wrapped
0x001	1	–1022	2049	1026	3rd wrapped
0x002	2	–1021	2050	1027	4th wrapped

(b) Small exponents and their values

Exponent	No Underflow Biased	No Underflow Unbiased	Underflow Biased	Underflow Unbiased	
0x002	2	–1021	2†	–1021†	not wrapped
0x001	1	–1022	1†	–1022†	not wrapped
0x000	special	special	0	–1023	1st wrapped
0x7FF	special	special	–1	–1024	2nd wrapped
0x7FE	2046	1023	–2	–1025	3rd wrapped
0x7FD	2045	1022	–3	–1026	4th wrapped

† These exponents have not overflowed/underflowed.

For information on results with interrupts enabled, see subsection C.1.2, *Output Exceptions With Interrupts Enabled*.

Wrapped numbers are only internal to the floating-point unit with one exception: some vector operations can produce a wrapped number as a result from the floating-point unit. For more information about vector exceptions, see Section C.2, *Vector Operation Exceptions*.

When the underflow or overflow interrupt is enabled, the floating-point unit returns a wrapped result to the interrupt handler if an overflow or underflow interrupt is taken. See Section C.5, *Results From Floating-Point Unit Exceptions*, for an explanation of the type of result that is returned to the interrupt handlers when floating-point unit interrupts are enabled.

Appendix B

Pipeline Implications of Floating-Point Denormals

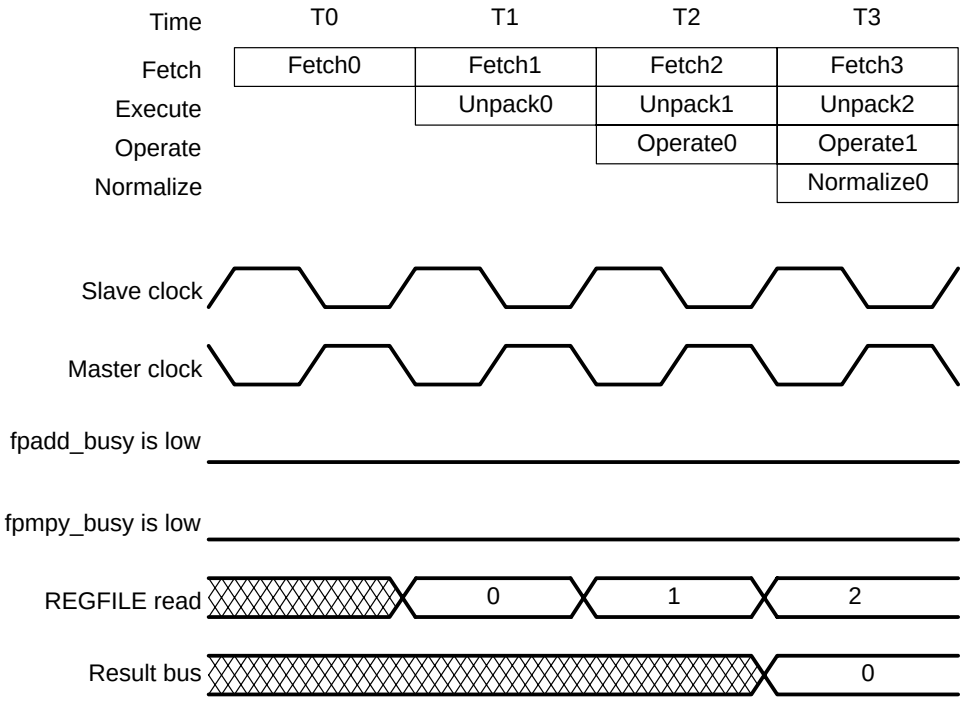
This appendix provides examples of floating-point unit pipelines.

Topics

B.1 Normal Floating-Point Multiply (No Denormals)

Figure B–1 shows the normal floating-point multiply pipeline when there are no denormal inputs or outputs.

Figure B–1. Normal Floating-Point Multiply Pipeline (No Denormals)



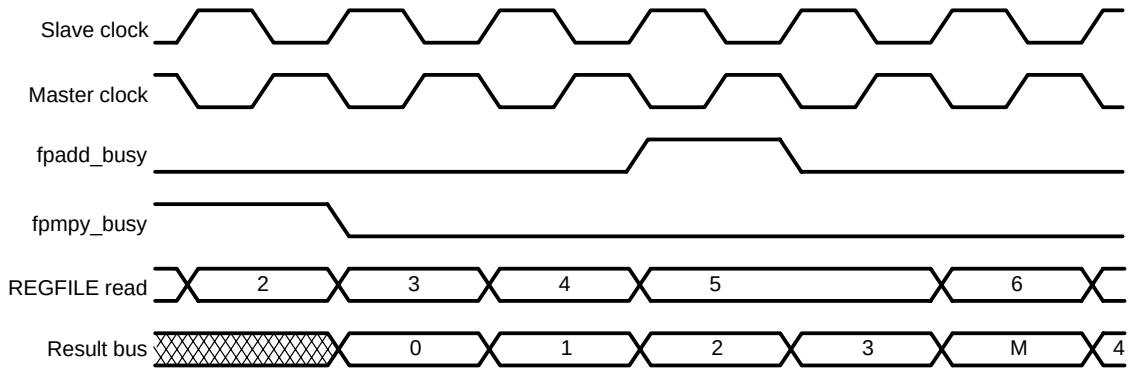
B.2 Floating-Point Multiply With One Output Denormal

Figure B–2 shows an output denormal from the floating-point multiply unit being processed by the floating-point add unit when the pipeline is fully piped with floating-point add operations.

- FetchM (not shown) is a multiple-cycle floating-point multiply instruction with an output denormal.
- ADD0–ADD5 are single-precision/double-precision floating-point add instructions.
- Fetch6/7 are don't care values.
- The FPST information, which corresponds to the FetchM instruction, is piped through the floating-point add unit.
- Signal fpadd_busy is 1 during T3 because at T2, the floating-point unit is unpacking a floating-point add instruction, and the floating-point multiply unit indicated that NormalizeM would require the floating-point add unit.

Figure B–2. Floating-Point Multiply Pipeline With One Output Denormal

Time	T0	T1	T2	T3	T4	T5
Fetch	ADD3	ADD4	ADD5	Fetch6	Fetch6	Fetch7
Execute	Unpack2	Unpack3	Unpack4	Unpack5	Unpack5	Unpack6
FP-MPY Operate	OperateM	OperateM	Idle	Idle	Idle	Idle
FP-MPY Normalize	Idle	Idle	NormalizeM	Idle	Idle	Idle
FP-ADD Align	Align1	Align2	Align3	Fix up M	Align4	Align5
FP-ADD Operate	Operate0	Operate1	Operate2	Operate3	Fix up M	Operate4
FP-ADD Normalize	Idle	Normalize0	Normalize1	Normalize2	Normalize3	Fix up M



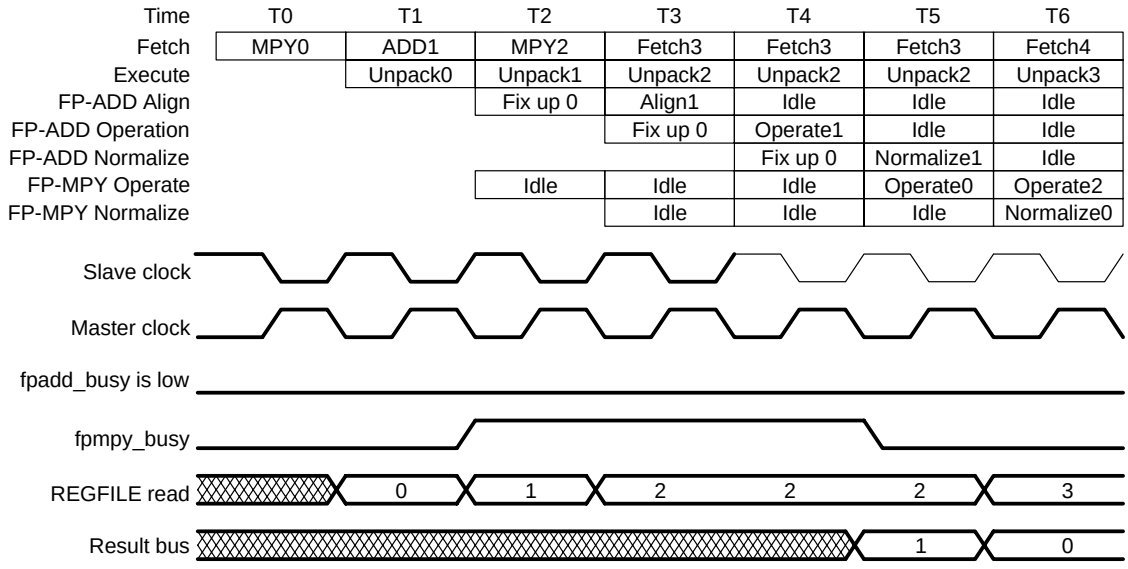
Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit

B.3 Floating-Point Multiply With One Input Denormal

Figure B–3 shows what happens if there is one denormal input to the floating-point multiply unit from MPY0, which is a single-precision multiply.

- The denormal number is sent to the floating-point add unit (instead of the floating-point multiply unit) to be normalized.
- The resulting wrapped number from the normalization stage is sent to the first pipeline stage of the floating-point multiply unit.
- ADD1 is a floating-point add instruction that can proceed in T2.
- MPY2 requires the floating-point multiply unit, and as a result, stalls the execution unit.
- It is assumed that no vmac-type instructions are occurring in the floating-point multiply/floating-point add pipeline.

Figure B–3. Floating-Point Multiply Pipeline With One Input Denormal



Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit

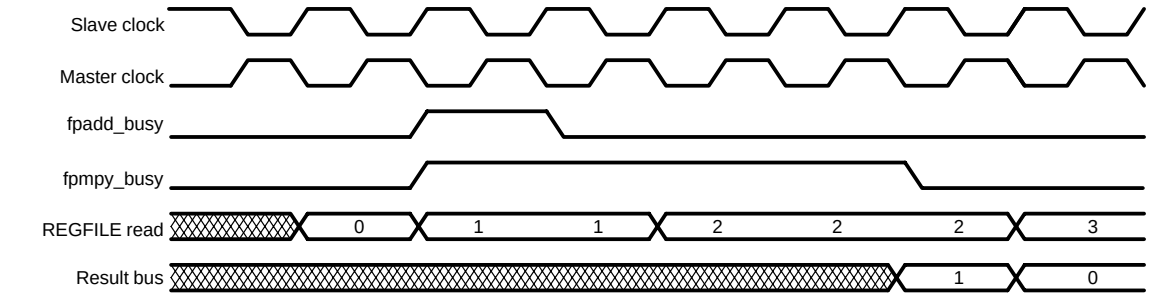
B.4 Floating-Point Multiply With Two Input Denormals

Figure B–4 shows what happens if there are two denormal inputs to the floating-point multiply unit from MPY0, which is a single-precision multiply.

- ADD1 is a floating-point add instruction that causes the execution unit to stall.
- MPY2 requires the floating-point multiply unit, and as a result, stalls the execution unit because the floating-point multiply unit is still waiting for the denormals to be fixed.
- It is assumed that no vmac-type instructions are occurring in the floating-point multiply/floating-point add pipeline.

Figure B–4. Floating-Point Multiply Pipeline With Two Input Denormals

Time	T0	T1	T2	T3	T4	T5	T6	T7
Fetch	MPY0	ADD1	MPY2	MPY2	Fetch3	Fetch3	Fetch3	Fetch4
Execute		Unpack0	Unpack1	Unpack1	Unpack2	Unpack2	Unpack2	Unpack3
FP-ADD Align			Fix up 0	Fix up 0	Align1	Idle	Idle	Idle
FP-ADD Operation				Fix up 0	Fix up 0	Operate1	Idle	Idle
FP-ADD Normalize					Fix up 0	Fix up 0	Normalize1	Idle
FP-MPY Operation			Idle	Idle	Idle	Idle	Operate0	Operate2
FP-MPY Normalize				Idle	Idle	Idle	Idle	Normalize0



Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit

B.5 Denormal With vmac-Type Instructions

If a vmac-type instruction (vmac, vmsc, or vmsub) is in the floating-point multiply/floating-point add pipeline and an input denormal to the floating-point multiply unit must be normalized, the denormal number is sent to the floating-point add unit after the vmac-type instructions that are in the floating-point multiply/floating-point add pipeline have passed the align stage in the floating-point add unit.

