Floating-Point Unit

6

This chapter describes the MIPS floating-point unit (FPU) features, including the programming model, instruction set and formats, and the pipeline.

The FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic*. In addition, the MIPS architecture fully supports the recommendations of the standard and precise exceptions.

6.1 Overview

The FPU operates as a coprocessor for the CPU (it is assigned coprocessor label *CP1*), and extends the CPU instruction set to perform arithmetic operations on floating-point values.

Figure 6-1 illustrates the functional organization of the FPU.

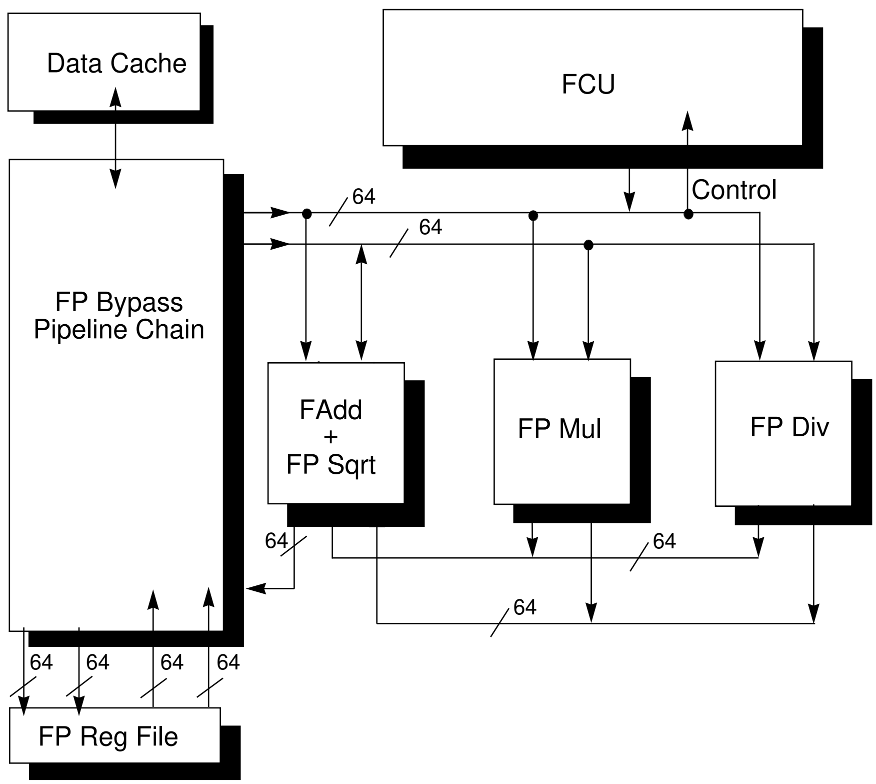


Figure 6-1 FPU Functional Block Diagram

6.2 FPU Features

This section briefly describes the operating model, the load/store instruction set, and the coprocessor interface in the FPU. A more detailed description is given in the sections that follow.

- **Full 64-bit Operation**. When the *FR* bit in the CPU *Status* register equals 0, the FPU is in 32-bit mode and contains thirty-two 32-bit registers that hold single- or, when used in pairs, double-precision values. When the *FR* bit in the CPU *Status* register equals 1, the FPU is in 64-bit mode and the registers are expanded to 64 bits wide. Each register can hold single- or double-precision values. The FPU also includes a 32-bit *Control/Status* register that provides access to all IEEE-Standard exception handling capabilities.
- **Load and Store Instruction Set**. Like the CPU, the FPU uses a load- and store-oriented instruction set, with single-cycle load and store operations. Floating-point operations are started in a single cycle and their execution overlaps other fixed-point or floating-point operations.
- **Tightly Coupled Coprocessor Interface**. The FPU resides on-chip to form a tightly coupled unit with a seamless integration of floating-point and fixed-point instruction sets. Since each unit receives and executes instructions in parallel, some floating-point instructions can execute at the same single-cycle-per-instruction rate as fixed-point instructions.