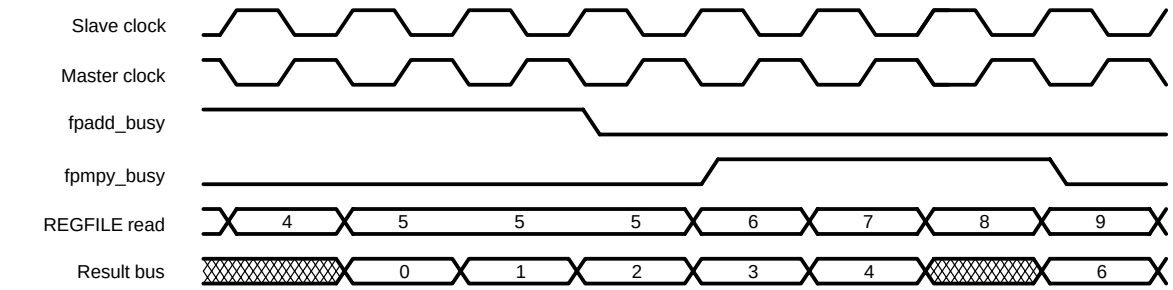
Figure B–5 shows what happens if there is one denormal input to the floating-point multiply unit from MPY5.

- MPY0–MPY4 are vmac-type instructions.
- MPY5 can be any floating-point multiply instruction (if MPY5 is a vmac-type instruction, fpadd_busy would go to 1 again at T5).
- ADD6–ADD8 can be any floating-point add instruction.
- Fetch9 is a don't care value.
- Signal fpadd_busy stays a 1 during vmac-type instructions because the instructions use the floating-point add unit.

Refer to Figure B–6 to see the normal pipeline for vmac-type instructions that require the floating-point multiply and the floating-point add units.

Figure B–5. Denormal With vmac-Type Instructions

Time	T0	T1	T2	T3	T4	T5	T6	T7	T8
Fetch	MPY4	MPY5	ADD6	ADD6	ADD6	ADD7	ADD8	Fetch9	Fetch10
Execute		Unpack4	Unpack5	Unpack5	Unpack5	Unpack6	Unpack7	Unpack8	Unpack9
FP-MPY Operation		Operate3	Operate4	Idle	Idle	Idle	Idle	Idle	Operate5
FP-MPY Normalize		Normalize2	Normalize3	Normalize4	Idle	Idle	Idle	Idle	Idle
FP-ADD Align		Operate1	Operate2	Operate3	Operate4	Fix up 5	Operate6	Operate7	Operate8
FP-ADD Operation		Operate0	Operate1	Operate2	Operate3	Operate4	Fix up 5	Operate6	Operate7
FP-ADD Normalize		Idle	Normalize0	Normalize1	Normalize2	Normalize3	Normalize4	Fix up 5	Normalize6



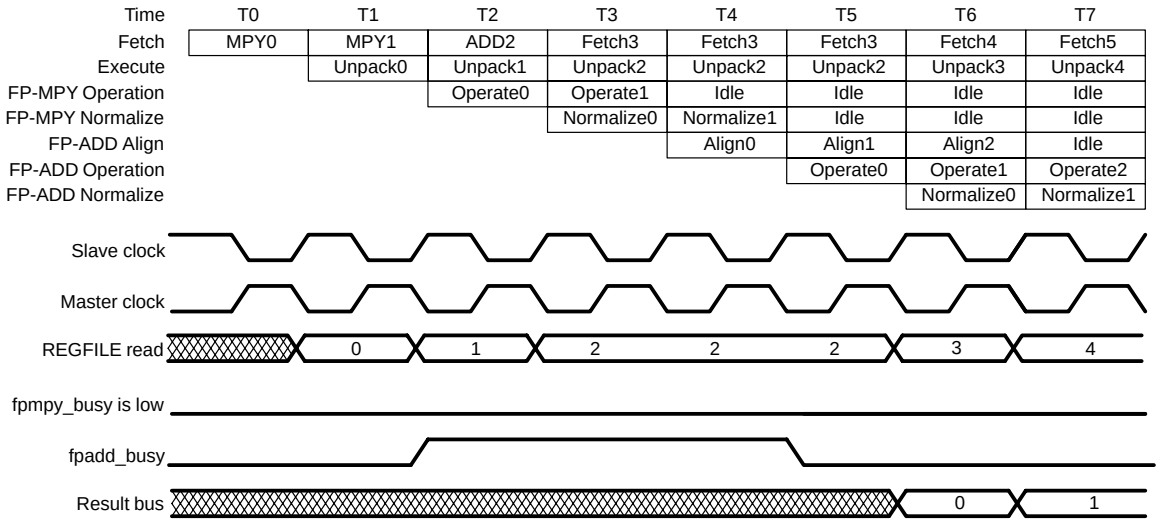
Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit

B.6 Vector Instruction Pipeline—vmac

The floating-point unit pipeline, which is used for vector instructions that require both the floating-point multiply and floating-point add pipelines, is shown in Figure B–6.

- MPY0 and MPY1 are both vmac-type operations.
- ADD2 is a floating-point add operation that is being sent to the floating-point add unit while that unit is busy, causing the execution unit to stall.
- Fetch3 and Fetch4 are don't care values.

Figure B–6. Vector Instruction Pipeline—vmac



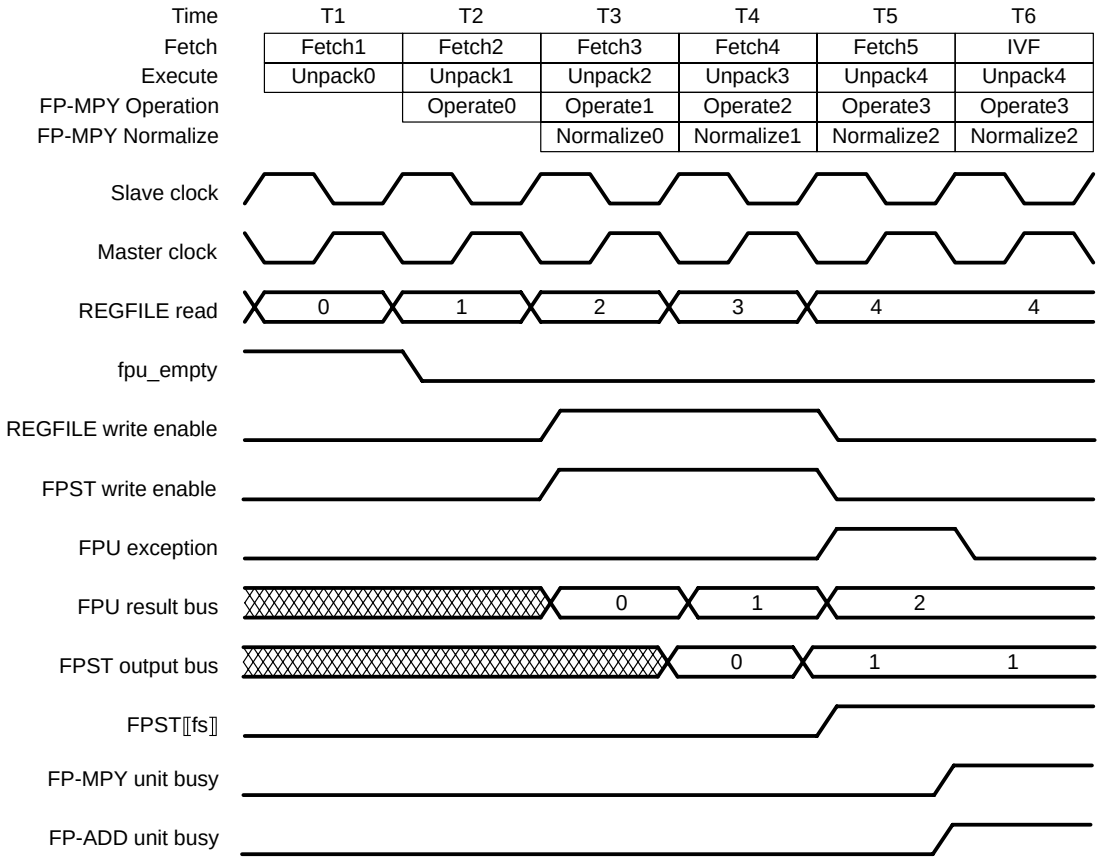
Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit

B.7 Input Exception With Interrupt Enabled

Figure B–7 shows what happens when an input exception caused by Fetch1 is detected during T2, the input exception interrupt is enabled, and the sequential mode is disabled.

- The instruction for Fetch1 is piped to the output of the floating-point unit.
- The exception is signaled on the cycle following the FPST write.
- The floating-point unit pipeline begins stalling at T5.
- The floating-point multiply operation stage may be longer than one cycle.
- Assuming one cycle for the floating-point multiply operation stage, Fetch4 is copied to the EIP register, and Fetch5 is copied to the EPC register.
- Fetch4 and Fetch5 are copied to the PC in sequence upon returning from the exception handler. (Fetch3 has already been loaded into the floating-point unit input registers and will resume execution in the appropriate operation stage when the floating-point unit is unstalled.)

Figure B–7. Input Exception With Interrupt Enabled



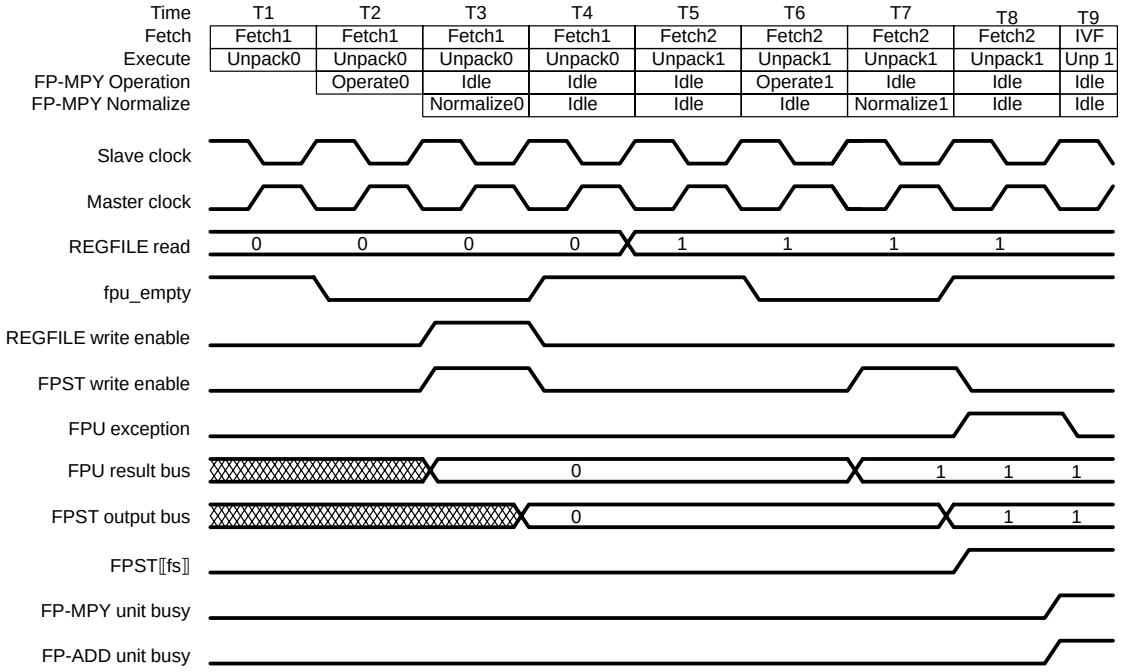
Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit
FPU = Floating-point unit
IVF = Interrupt vector transfer address fetch for the interrupt service routine

B.8 Input Exception With Interrupt Enabled (in Sequential Mode)

Figure B–8 shows the same situation as Figure B–7 shows, except that the floating-point unit is in the sequential mode. Note that although fpu_empty is a 1 at T4, the execution unit does not execute the Unpack0 at T4. The floating-point multiply operation stage may be longer than one cycle; however, assume one cycle for the floating-point multiply operation stage.

The execution unit uses a signal (the fpu_empty signal) delayed by one clock in order to determine when it should execute a floating-point instruction during the sequential mode.

Figure B–8. Input Exception With Interrupt Enabled (in Sequential Mode)



Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit
FPU = Floating-point unit
IVF = Interrupt vector transfer address fetch for the interrupt service routine

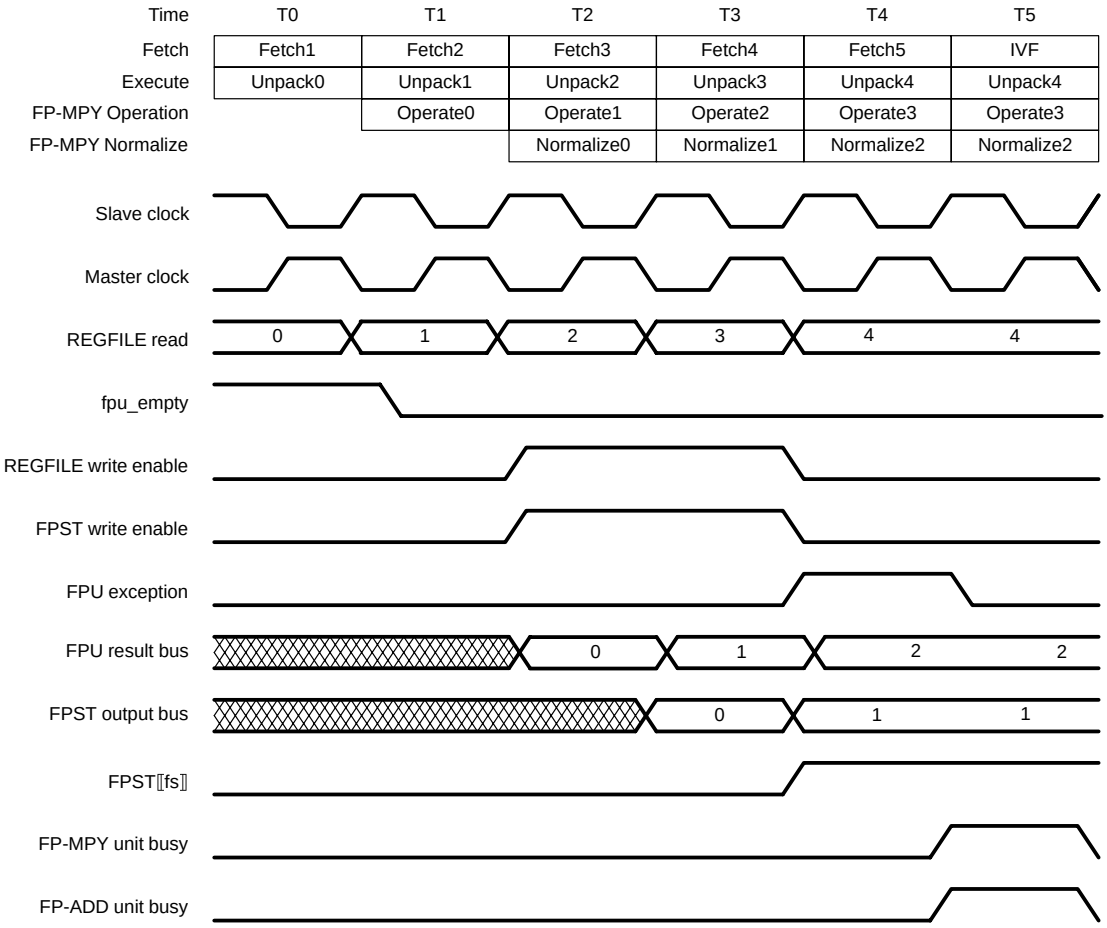
B.9 Output Exception With Interrupt Enabled

Figure B–9 shows what happens when an output exception caused by Fetch1 is detected during T3 and that output exception interrupt is enabled.

- The floating-point unit stalls on the next cycle and remains stalled until the exception handler has completed.
- The floating-point multiply operation stage may be more than one clock cycle. Assuming one clock cycle for the floating-point multiply operation stage, the execution unit will not execute the instruction from Fetch4 and will not fetch any other instructions until the exception handler has completed. Fetch4 should be in the EIP, and Fetch5 should be in the EPC.
- Fetch4 and Fetch5 are copied to the PC in sequence upon returning from the exception handler. (Fetch3 has already been loaded into the floating-point unit input registers and will resume execution in the appropriate operation stage when the floating-point unit is unstalled.)

The sequential mode does **not** affect the register file write enable when an exception is an output exception.

Figure B–9. Output Exception With Interrupt Enabled

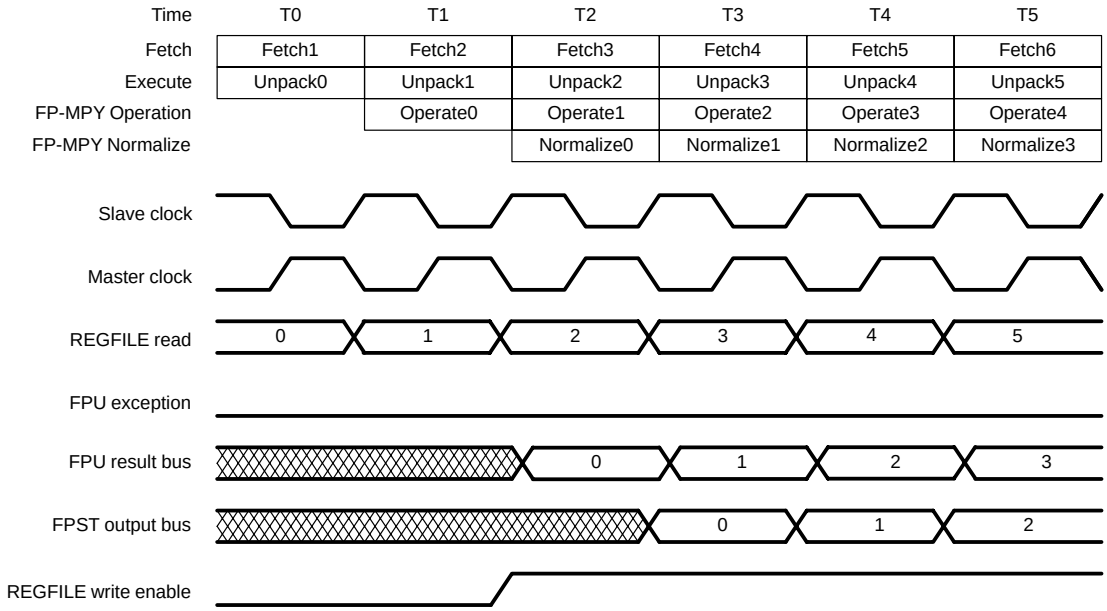


Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit
FPU = Floating-point unit
IVF = Interrupt vector transfer address fetch for the interrupt service routine

B.10 Input/Output Exception With No Interrupts Enabled

Figure B–10 shows what happens when a disabled input exception is detected during Unpack1 or a disabled output exception is detected during Normalize1. The input exception is piped through the floating-point unit and is written to the register file with the timing of a single-precision instruction. Neither the execution unit or floating-point unit pipeline is affected.

Figure B–10. Input/Output Exception With No Interrupts Enabled



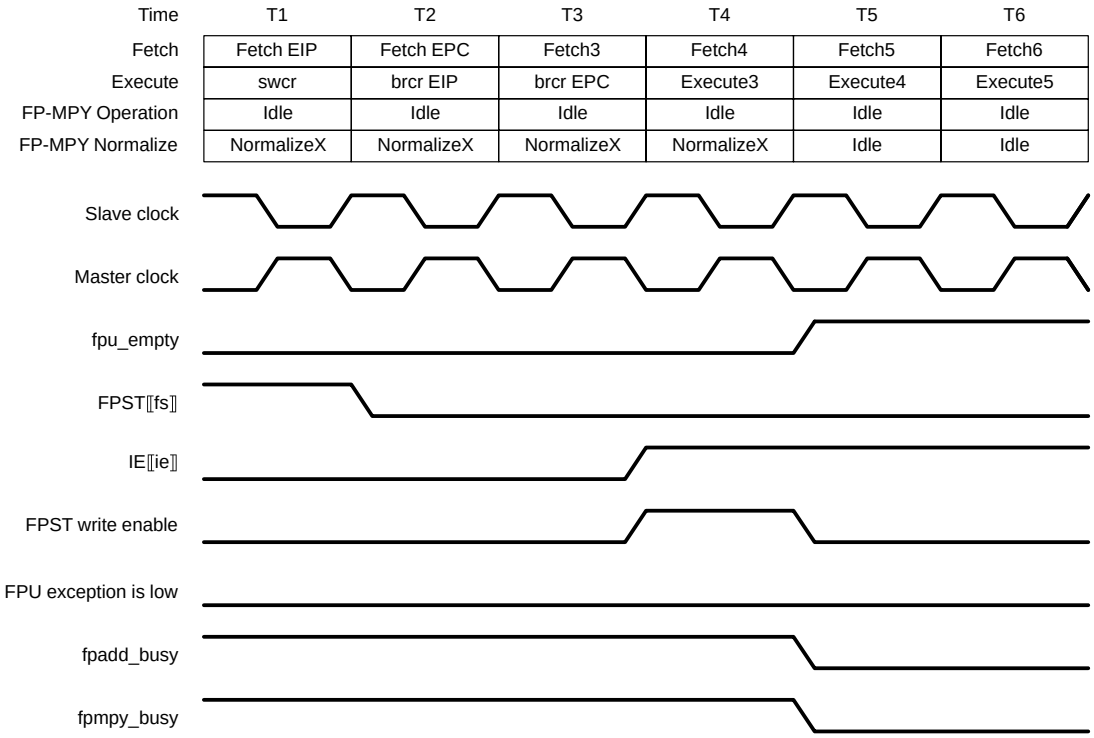
Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit
FPU = Floating-point unit

B.11 End of a Floating-Point Exception Interrupt-Handling Routine

Figure B–11 shows the signals at the end of a floating-point exception interrupt-handling routine. There may be several clock cycles between the **swcr** instruction at T1 and the **brcr EIP** instruction at T2.

- The swcr instruction at T1 is executed by the floating-point exception interrupt-handling routine to change FPST⟦fs⟧ to 0 (floating-point unit unstalled).
- The two branches (brcr EIP and brcr EPC) **must** be the last instructions of most floating-point exception interrupt-handling routines. The return value of ie from the **brcr EPC** instruction is a 1 (interrupt enabled), which signals to the floating-point unit that the floating-point exception interrupt-handling routine has completed, since fs is 0 also.
- One other instruction still in the floating-point unit pipeline causes the FPST write enable to go to a 1 at T4. This floating-point unit instruction does not cause another exception.
- Execute3 is the instruction copied to the PC by the **brcr EIP** instruction.
- Execute4 is the instruction copied to the PC by the **brcr EPC** instruction.