6.3 FPU Programming Model

This section describes the set of FPU registers and their data organization. The FPU registers include *Floating-Point General Purpose* registers *(FGRs)* and two control registers: *Control/Status* and *Implementation/Revision*.

Floating-Point General Registers (FGRs)

The FPU has a set of *Floating-Point General Purpose* registers *(FGRs)* that can be accessed in the following ways:

- As 32 general purpose registers (32 FGRs), each of which is 32 bits wide when the *FR* bit in the CPU *Status* register equals 0; or as 32 general purpose registers (32 FGRs), each of which is 64-bits wide when *FR* equals 1. The CPU accesses these registers through move, load, and store instructions.
- As 16 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the CPU *Status* register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacently numbered FGRs as shown in Figure 6-2.
- As 32 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the CPU *Status* register equals 1. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to an FGR as shown in Figure 6-2.

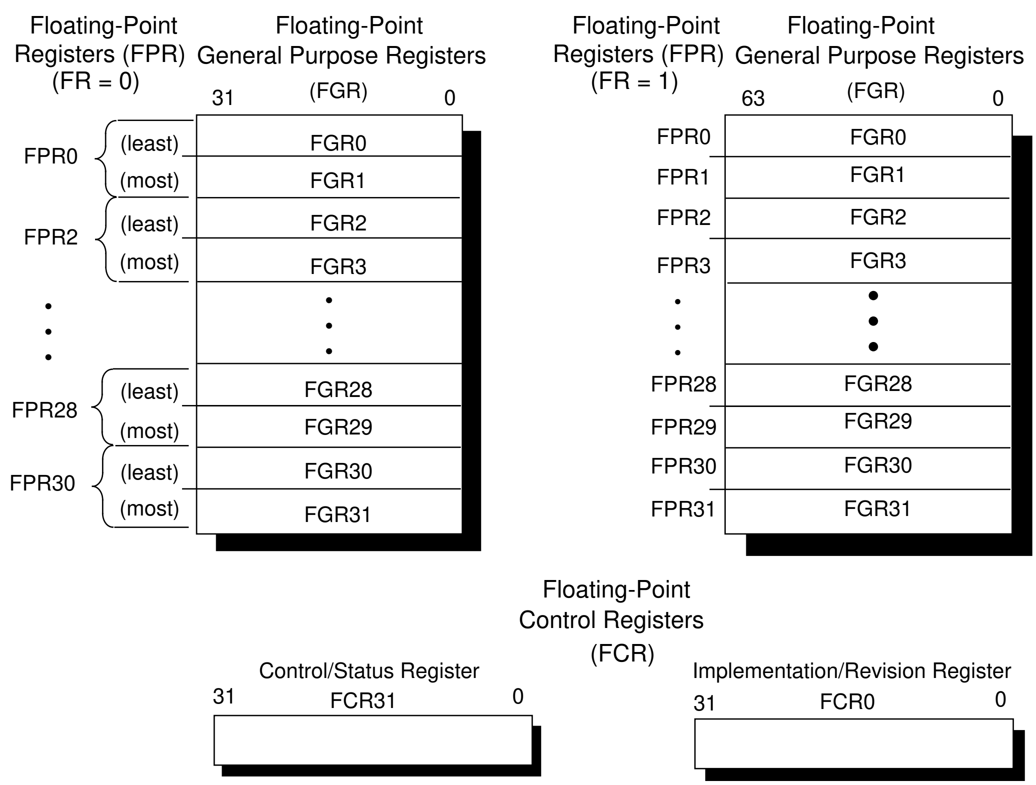


Figure 6-2 FPU Registers

Floating-Point Registers

The FPU provides:

- 16 *Floating-Point* registers (*FPR*s) when the *FR* bit in the *Status* register equals 0, or
- 32 *Floating-Point* registers (*FPR*s) when the *FR* bit in the *Status* register equals 1.