Figure B–11. End of a Floating-Point Exception Interrupt-Handling Routine



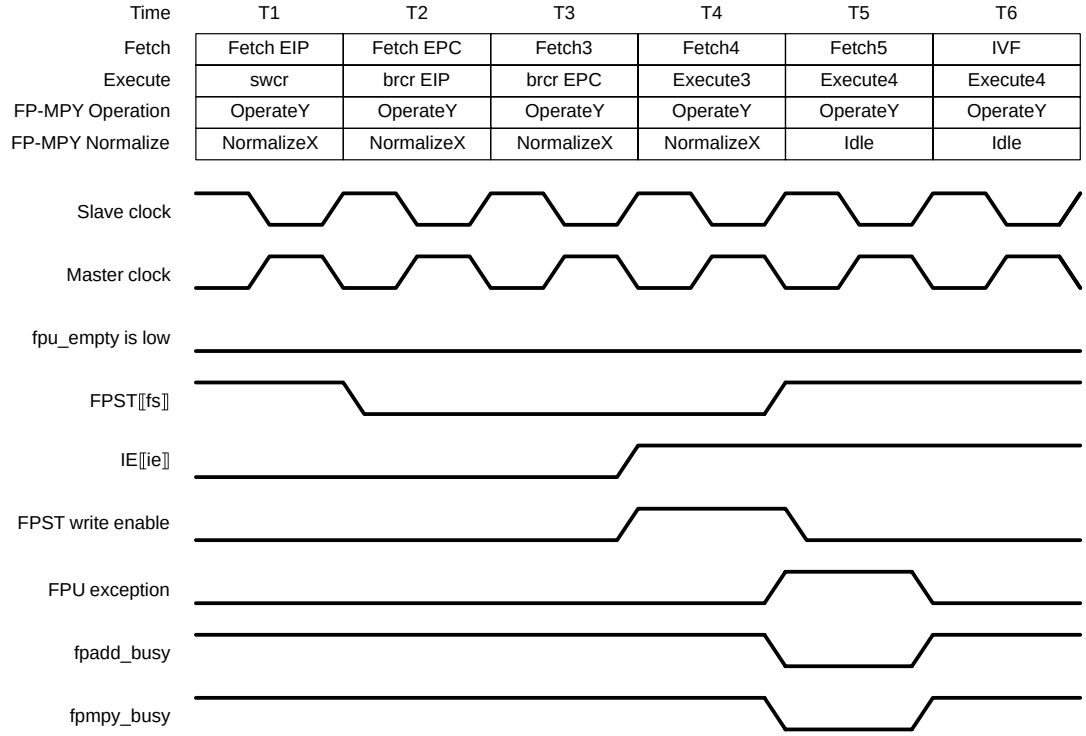
Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit
FPU = Floating-point unit

B.12 Back-to-Back Interrupt Exceptions

Figure B–12 shows the signals at the end of a floating-point exception interrupt-handling routine.

- This time there is a collision (at a time before T1) at the floating-point add and floating-point multiply normalization stages.
- The floating-point add instruction caused an exception.
- The end of the floating-point exception interrupt-handling routine for the floating-point add unit instruction is at T3 (IE⟦ie⟧ = 1).
- The floating-point multiply output from the collision is written at T4. This output causes an exception, which is signaled at T5.
- One other instruction is still in the floating-point unit pipeline in one of the operation stages. It does not matter which operation stage (floating-point add or floating-point multiply).
- Both floating-point unit operation stages are stalled during a floating-point exception interrupt-handling routine and for one clock following a floating-point exception interrupt-handling routine. This way, OperateY does not advance between two consecutive exceptions.
- If the second NormalizeX floating-point multiply instruction had not caused an exception, OperateY would have advanced to a normalization stage (assuming that it was supposed to in the normal operation of the instruction) at T5. Because of the floating-point unit exception signaled at T5, Execute4 is copied to the EIP register, and Fetch5 is copied to the EPC register. The ie bit will go to 0 at T8 (not shown).

Figure B–12. Back-to-Back Interrupt Exceptions



Note: **FP-MPY** = Floating-point multiply unit
FP-ADD = Floating-point add unit
FPU = Floating-point unit
IVF = Interrupt vector transfer address fetch for the interrupt service routine

Appendix C

Floating-Point Unit Exceptions

This appendix describes the floating-point, vector, and accumulator-destination exceptions and provides additional information about the results from these exceptions.

Topics

C.1 Input and Output Exceptions

This section discusses the two types of floating-point unit exceptions: input and output. In general, input and output exceptions are handled in the same manner. The difference between the two types is that input exceptions are detected during the first execute cycle of the floating-point instruction, and output exceptions are detected during the last execute cycle of the floating-point instruction.

This section covers the following scenarios and concludes with a discussion of floating-point unit exception priorities.

- Input exceptions with interrupts enabled
- Output exceptions with interrupts enabled
- Input exceptions—no interrupts enabled
- Output exceptions—no interrupts enabled

C.1.1 Input Exceptions With Interrupts Enabled

If the floating-point unit unpackers detect an input exception and the corresponding input-exception interrupt **is** enabled, then the exception is piped through the appropriate floating-point unit stage(s) (floating-point multiply/floating-point add) to the floating-point unit output.

For instructions other than the vmac-type instructions (vmac, vmsc, and vmsub), the correct exception result is written with the timing of a single-precision instruction. For the vmac-type instructions, the correct exception result is first piped through the floating-point multiply unit and then through the floating-point add unit before it is written to the register file or accumulator. The exception is written to the FPST register on the same cycle that the result is written to the register file or accumulator; on the next cycle, the output bus of the FPST register (including the fs bit) reflects the correct status. The fpu-exception signal will go to a 1 on the cycle following the write to the FPST register.

The floating-point unit pipe stalls on the same cycle as of the fpu-exception signal because the FPST⟦fs⟧ bit is set by the floating-point unit. Setting fs also makes the registers that are scoreboarded for the floating-point unit available to the exception handler.

Handling input exceptions with interrupts enabled in this manner does not allow the floating-point interrupt handling routines to have access to the input operands that caused the exception (as the IEEE 754-1985 standard states they should), because the input register information is not stored in FPST. Also, another instruction may write to the register containing the input exception operand before the floating-point unit signals the exception, or the result of the instruction that caused the exception may have a destination register that is the same as one of the input registers (as required by some of the vector instructions). Therefore, the sequential mode affects the way input exceptions with interrupts enabled are processed.

When the invalid or divide-by-zero interrupt is enabled, you must operate the floating-point unit in sequential mode in order to have access to input operands that cause an input exception. When the sequential mode is enabled, the execution unit stalls when the floating-point unit instruction is executed, allowing only one floating-point instruction in the floating-point unit at a time. Therefore, when the floating-point unit signals the input exception after the exception is piped to the output of the floating-point unit, and the floating-point unit interrupt is taken, the address to the instruction that caused the input exception is stored in the EIP register.

In addition, if the following conditions are true:

- the sequential mode is enabled,
- an input exception is detected, and
- the exception's corresponding interrupt is enabled,

when the exception is piped through to the output of the floating-point unit, the register-file write-enable is inhibited so that the result of the exception does not write over one of the input operands that caused the exception. This is true for all divide-by-zeros and all invalid-exceptions, except round from floating-point number to integer when overflow occurs.

Note that this is classified as an output exception. There will be no change to the FPST write enable or to the fpu-exception signal because of the sequential mode. The IEEE 754-1985 standard does not define the result that should be returned to the invalid or divide-by-zero interrupt handlers, so it is permissible not to return any result at all.

For vector instructions, parallel loads and stores are **not** aborted when input exceptions are detected. Therefore, you should be aware that a parallel load may overwrite the contents of a source register that causes an input exception, even in the sequential mode.

At the end of a floating-point exception interrupt handling routine, the exception handler must clear (set to 0) the fs bit in order to unstall the floating-point unit. However, the floating-point unit will not actually resume until interrupts are re-enabled (as indicated by IE[[ie]] = 1). This prevents the floating-point unit from resuming operations and writing to a destination register that the exception handler has saved and is using for a different purpose.

The method for processing an input exception with its interrupt enabled is as follows:

- The instruction is **not** aborted, including any parallel loads/stores, address increments, and scoreboarding. (Note that parallel loads may destroy the contents of a register that the floating-point interrupt handler needs.)
- The instruction is piped to the output of the floating-point unit. This helps guarantee that only one floating-point unit exception is processed at a time—the input exception takes up a slot in the floating-point unit pipeline.
- The latency for input exceptions is three clocks for floating-point multiply unit instructions, four clocks for floating-point add unit instructions, and six clocks for the three vmac-type instructions.
- The FPST write-enable bit goes to 1 during the last execute cycle of the instruction. The floating-point unit sets the fs (floating-point unit stall) bit to 1 while updating the FPST register. This stalls the whole floating-point unit and makes registers scoreboarded by the floating-point unit available to the exception handler.
- If **not** in the sequential mode, the register file write-enable bit goes to 1 at the same time as the FPST write enable.
- If in the sequential mode, the register file write enable is inhibited.
- The fpu exception is signaled on the cycle following the FPST write.

C.1.2 Output Exceptions With Interrupts Enabled

If an output operand exception is detected and the exception interrupt is enabled, the interrupt is taken after the proper IEEE-defined result is written to the destination register. The fpu-exception signal is sent to the execution unit, which indicates that a floating-point unit exception has occurred.

The result is written to the register file during the cycle that the exception is detected. The exception is written to the FPST register on the same cycle that the result is written to the register file; on the next cycle, the output bus of FPST (including fs) reflects the correct status. On the cycle following the write to the FPST register, the fpu-exception signal goes to 1.

The floating-point unit pipe stalls on the cycle following the rise of the fpu-exception signal because the FPST⟦fs⟧ bit is set by the floating-point unit. It remains stalled until the exception handling routine has been completed. Setting fs also makes the registers that are scoreboarded for the floating-point unit available to the exception handler. The contents of the FPST register is all you need to implement a proper IEEE interrupt handler.

Refer to subsection 8.5.2, *Floating-Point Sequential Mode Versus Pipelined Mode*, for information about accessing the source operands and the instruction that causes an output exception.

Before returning, the exception handler should clear the FPST⟦fs⟧ bit to unfreeze the floating-point unit. However, the floating-point unit will not actually resume until interrupts are re-enabled (as indicated by IE⟦ie⟧ = 1). This is to avoid the floating-point unit from resuming operations and writing to a destination register that the exception handler has saved and is using for a different purpose.

The sequential mode does **not** affect the register file write enable when an exception is an output exception.

C.1.3 Input Exceptions—No Interrupts Enabled

If an input exception is detected by the floating-point unit unpackers and its corresponding interrupt is disabled, the exception is piped through the appropriate floating-point unit stages (floating-point multiply unit/floating-point add unit) to its output.

For instructions other than the vmac-type instructions (vmac, vmsc, and vmsub), the correct exception result is written with the timing of a single-precision instruction. For vmac-type instructions, the correct exception result is also piped through the floating-point add unit before it is written to the register file or accumulator. The exception is written to FPST on the same cycle that the result is written to the register file or accumulator, and on the next cycle, register FPST reflects the correct status. The fpu-exception signal remains a 0. The floating-point unit pipeline is **not** stalled by an input exception with its interrupt disabled.

The sequential mode does **not** affect the register file write enable when an exception is an input exception with its corresponding interrupt disabled.

C.1.4 Output Exceptions—No Interrupts Enabled

If an output exception is detected and its corresponding interrupt is disabled, the correct exception result is written to the register file during the current cycle. The exception is written to the FPST register on the same cycle that the result is written to the register file, and on the next cycle, register FPST reflects the correct status. The fpu-exception signal remains a 0. The floating-point unit pipeline is **not** stalled by an output exception with its interrupt disabled.

The sequential mode does **not** affect the register file write enable when an exception is an output exception with its corresponding interrupt disabled.

C.1.5 Priority of Floating-Point Unit Exceptions

Since input exceptions are piped to the output of the floating-point unit, the priority of handling floating-point unit results is very simple. If a floating-point add result is generated on the same clock as a floating-point multiply result, then

- the floating-point add result (with a disabled/enabled input exception, a disabled/enabled output exception, or no exceptions) is written to its destination,
- if applicable, the floating-point add result is handled by a interrupt routine before the floating-point multiply result (with a disabled/enabled input exception, a disabled/enabled output exception, or no exceptions) is written to its destination, and
- if applicable, the floating-point multiply result is handled by an interrupt routine.

This is true, whether or not the floating-point add write is to the register file or to one of the four accumulators. For more information on exception handling when the destination of a floating-point add write is an accumulator, see Section C.3.

The only exceptions that can occur simultaneously are overflow and inexact or underflow and inexact. For any one instruction, if enabled, the overflow and underflow interrupts take precedence over a separate inexact interrupt.

If more than one exception is signaled (such as underflow and inexact) and their corresponding interrupts are enabled, then all of the exceptions whose interrupts are enabled will be taken in the order of their priority unless one of the interrupts clears the other interrupt(s) from the interrupt pending register (INTPEN).

C.2 Vector Operation Exceptions

Refer to Section 8.3.7, *Denormals in the Multiplier*, for an explanation of how denormal numbers as inputs to vmac-type instructions are handled.

The vmac-type instructions (vmac, vmsc, and vmsub) cannot produce an output exception in the multiply phase of their vector operation, because the input operands are always single-precision and the result operand is always double-precision. Any single-precision number can be exactly represented in the double-precision format.

It is possible to generate an input exception at the floating-point add unit during vmac-type instructions when the appropriate values of infinity are stored in the accumulator and generated by the floating-point multiply unit. If an input exception is generated at the input of the floating-point add unit during these three instructions, then the exception is piped to the output of the floating-point unit. For a description of how to avoid an invalid operation at the floating-point add during these three instructions, see subsection 10.3.6, *Constraints Associated With Using the Accumulators*.

Input exceptions to the vmac-type instructions produce a QNaN (quiet not-a-number) input to the floating-point add unit. The QNaN input to the floating-point add unit produces a QNaN output from the floating-point add unit and is written to the appropriate register or accumulator. If the input interrupt is enabled and the floating-point unit is in the sequential mode, the result is not written to the register file.

Enabled input exceptions to vector instructions are treated the same way as all other enabled input exceptions. For more details, see subsection C.1.1.

For vector instructions that use both the floating-point multiply and the floating-point add pipelines and have a single-precision destination in the register file, a double-precision, normalized number can be produced, which, when rounded to single-precision, falls in the denormal range for single-precision numbers. Given that fadd.dds and fsub.dds, which are used in the floating-point add half of vmac.sss type instructions, are not specified by the IEEE standard, the result of these instructions are forced to zero, whenever a double-precision normalized result that falls in the denormal range for single-precision numbers (**before** rounding) is produced. No rounding takes place. The inexact and underflow flags will be signaled.

The method for detecting these denormal outputs is as follows. The double-precision result of the floating-point add is first normalized. This could produce an unbiased exponent value less than –126 before rounding. Therefore, the result is forced to zero, and inexact and underflow is signaled.

C.3 Exceptions With Accumulator Destinations

For instructions that can write to an accumulator, either the destination register must be encoded as r0 and the destination precision must be double-precision, or there must be a parallel load or store. Only the floating-point add unit can write to an accumulator.

The procedures for writing results and exceptions to the register file and to the FPST register (described in earlier sections about exceptions) are followed, even though an accumulator is the destination. Moreover, even though an accumulator is the real destination of the result, the write to the register file location (encoded as r0) and to the FPST register takes place at the same time that the accumulator is written. This implies that if the floating-point add unit is writing to an accumulator and at the same time the floating-point multiply unit wants to write to the register file, the floating-point multiply unit pipeline is stalled so that the floating-point add unit can write to the register file location (encoded as r0) and to the FPST register. Any enabled interrupts due to input or output exceptions are taken at this time.

Any exception generated in any stage of an instruction is written to the FPST register when the result of the entire instruction is written to an accumulator. The value in an accumulator can be used only as the source2 operand to the floating-point add unit.

The accumulative flags in the FPST register indicate whether a group of instructions have set any status bits. However, since the recommended method for using vmac-type instructions has two or more unrelated strings of vmac-type instructions intermixed (as shown if Example C–1), the accumulative flags in the FPST register indicate the accumulative status of both unrelated strings of vmac instructions. You must separate the two unrelated strings of vmac instructions to look at the accumulative flags for each string of instructions individually.

Example C–1. vmac's Accumulated Status

```
vmac.ssd r2,r3, 0,a0 ; (r2 × r3) + 0 --> a0
vmac.ssd r4,r5, 0,a1 ; (r4 × r5) + 0 --> a1
vmac.ssd r2,r3,a0,a0 ; (r2 × r3) + a0 --> a0
vmac.ssd r4,r5,a1,a1 ; (r4 × r5) + a1 --> a1
vmsc.ssd r2,r3,a0,a0 ; a0 – (r2 × r3) --> a0
vmac.ssd r4,r5,a1,a1 ; (r4 × r5) + a1 --> a1
vmac.ssd r2,r3,a0,r6 ; (r2 × r3) + a0 --> r6
vmsub.sd r4,a1,r8    ; (r4 × 1.0) – a1 --> r8
```

C.4 Freezing the Floating-Point Unit Pipeline

The floating-point unit stalls on a floating-point unit exception by using the FPST⟦fs⟧ bit when that exception's interrupt is enabled. This implies that the exception interrupt handling routine cannot use floating-point unit instructions, and, in addition, it must not enable any interrupts whose service routines use floating-point unit instructions.

The alternative is to store the intermediate results in the floating-point unit pipeline so that the exception handling routine can use floating-point unit instructions. This requires several kilobits of information to be stored from the floating-point unit in order to save the current state. This added hardware complexity is not implemented.

While the fs bit is set or the ie bit is 0, any registers marked **in-use** by the floating-point unit in the scoreboard are temporarily made free. This is done by having two bits per register in the scoreboard—one for memory loads and one for the floating-point unit. The effective scoreboard bit for a register is:

SB_{MEM} OR ($SB_{\text{floating-point unit}}$ AND $\overline{fs}$ AND ie)

For more information about scoreboarding, refer to subsection 2.2.3, *Register Scoreboarding*.

C.5 Results From Floating-Point Unit Exceptions

For any exception whose corresponding interrupt is enabled, the IEEE specification states that the result of the operation is undefined, unless it is returned by the interrupt handler.

C.5.1 Invalid Exception

If any of the following events occur, the floating-point invalid exception is signaled.

1) Any operation on a signaling NaN (not-a-number).
2) Magnitude subtraction of infinities: $(+\infty) - (+\infty)$, or $(+\infty) + (-\infty)$, or $(-\infty) - (-\infty)$.
3) Zero times infinity: $\pm 0 \times \pm \infty$.
4) Zero divided by zero: $\pm 0 \div \pm 0$.
5) Infinity divided by infinity: $\pm \infty \div \pm \infty$.
6) Square root of any negative number other than negative zero.
7) Conversion of any negative number (other than negative zero) to an unsigned integer.
8) Conversion of a binary floating-point number to integer if the input operand is an infinity or a NaN.
9) Conversion of a binary floating-point number to integer if an overflow occurs.

For invalid operations 1–8, when the invalid interrupt is disabled, the floating-point unit returns the QNaN (all 1s) as the result to the destination. Refer to subsection C.5.6 for the results of invalid operation 9 when the invalid interrupt is disabled.

For invalid operations when the invalid interrupt is enabled, the IEEE specification does not specify what result should be returned from floating-point unit to the interrupt handler.

For invalid operations 1–8, when the invalid interrupt is enabled:

- If the destination is an accumulator, then the QNaN is written to the accumulator.
- If the destination is the register file, then the QNaN is written to the register file only if the floating-point unit is **not** in the sequential mode (for example, FPST⟦sm⟧ = 0). No result is returned from the floating-point unit to the interrupt handler if the floating-point unit is in the sequential mode and the destination is the register file. This prevents the result from destroying the contents of an input register, which may be needed by the interrupt handler. The interrupt handler can access the input operands of the exception instruction via the EIP register because the sequential mode ensures that the EIP contains the address of the instruction that caused the interrupt.

Refer to subsection C.5.6 for the results of invalid operation 9 when the invalid interrupt is enabled.

C.5.2 Divide-by-Zero Exception

During a floating-point division, if the divisor is zero and the dividend is a finite nonzero number, then the floating-point divide-by-zero exception is signaled. Whether or not the divide-by-zero interrupt is enabled or disabled, the floating-point unit returns a correctly signed infinity, unless the floating-point unit is operating in the sequential mode. The floating-point unit does not return a result to the interrupt handler if the floating-point unit is in the sequential mode.

C.5.3 Inexact Exception

When a rounded result of an operation is not exact or if overflow occurs without the overflow interrupt enabled, then the inexact exception is signaled. The floating-point unit returns the same result, whether or not the inexact interrupt is enabled or disabled.

C.5.4 Overflow Exception

A square root operation cannot overflow. When the overflow interrupt is disabled and an overflow occurs, the floating-point unit returns the result specified by IEEE. Except for conversions from double precision to single precision, when the overflow interrupt is enabled and overflow occurs, the floating-point unit returns to the interrupt handler a result that has the correctly rounded and normalized mantissa with a wrapped exponent that follows the format shown in Section A.6, *Wrapped Numbers*.

The IEEE standard states that for an interrupt-overflow during conversion from a binary floating-point format, the floating-point unit shall deliver to the interrupt handler a result in the input format or in a wider format; in addition, the exponent bias may be adjusted but must be rounded to the destination's precision.

For the MP's floating-point unit, this standard affects double-precision to single-precision conversions. Since the floating-point unit must return a result in the input format or in a wider format rounded to the destination's precision (single precision), the result required from the floating-point unit is 35 bits wide:

- A 23-bit mantissa (not including the hidden bit)
- An 11-bit exponent
- A 1-bit sign

However, the destination register is only 32 bits wide. For interrupt-overflow during conversions from double precision to single precision, the floating-point unit returns the following to the 32-bit single-precision destination register:

- The correctly rounded and normalized mantissa is written to the 23 LSBs of the destination register (the normal mantissa location for single-precision numbers shown in Figure A–1, *Single-Precision Format Example—Packed Notation*). Remember that the hidden bit (the MSB of the mantissa) of the result is not stored in register format.

- The 8 LSBs (7–0) of the 11-bit input exponent are stored in bits 30–23 of the destination register.

- Bit 8 of the 11-bit input exponent is written to the FPST⟦e0⟧ bit.

- Bit 9 of the 11-bit input exponent is written to the FPST⟦e1⟧ bit.

- Bit 10 of the 11-bit input exponent is **not** stored. Since overflow has occurred, bit number 10 **must** be a 1.
- The correct sign is written to the MSB of the single-precision destination register. This is the normal location of sign for single-precision results, as specified in Figure A–1, *Single-Precision Format Example—Packed Notation*.

If overflow occurs when floating-point numbers are converted to signed or unsigned integers, an invalid interrupt is signaled, not an overflow.

C.5.5 Underflow Exception

A square root operation cannot underflow. Additionally, underflow cannot occur when floating-point numbers are converted to signed or unsigned integers; an inexact zero is produced for this situation. When the underflow interrupt is disabled and underflow occurs, the floating-point unit returns the result specified by IEEE.

- For multiplication and division, when the underflow interrupt is enabled and underflow occurs, the floating-point unit returns to the interrupt handler a result that has the correctly rounded and normalized mantissa with a wrapped exponent, which follows the format shown is Section A.6, *Wrapped Numbers*.

- For addition and subtraction, when the underflow interrupt is enabled and underflow occurs, the floating-point unit returns a denormalized result because the only tiny numbers that can be produced are exact denormal numbers. Therefore, the only value for the exponent is a biased 1, which is stored in the register file as all 0s.

The IEEE standard states that for interrupt-underflow during conversion from a binary floating-point format, the floating-point unit shall deliver to the interrupt handler a result in the input format or in a wider format; in addition, the exponent bias may be adjusted but must be rounded to the destination's precision.

For the MP's floating-point unit, this standard affects double-precision to single-precision conversions. Since the floating-point unit must return a result in the input format or in a wider format rounded to the destinations' precision (single precision), the result required from the floating-point unit is 35 bits wide:

- A 23-bit mantissa (not including the hidden bit)
- An 11-bit exponent
- A 1-bit sign

However, the destination register is only 32 bits wide. For interrupt-underflow during conversions from double precision to single precision, the floating-point unit returns to the 32-bit single-precision destination register:

- The correctly rounded and normalized mantissa is written to the 23 LSBs of the destination register(the normal mantissa location for single-precision numbers shown in Figure A–1, *Single-Precision Format Example—Packed Notation*). Remember that the hidden bit (the MSB of the mantissa) of the result is not stored in register format.
- The 8 LSBs (7–0) of the 11-bit input exponent are stored in bits 30–23 of the destination register.
- Bit 8 of the 11-bit input exponent is written to the FPST⟦e0⟧ bit.
- Bit 9 of the 11-bit input exponent is written to the FPST⟦e1⟧ bit.
- Bit 10 of the 11-bit input exponent will **not** be stored. Since underflow has occurred, bit 10 **must** be a 0.
- The correct sign is written to the MSB of the single-precision destination register. This is the normal location of sign for single-precision results, as specified in Figure A–1, *Single-Precision Format Example—Packed Notation*.