These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *General Purpose* registers (*FGRs*). When the *FR* bit in the *Status* register equals 1, the *FPR* references a single 64-bit *FGR.*

The *FPR*s hold values in either single- or double-precision floating-point format. If the *FR* bit equals 0, only even numbers (the *least* register, as shown in Figure 6-2) can be used to address *FPR*s. When the *FR* bit is set to a 1, all *FPR* register numbers are valid.

If the *FR* bit equals 0 during a double-precision floating-point operation, the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting *Floating-Point Register 0* (*FPR0*) actually addresses adjacent *Floating-Point General Purpose* registers *FGR0* and *FGR1*.

Floating-Point Control Registers

The FPU has 32 control registers (*FCR*s) that can only be accessed by move operations. The *FCR*s are described below:

- The *Implementation/Revision* register *(FCR0)* holds revision information about the FPU.
- The *Control/Status* register *(FCR31)* controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.
- *FCR1* to *FCR30* are reserved.

Table 6-1 lists the assignments of the *FCR*s.

Table 6-1 Floating-Point Control Register Assignments

FCR Number	Use
FCR0	Coprocessor implementation and revision register
FCR1 to FCR30	Reserved
FCR31	Rounding mode, cause, trap enables, and flags

Implementation and Revision Register, (FCR0)

The read-only *Implementation and Revision* register (*FCR0*) specifies the implementation and revision number of the FPU. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software.

Figure 6-3 shows the layout of the register; Table 6-2 describes the *Implementation and Revision* register (*FCR0*) fields.

Implementation/Revision Register (FCR0)

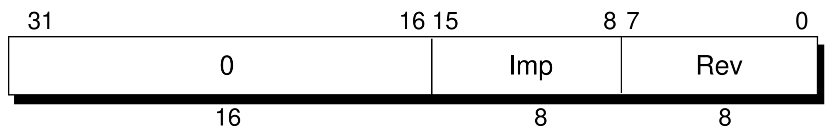


Figure 6-3 Implementation/Revision Register

Table 6-2 FCR0 Fields

Field	Description
Imp	Implementation number (0x05)
Rev	Revision number in the form of *y.x*
0	Reserved. Must be written as zeroes, and returns zeroes when read.

The revision number is a value of the form *y.x*, where:

- *y* is a major revision number held in bits 7:4.
- *x* is a minor revision number held in bits 3:0.

The revision number distinguishes some chip revisions; however, MIPS does not guarantee that changes to its chips are necessarily reflected by the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason revision number values are not listed, and software should not rely on the revision number to characterize the chip.

Control/Status Register (FCR31)

The *Control / Status* register *(FCR31*) contains control and status information that can be accessed by instructions in either Kernel or User mode. *FCR31* also controls the arithmetic rounding mode and enables User mode traps, as well as identifying any exceptions that may have occurred in the most recently executed instruction, along with any exceptions that may have occurred without being trapped.

Figure 6-4 shows the format of the *Control/Status* register, and Table 6-3 describes the *Control/Status* register fields. Figure 6-5 shows the *Control/ Status* register *Cause, Flag*, and *Enable* fields.

Control/Status Register (FCR31)

31 — 25	24	23	22 — 18	17 — 12	11 — 7	6 — 2	1 — 0
0	FS	C	0	Cause E V Z O U I	Enables V Z O U I	Flags V Z O U I	RM
7	1	1	5	6	5	5	2

Figure 6-4 FP Control/Status Register Bit Assignments

Table 6-3 Control/Status Register Fields

Field	Description
FS	When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.
C	Condition bit. See description of *Control/Status* register *Condition* bit.
Cause	Cause bits. See Figure 6-5 and the description of *Control/Status* register *Cause, Flag*, and *Enable* bits.
Enables	Enable bits. See Figure 6-5 and the description of *Control/Status* register *Cause, Flag*, and *Enable* bits.
Flags	Flag bits. See Figure 6-5 and the description of *Control/Status* register *Cause, Flag*, and *Enable* bits.
RM	Rounding mode bits. See Table 6-4 and the description of *Control/Status* register *Rounding Mode Control* bits.