C.5.6 Integer Conversion Errors

The IEEE standard does not specify what result should be returned during conversion of a floating-point number to an integer when invalid occurs. It states that a QNaN (quiet not-a-number) should be returned during invalids if the destination is a floating-point format, but in this case, the destination is an integer. For this invalid operation, the floating-point unit returns the same result, whether or not the invalid interrupt is enabled or disabled.

If an invalid operation occurs during a conversion of a floating-point number to a signed integer, the floating-point unit returns the largest positive number (0x7FFF FFFF) if the source is positive, SNaN (signaling not-a-number), or QNaN; or the largest negative number (0x8000 0000) if the source is negative.

If an invalid operation occurs during a conversion of a floating-point number to an unsigned integer, the floating-point unit returns the largest number (0xFFFF FFFF) if the source is positive, SNaN, or QNaN; or the smallest number (0x0000 0000) if the source is negative. Conversions of negative numbers to unsigned integers is invalid, as stated in Section 8.4, *Floating-Point Unit Exceptions*.

During conversion from floating-point to integer, before an invalid due to overflow can be signaled, the conversion must take place in order to determine the overflow. Therefore, this becomes an output exception similar to overflow, underflow, and inexact. In order for the invalid interrupt handler to determine the input operand that caused the invalid, the sequential mode must be enabled. Since this is an output exception, the sequential mode does **not** affect the register file write for this exception (FPST⟦sm⟧ = 1). Therefore, when the invalid interrupt is enabled, you should not write the result to the input register, because the interrupt handler may need the input operand.

C.5.7 Floating-Point Multiply Overflow of Integers

When an overflow occurs during integer multiply, the 32 LSBs of the result of that multiply are output from the floating-point unit onto the result bus. The multiply integer overflow flag is set, and the accumulative invalid is **not** set in the FPST register.

There are no floating-point exceptions signaled in the INTPEN register as a result of a floating-point integer multiply overflow.

Appendix D

Examples of MP Packet Transfers

This appendix offers examples of packet transfers. The types of packet transfers illustrated fall into these categories:

- Guided transfers
- Dimensioned transfers

For more detailed information on packet transfers, see Chapter NO TAG, *Packet Transfers*, in the *MVP Transfer Controller User's Guide*.

Topics

D.1 Guided Transfer Examples

In guided transfers (source and/or destination), the address sequence is guided by an on-chip table, rather than calculated from the packet transfer parameters. The base address is contained in the packet transfer parameters. Subsequent addresses are generated from either an offset or delta table (located in the on-chip RAMs), depending on the type of guided transfer indicated by the packet transfer options.

There are two classes of guided transfers:

- In **fixed-patch guided transfers**, the sizes of 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. Subsection D.1.1, *Drawing a Line Using a Fill-With-Value to Fixed-Patch Delta-Guided Packet*, illustrates how you can use a fill-with-value to fixed-patch delta-guided packet to draw a line.
- **Variable-patch guided transfers** are similar to fixed-patch transfers, except that the sizes of the first and second dimensions are specified in the guide table instead of in the packet transfer. Variable-patch guided transfers are effective for functions such as a trapezoidal fill. Subsection D.1.2, *Drawing a Line Using a Delta-Guided to Dimensioned Packet*, illustrates how you can use a delta-guided to dimensioned packet to draw a line.

For more information about guided transfers, see Section NO TAG, *Guided Transfers*, in the *MVP Transfer Controller User's Guide*.

D.1.1 Drawing a Line Using a Fill-With-Value to Fixed-Patch Delta-Guided Packet

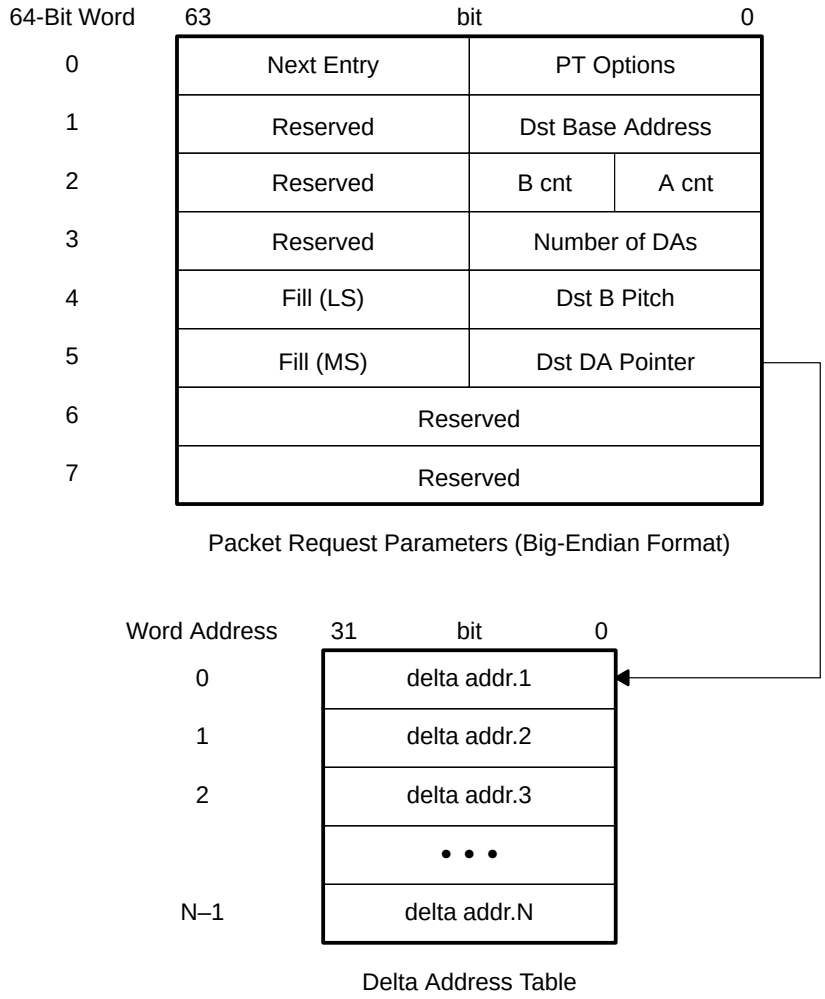
To draw a line with a fixed color in an image, the TC can write to the appropriate pixels in external memory without ever bringing the pixels into the MVP. This is done with a special packet feature referred to as fill-with-value.

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
  - ■ Packet transfer options (fill-with-value to fixed-patch delta-guided)
  - ■ Starting address for the line
  - ■ Fill value
  - ■ B cnt = 0
  - ■ A cnt = number of bytes per pixel
- ❑ Poll until the processor does not have a queued packet transfer request.
- ❑ Issue the packet transfer request to the TC (once the TC has no queued packet transfer for this processor).

The fill-with-value to fixed-patch delta-guided packet shown in Figure D–1 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 form to avoid the delta address calculations.

Figure D–1. Fill-With-Value to Fixed-Patched Delta-Guided Packet Request



The packet in Figure D–1 writes the fill value to pixel start addresses as shown:

```
Address(pixel 1) = delta address 1 + dst 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)
```

The fill value that overwrites the old pixel value must be replicated to use all 64 bits in the packet parameter words. As a result, if the pixel is a byte, halfword, or word, the fill value would be repeated eight, four, or two times, respectively.

D.1.2 Drawing a Line Using a Delta-Guided to Dimensioned Packet

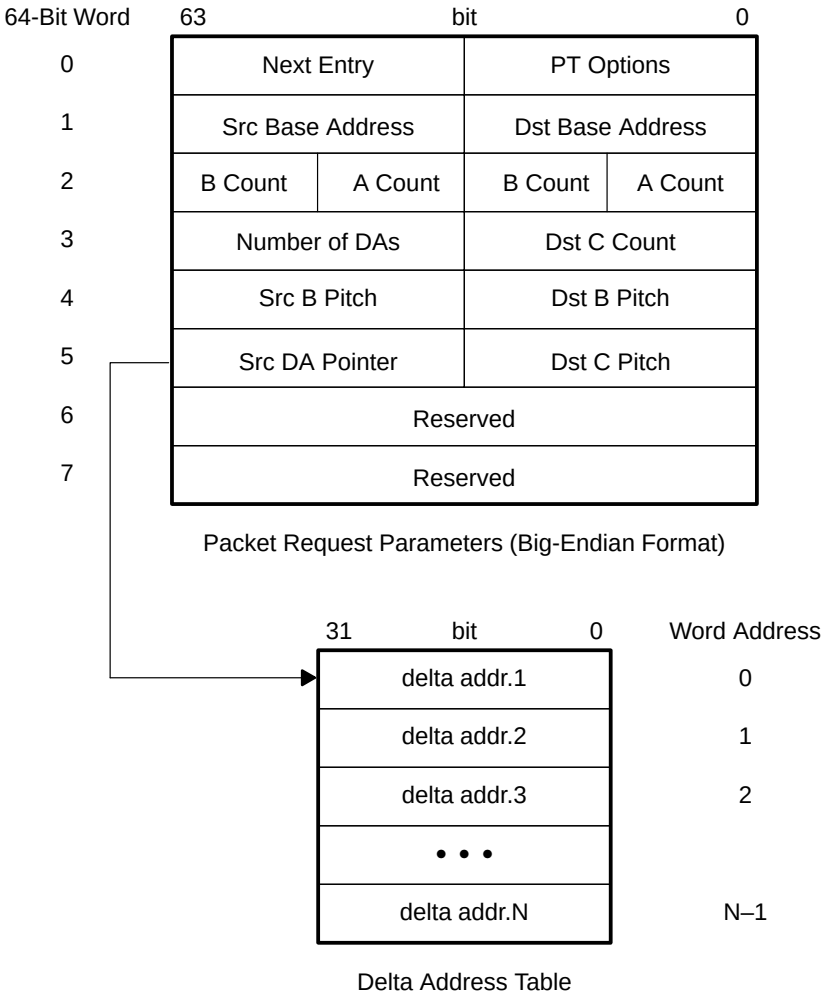
The example shown in this subsection a fill-with-value to fixed-patch delta-guided transfer, using the fill value as the source and the fixed-patch delta-guided destination; this example should be considered a special case. Typically, a line drawing must consider aspects such as z-buffering and/or plane-masking, and you may want to draw patterned instead of fixed-color lines.

A more general line draw uses the fixed-patch delta-guided source and the dimensioned array in on-chip RAMs as destination (see Figure D–2) to bring the pixels linearly into the on-chip RAMs. Operations are performed on the pixels in the line. The processed line is then transferred back to external memory.

A general line-draw sequence is illustrated below:

- ❑ Determine the source input delta-guided address values.
- ❑ Store packet transfer parameters for fixed-patch delta-guided to dimensioned.
- ❑ Issue input packet transfer request to TC.
- ❑ Operate on the pixel data after it becomes available in the on-chip RAMs.
- ❑ Toggle bit 19 in packet transfer options to reverse source and destination parameters.
- ❑ Issue output packet transfer request to TC.

Figure D–2.Fixed-Patched Delta-Guided to Dimensioned Packet Request



D.2 Dimensioned Transfer Examples

Dimensioned transfers are well-suited for imaging algorithms. Image processing algorithms often process an entire image by rows, columns, or N × M blocks. Dimensioned transfers describe source and/or destinations that may be a simple contiguous linear sequence of data bytes, or up to three dimensions of such regions:

- The distance between data items along the first dimension is always one byte.
- The distance between entries in the second dimension is defined by the B pitch packet transfer parameter.
- The distance between entries in the third dimension is defined by the C pitch packet transfer parameter.

This section provides two examples of dimensioned transfers:

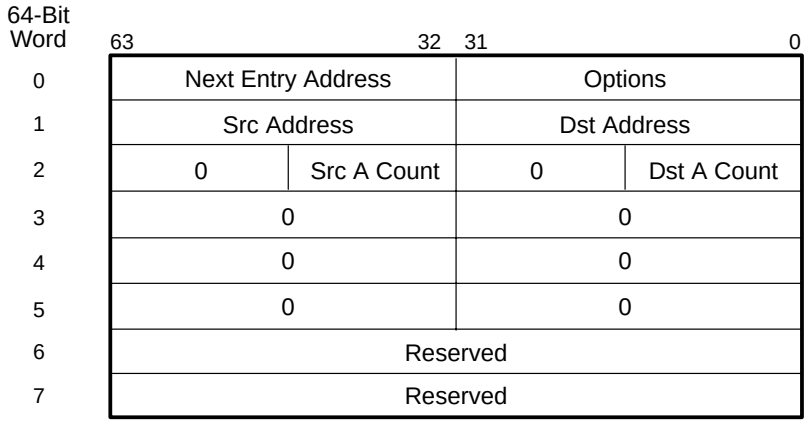
- The first example is a simple block transfer of data from external memory to on-chip RAM.
- The second example illustrates how to set up a 3-D packet transfer request for transferring multiple 8 × 8 blocks of data from the on-chip RAMs to external memory.

For more information about dimensioned transfers, see Section NO TAG, *Dimensioned Transfers*, in the *MVP Transfer Controller User's Guide*.

D.2.1 Transferring a Block of Data

This example illustrates the parameter list required to block-transfer 512 bytes starting at external memory location 0x0204 0000 to on-chip RAM location 0x3000 (the PP3's data RAM 0). The big-endian parameter list is shown in Figure D–3.

Figure D–3. Block Transfer Parameters—Big Endian



- The Next Entry Address = 0x0101 0400 and points back to this packet transfer.
- The Options field = 0x8000 0000 and represents a dimensioned-to-dimensioned transfer, MSB = 1, as last packet transfer.
- The Src Address = 0x0204 0000 and represents the first location in external memory.
- The Dst Address = 0x3000 and points to the first location in PP3's data RAM 0.
- Both Src A Count and Dst A Count = 512, which represents the number of bytes in the block transfer.

The sample set-up code for the block transfer is shown in Example D–1.

Example D–1. MP Block Transfer Set-Up Code (Big-Endian Mode)

```
; Sample code to generate block transfer packet transfer parameter
; set–up:

       or     0x01010400,r0,r5  ; Packet transfer linked-list address –>
                                ; r5
       wrcr   OUTP,r5           ; r5 –> OUTP
       or     0x80000000,r0,r4  ; last packet transfer request, dim–dim
                                ;  transfer options –> r4
       vst.d  r4                ; 64-bit r4 –> Memory(OUTP++)
       or     0x02040000,r0,r7  ; Src start address –> r7
       or     0x3000,r0,r6      ; Dst start address –> r6
       vst.d  r6                ; 64-bit r6 –> Memory(OUTP++)
       or     512,r0,r9         ; Source counts B = 0, A = 512 –> r9
       or     512,r0,r8         ; Dst counts B = 0, A = 512 –> r8
       vst.d  r8                ; 64-bit r8 –> Memory(OUTP++)
       or     0,r0,r11          ; Source C count = 0 –> r11
       or     0,r0,r10          ; Destination C count = 0 –> r10
       vst.d  r10               ; 64-bit r10 –> Memory (OUT++)
       vst.d  r10               ; Src B pitch = 0, Dst B pitch = 0
       vst.d  r10               ; Src C pitch = 0, Dst C pitch = 0

; Test PKTREQ bit Q for any queued packet transfer requests:

       rdcr   PKTREQ,r4         ; PKTREQ –> r4
Poll:  bbo    Poll,r4,1         ; is Q (bit 1) = 1 ?
       rdcr   PKTREQ,r4         ; PKTREQ –> r4 in delay slot

; Install linked–list address:

       or     0x01010400,r0,r5  ; address of packet transfer parameters
       st     0x010100FC(r0),r5 ; r5 –> 0x010100FC

; Set P bit in PKTREQ to issue packet transfer request to TC:

       rdcr   PKTREQ,r4         ; PKTREQ –> r4
       or     1,r4,r4           ; set P (bit 0) to a 1
       wrcr   PKTREQ,r4         ; r4 –> PKTREQ issues packet transfer
                                ;  request to TC at the same priority as
                                ;  the last time.

; MP must be in the supervisor mode to store into the MP's parameter RAM
; or to write control registers <0x4000.
```

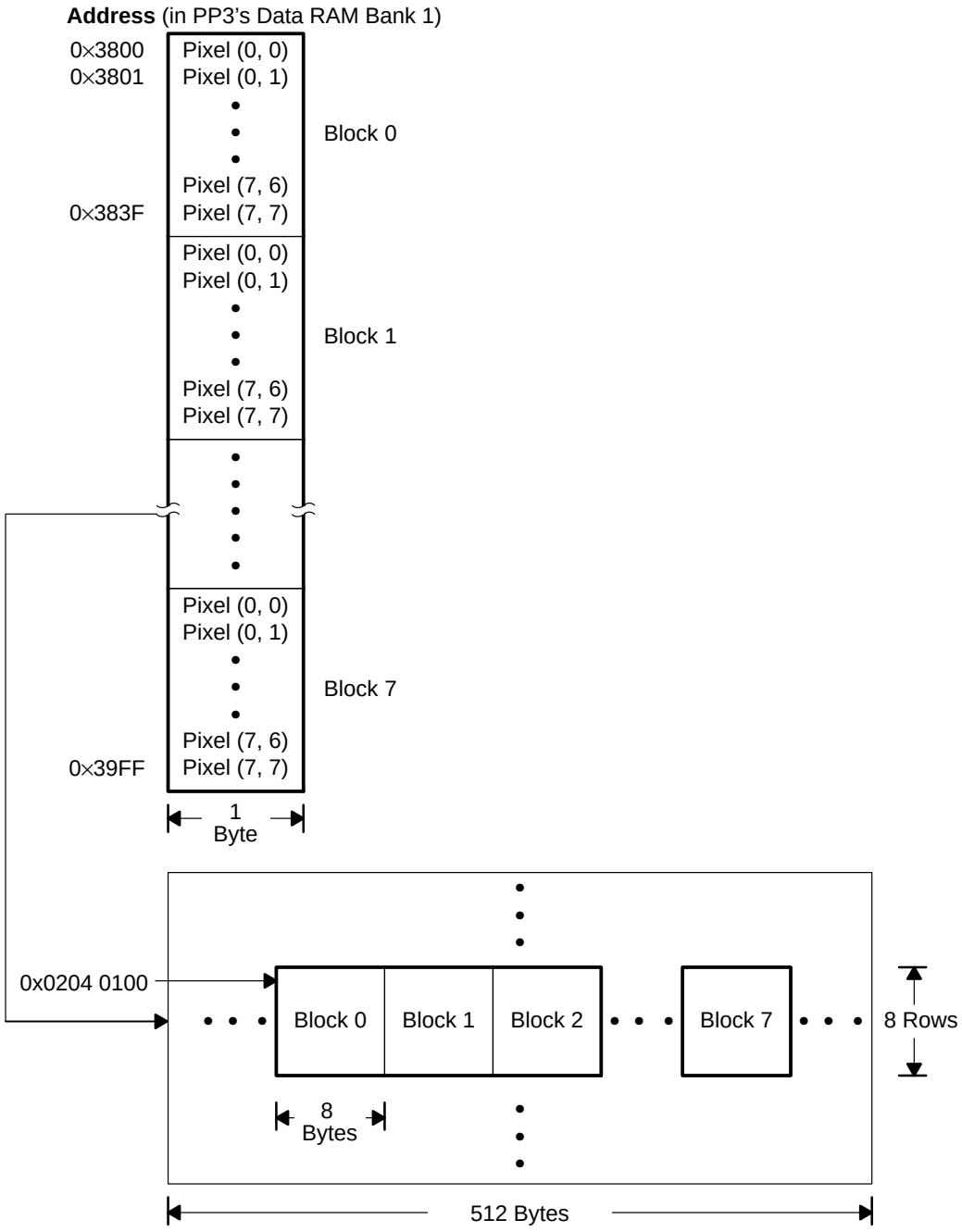
Notes: 1) You may want to take advantage of structures and use them when generating packet transfer requests. For more information on structures, see Chapter NO TAG, *Assembler Directives*, in the *MVP Code Generation Tools User's Guide.*

2) When creating your parameter list, you can make your software independent of the big-/little-endian mode by packing these values into a 32-bit word and using a store word instruction. Also, pack B cnt and A cnt into a 32-bit word as the most significant halfword and least significant halfword, respectively.

D.2.2 Transferring Multiple 8x8 Blocks of Data

This example demonstrates how to set up a dimensioned-to-dimensioned packet request to transfer eight 8×8 blocks of byte-sized pixels that have been processed by a PP back out to external memory. The 8×8 blocks of data are stored linearly in the on-chip RAMs but must be output to external memory with the associated pitches of the image, as shown in Figure D–4. Although for this example the 8×8 blocks are contiguous in external memory, this is not a requirement.

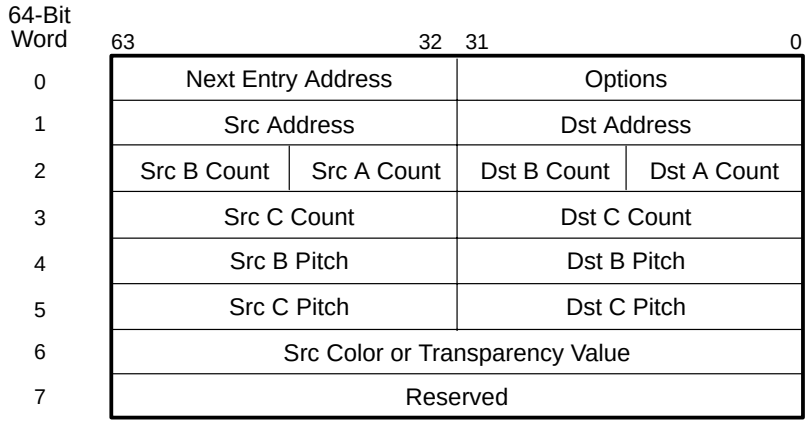
Figure D–4. Transfer From 1-D to 2-D Space



D.2.2.1 Setting Up the Packet Request Parameters

The parameter template for a dimensioned source to dimensioned destination packet transfer is shown in Figure 7–1, *Packet Transfer Parameters—Big Endian*. 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 Chapter NO TAG, *Packet Transfers*, in the *MVP Transfer Controller User's Guide*.

Figure D–5. Transfer From 1-D to 2-D Space Packet Request Parameters—Big Endian



In order to transfer the eight contiguous 1-D 64-byte sequences in the on-chip RAM to their corresponding 2-D 8 × 8 block in external memory, the packet transfer parameters for the template in Figure D–5 are set up as follows (Section NO TAG, *Packet Transfer Parameter Fields*, in the *MVP Transfer Controller User's Guide* describes each of these parameters):

- Next Entry Address = 0x0101 0400. The content of the next entry address field is written to the linked-list pointer upon completion of the packet transfer. Since for this example the same packet transfer is performed repeatedly, the next entry address should point back to the same packet transfer parameters.

- Options = 0x8000 0000. 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 packet transfer options can be used to select special access modes and automatic updates to the source and/or destination start address. Also, MSB = 1 signals that this is the last packet transfer in this linked list.

- Src Start Address = 0x3800. The source start address points to the source location in the on-chip RAMs of the first pixel in block0. For this example, this is the base address for PP3's data RAM 1 (for example, 0x3800). Note that in real application code, the source start address might be passed as an argument to the packet transfer set-up routine or loaded from memory.

- Dst Start Address = 0x0204 0100. The destination start address points to the destination location in external memory of the first pixel in block0. For this example, this address is taken to be 0x0204 0100. In real application code, the destination location is typically passed as an argument to the packet transfer set-up routine or loaded from memory.

- Src A Count = 512. The 16-bit source A count is 512 because eight blocks of 64 pixels (a pixel is one byte) must be transferred.

- Src B Count = 0. This indicates that the 16-bit second dimension is disabled for the source.

- Dst A Count = 8. For external memory, the 16-bit destination A count is 8.

- Dst B Count = 7. Note that the 16-bit B count is the number of steps that should occur within the second dimension. For this example, the destination B count corresponds to the number of rows in a block, minus one.

- Src C Count = 0. This indicates that the 32-bit third dimension is disabled for the source.

- Dst C Count = 7. This specifies the number of steps that should occur in the 32-bit third dimension of the destination. For this example, this equals the number of blocks, minus one.