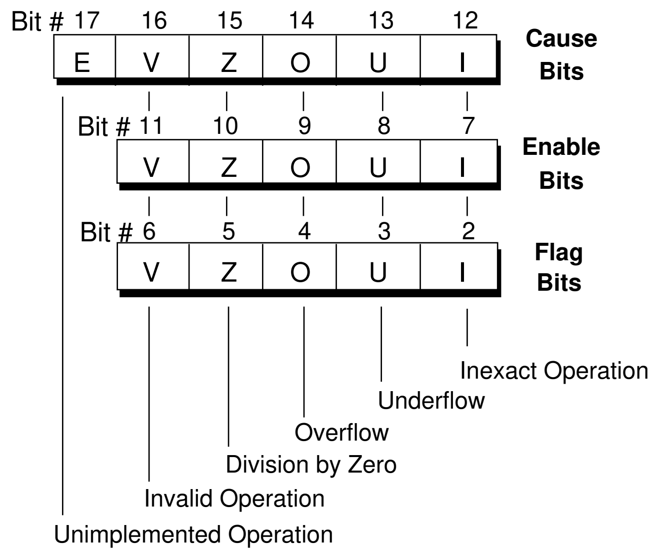


Figure 6-5 Control/Status Register Cause, Flag, and Enable Fields

Accessing the Control/Status Register

When the *Control/Status* register is read by a Move Control From Coprocessor 1 (CFC1) instruction, all unfinished instructions in the pipeline are completed before the contents of the register are moved to the main processor. If a floating-point exception occurs as the pipeline empties, the FP exception is taken and the CFC1 instruction is re-executed after the exception is serviced.

The bits in the *Control/Status* register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. *FCR31* must only be written to when the FPU is not actively executing floating-point operations; this can be ensured by reading the contents of the register to empty the pipeline.

IEEE Standard 754

IEEE Standard 754 specifies that floating-point operations detect certain exceptional cases, raise flags, and can invoke an exception handler when an exception occurs. These features are implemented in the MIPS architecture with the *Cause*, *Enable*, and *Flag* fields of the *Control/Status* register. The *Flag* bits implement IEEE 754 exception status flags, and the *Cause* and *Enable* bits implement exception handling.

Control/Status Register FS Bit

When the *FS* bit is set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.

Control/Status Register Condition Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the *Condition* bit, to save or restore the state of the condition line. The *C* bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by compare and Move Control To FPU instructions.

Control/Status Register Cause, Flag, and Enable Fields

Figure 6-5 illustrates the *Cause*, *Flag*, and *Enable* fields of the *Control/Status* register.

Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 6-5, which reflect the results of the most recently executed instruction. The *Cause* bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation and raise an interrupt or exception if the corresponding enable bit is set. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are written by each floating-point operation (but not by load, store, or move operations). The Unimplemented Operation (*E*) bit is set to a 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE 754 exception.

When a floating-point exception is taken, no results are stored, and the only state affected is the *Cause* bit.

Enable Bits

A floating-point exception is generated any time a *Cause* bit and the corresponding *Enable* bit are set. A floating-point operation that sets an enabled *Cause* bit forces an immediate exception, as does setting both *Cause* and *Enable* bits with CTC1.

There is no enable for Unimplemented Operation (*E*). Setting Unimplemented Operation always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the enabled *Cause* bits with a CTC1 instruction to prevent a repeat of the interrupt. Thus, User mode programs can never observe enabled *Cause* bits set; if this information is required in a User mode handler, it must be passed somewhere other than the *Status* register.

For a floating-point operation that sets only unenabled *Cause* bits, no exception occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the *Cause* field.

Flag Bits

The *Flag* bits are cumulative and indicate that an exception was raised by an operation that was executed since they were explicitly reset. *Flag* bits are set to 1 if an IEEE 754 exception is raised, otherwise they remain unchanged. The *Flag* bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the *Status* register, using a Move To Coprocessor Control instruction.