- Src B Pitch = 0. Supplies as 32-bit 0 (don't care) because the source second dimension isn't active.

- Dst B Pitch = 512. This 32-bit field specifies the pitch of the second dimension of the destination. In this example, this 512 corresponds to the address delta between pixels in the same column of adjacent rows.

- Src C Pitch = 0. Supplied as 0 (don't care) because the 32-bit source third dimension isn't active.

- Dst C Pitch = 8. This 32-bit field specifies the pitch of the second dimension of the destination. For this example, the C pitch is the external memory address delta between the upper left corner pixels in adjacent 8×8 blocks.
- Src Color/Transparency = 0. This is supplied as 0 but is not used for this PT option.

Example D–2 shows MP code that sets up the packet transfer parameters described above.

Example D–3 shows the same set-up code using a packet transfer request big-endian template that is copied into the packet transfer parameter list in on-chip SRAM.

Example D–2. MP Packet Request Parameters 1-D to 2-D Set-Up Code (Big-Endian Mode)

```
; Sample code to generate 1–D to 2–D packet transfer parameter set–up:

or    0x01010400,r0,r5 ; Packet Request linked–list address –> r5
wrcr  OUTP,r5          ; r5 ––> OUTP
or    0x80000000,r0,r4 ; last packet transfer request, dim–dim transfer
                       ;  options –> r4
vst.d r4               ; 64-bit r4 –> Memory(OUTP++)
or    0x3800,r0,r7     ; Src start address –> r7
or    0x02040100,r0,r6 ; Dst start address –> r6
vst.d r6               ; 64-bit r6 –> Memory(OUTP++)
or    512,r0,r9        ; Source counts B = 0, A = 512 –> r9
or    0x70008,r0,r8    ; Dst counts B = 7, A = 8 –> r8
vst.d r8               ; 64-bit r8 –> Memory (OUTP++)
or    r0,r0,r11        ; Source C count = 0 –> r11
or    7,r0,r10         ; Destination C count = 7 –> r10
vst.d r10              ; 64-bit r10 –> Memory (OUT++)
or    512,r0,r10       ; Destination B pitch = 512 bytes
vst.d r10              ; Src B pitch = 0, Dst B pitch = 512
or    8,r0,r10         ; Destination C pitch = 8 –> r10
vst.d r10              ; Src C pitch = 0, Dst C pitch = 8
or    r0,r0,r10        ; 0 –> r10
vst.d r10              ; 64-bit Color/Transparency as 0
vst.d r10              ; reserved doubleword as 0

; MP must be in the supervisor mode to store into the MP’s parameter RAM.
```

Example D–3. MP Packet Request Parameters Set-Up Code Using 1-D to 2-D Templates (Big-Endian Mode)

```
; Copy 1–D to 2–D template from external memory to MP parameter RAM:

        or     PRTemp,r0,r4     ; Packet transfer template address
        wrcr   IN1P,r4          ; r4 –> IN1P
        or     0x01010400,r0,r5 ; Packet transfer address in MP parameter
                                ; RAM
        wrcr   OUTP,r5          ; r5 –> OUTP
        or     6,r0,r6          ; loop count
L:      vld1.d r4               ; Memory(IN1P++) –> 64-bit r4
        vst.d  r4               ; 64-bit r4 –> Memory(OUTP++)
        bcnd   L,r6,gt0.w       ; loop if r6 > 0
        addu   –1,r6,r6         ; decrement loop

        .align 8                ; align to a 64-bit boundary

PRTemp: .word  0x01010400       ; Next Entry address (MP Parameter RAM)
        .word  0x80000000       ; stop, dim–dim options
        .word  0x3800           ; Src start address (PP3’s Data RAM 1)
        .word  0x02040100       ; Dst start address (external memory)
        .word  512              ; src counts B = 0, A = 512
        .word  0x70008          ; dst counts B = 7, A = 8
        .word  0                ; src C count = 0
        .word  7                ; dst C count = 7
        .word  0                ; src B pitch = 0
        .word  512              ; dst B pitch = 512
        .word  0                ; src C pitch = 0
        .word  8                ; dst C pitch = 8
        .word  0                ; color/transparency 64 bits as 0
        .word  0
```

Note: The MP must be in the supervisor mode to store into the MP’s parameter RAM.

Appendix E

C Compiler Registers

This appendix describes the register allocation for the MP compiler (mpcl).

Topics

E.1 MP C Compiler Registers

The MP C compiler (mpcl) and associated assembly language subroutines use the register conventions shown in Table E–1.

Note:

One and only one register convention **must** be used in a linked program and executive.

Table E–1.C Compiler Register Conventions

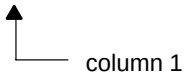
	32-Bit	64-Bit
argument 1	r2	r2–r3
argument 2	r4	r4–r5
argument 3	r6	r6–r7
argument 4	r8	r8–r9
argument 5	r10	r10–r11
argument 6	r12	r12–r13
optional answer	r2	r2–r3
working registers (not saved/restored by subroutine)	r2–r19	
working registers (saved/restored by subroutine)	r20–r30 (r31 is link)	
stack pointer	r1	
frame pointer	N/A	
link register	r31	

E.2 Using Aliases for Register Names

You can name the registers and then use an alias to quickly change from one convention to another, as shown in Example E–1.

Example E–1. Register Aliases Using the .set Directive

```
sp:    .set   r1 ; stack pointer
link:  .set   r31; link register
ans:   .set   r2 ; returned answer
arg1:  .set   r2 ; input argument 1
arg2:  .set   r4 ; input argument 2
arg3:  .set   r6 ; input argument 3
arg4:  .set   r8 ; input argument 4
arg5:  .set   r10; input argument 5
arg6:  .set   r12; input argument 6
```



Appendix F

Glossary

A

access stage: The optional third stage of the MP's FEA pipeline, during which memory accesses (load or store operations) occur. See also *FEA pipeline sequence*

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.

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*

bit detection: The special logic that supports leftmost-one, rightmost-one, leftmost-bit-change, and rightmost-bit-change detection.

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 clean: An MP instruction that updates external memory by writing modified (dirty) data-cache subblocks back to memory, thus resetting that subblock's dirty bit to 0.

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.

$\overline{\text{CAS}}$: *Column address strobe*. A memory interface signal that drives the column address strobe inputs of DRAMs/VRAMs.

clean: See *cache clean*

coherency: See *cache coherency*

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.

convolution: A time domain reference for digital filtering that makes extensive use of sum-of-products. See *vector dot product*

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.

dcache: See *data cache*

DCR: *Data-cache reset*. A command you send with the MP cmnd instruction that resets all data cache tag registers and the DLRU register.

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*

denormal: A floating-point number with a zero exponent and a nonzero mantissa.

Dhrystones: An algorithm used to benchmark processor performance.

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*

dirty flag: A storage bit associated with each subblock of MP data-cache memory that indicates whether the subblock contains modified data that needs to be written back to main memory. See also *cache flush*, *cache clean*

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.

double-precision floating-point: A floating-point number with 64 bits plus an additional hidden bit.

doubleword: A 64-bit value.

DRAM: *Dynamic random access memory.* Memory typically used for external memory; it must be refreshed periodically to maintain valid data.

E

exception: A condition that is handled outside the normal program flow of a task. An exception in a task is the software equivalent of a hardware interrupt in a processor.

exception flag: A bit in a task's exception register that indicates the status of a particular type of exception.

exception handler: A function associated with a task that handles exceptions raised in that task. The manner in which an exception handler responds to an exception in a task is analogous to the way an interrupt service routine (ISR) responds to a hardware interrupt in a processor.

exception register: A field in a task descriptor that contains the associated task's 32 exception flags.

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.

executive: The portion of a multitasking software system that is responsible for executing application tasks, providing communications among tasks, and managing shared resources.

external address: See *off-chip address*

externally initiated packet transfer: See *XPT*

F

fast Fourier transform: See *FFT*

fast mode: An MP floating-point unit mode in which MP floating-point instructions are executed with all denormals treated as 0 for both input operands and the output result.

fault: Any condition that causes a system to fail.

FEA pipeline sequence: *Fetch, execute, access pipeline sequence.* The instruction-execution integer unit pipeline for the MP. The fetch stage includes instruction and operand fetch; the execute stage includes operations and register-to-register moves; and the optional access stage includes memory accesses (loads and stores).

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 *FEA 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*

fixed-patch guided transfer: Guided transfer that uses an on-chip guide table consisting of 32-bit word-aligned entries. See also *guided transfer*

floating-point add unit busy: A condition in which the floating-point add unit is stalled or its pipeline is full; therefore, the unit cannot accept a new instruction.

floating-point empty: A condition in which there are no instructions in any of the pipeline stages of the floating-point unit.

floating-point exception handler routine: A routine used to service MP floating-point exception traps in an effort to provide information to the user or to attempt recovery.

floating-point multiply unit busy: A condition in which the floating-point multiply unit is stalled or its pipeline is full; therefore, the unit cannot accept a new instruction.

floating-point unit: See *FPU*

flush: See *cache flush*

FPU: *floating-point unit.* The MP's IEEE-754-standard hardware that consists of a full double-precision floating-point add unit and a double-precision floating-point multiply unit with a single precision core.

frame timers: In the VC, timers that provide video timing control. The frame timers indicate to the serial-register-transfer (SRT) controller when an SRT is necessary.

G

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

ICR: *Instruction-cache reset*. A command you send with the MP cmnd instruction that resets all instruction cache tag registers and the ILRU register.

IEEE-754 standard: A standard sponsored by the Standards Committee of the IEEE Computer Society that describes various aspects for floating-point numbers, such as definitions, formats, exceptions, and rounding details.

IEEE mode: An MP floating-point unit mode in which MP floating-point instructions are executed with denormals handled as defined in the IEEE-754 standard.

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.

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.

IR: *Instruction register*. The MP register containing the actual instruction being executed.

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*

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.

long immediate: On the MP, a 32-bit value that is either a signed or unsigned integer, a single-precision floating-point constant, a relative branch word offset, or an absolute address. The MP long-immediate instruction format requires two 32-bit instruction words.

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

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.

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.

mpcl: A shell utility that invokes the MVP master processor compiler, assembler, and linker to create an executable object file version of your MP program.

MPEG standard: *Moving Picture Experts Group standard.* A proposed standard for compressed video data.

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*

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*

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*

P

P × 64: A CCITT video teleconferencing standard designed to permit full duplex video displays over data lines that have a bit rate of P × 64, where $1 < P < 30$.

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*

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.

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.

pipelined mode: An MP floating-point unit mode in which multiple MP floating-point instructions are in various stages of completion in the floating-point multiply and/or add unit simultaneously.

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.

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 $\times$ 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.

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.

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.

Q

QNaN: *Quiet not-a-number.* A floating-point number with no numerical value that is used as a signal to the MP in certain exception conditions.

R

refresh: A method of restoring the charge capacitance to a memory device (such as a DRAM or VRAM) or of restoring memory contents.

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.

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

sequential mode: An MP floating-point unit mode in which the MP executes a single floating-point instruction to completion before starting another floating-point instruction.

short immediate: On the MP, a 15-bit signed or unsigned integer provided by MP instructions as one of the operands within a 32-bit instruction format.

single-precision floating-point: 32-bit floating-point number.

SNaN: *Signaling not-a-number.* A floating-point number that has no numerical value but is used to signal the MP in certain exception conditions.

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.

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.

supervisor mode: A mode in which the MP has write access to all control registers and can write into the MP parameter RAM.

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*

trap: An exceptional condition caused by the currently executing instruction that forces a program to be interrupted before execution of the next instruction begins. After the processor has serviced the trap, it typically resumes execution of the interrupted program at the instruction that immediately follows the instruction that caused the trap.

user mode: A mode in which the MP cannot write into control register numbers lower than 0x4000 and cannot write into the MP's parameter RAM.

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*, *guided transfer*

VC: *Video controller*. The portion of the MVP responsible for the video interface.

vector dot product: A mathematical term applied to the sum of the products of individual elements from two different vectors a and b.

vector instruction: An operation that allows you to perform a floating-point operation in parallel with a load or store instruction.

video controller: See *VC*

VRAM: *Video random access memory.* A type of memory that is easily interfaced to a video display.

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

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

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X