When a floating-point exception is taken, the flag bits are not set by the hardware; floating-point exception software is responsible for setting these bits before invoking a user handler.

Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the *Control/Status* register constitute the *Rounding Mode* (*RM*) field.

As shown in Table 6-4, these bits specify the rounding mode that the FPU uses for all floating-point operations.

Table 6-4 Rounding Mode Bit Decoding

Rounding Mode RM(1:0)	Mnemonic	Description
0	RN	Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near.
1	RZ	Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise result.
2	RP	Round toward +∞: round to value closest to and not less than the infinitely precise result.
3	RM	Round toward – ∞: round to value closest to and not greater than the infinitely precise result.

6.4 Floating-Point Formats

The FPU performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field (*f+s*) and an 8-bit exponent (*e*), as shown in Figure 6-6.

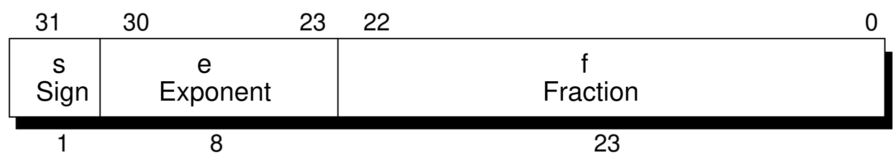


Figure 6-6 Single-Precision Floating-Point Format

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field (*f+s)* and an 11-bit exponent, as shown in Figure 6-7.

Figure 6-7 Double-Precision Floating-Point Format

As shown in the above figures, numbers in floating-point format are composed of three fields:

- sign field, *s*
- biased exponent, $e = E + bias$
- fraction, $f = .b_1 b_2 .... b_{p-1}$

The range of the unbiased exponent *E* includes every integer between the two values E_{min} and E_{max} inclusive, together with two other reserved values:

- E_{min} -1 (to encode 0 and denormalized numbers)
- E_{max} +1 (to encode ∞ and NaNs [Not a Number])

For single- and double-precision formats, each representable nonzero numerical value has just one encoding.

For single- and double-precision formats, the value of a number, v, is determined by the equations shown in Table 6-5.

Table 6-5 Equations for Calculating Values in Single and Double-Precision Floating-Point Format

No.	Equation
(1)	if $E = E_{max}+1$ and $f \neq 0$, then v is NaN, regardless of s
(2)	if $E = E_{max}+1$ and $f = 0$, then $v = (-1)^s \infty$
(3)	if $E_{min} \leq E \leq E_{max}$, then $v = (-1)^s 2^E (1.f)$
(4)	if $E = E_{min}-1$ and $f \neq 0$, then $v = (-1)^s 2^{Emin} (0.f)$
(5)	if $E = E_{min}-1$ and $f = 0$, then $v = (-1)^s 0$

For all floating-point formats, if v is NaN, the most-significant bit of f determines whether the value is a signaling or quiet NaN: v is a signaling NaN if the most-significant bit of f is set, otherwise, v is a quiet NaN.

Table 6-6 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 6-7.

Table 6-6 Floating-Point Format Parameter Values

Parameter	Format	
	Single	Double
E_{max}	+127	+1023
E_{min}	–126	–1022
Exponent *bias*	+127	+1023
Exponent width in bits	8	11
Integer bit	hidden	hidden
f (Fraction width in bits)	24	53
Format width in bits	32	64

Table 6-7 Minimum and Maximum Floating-Point Values

Type	Value
Float Minimum	1.40129846e–45
Float Minimum Norm	1.17549435e–38
Float Maximum	3.40282347e+38
Double Minimum	4.9406564584124654e–324
Double Minimum Norm	2.2250738585072014e–308
Double Maximum	1.7976931348623157e+308

6.5 Binary Fixed-Point Format

Binary fixed-point values are held in 2's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 6-8 illustrates binary fixed-point format; Table 6-8 lists the binary fixed-point format fields.

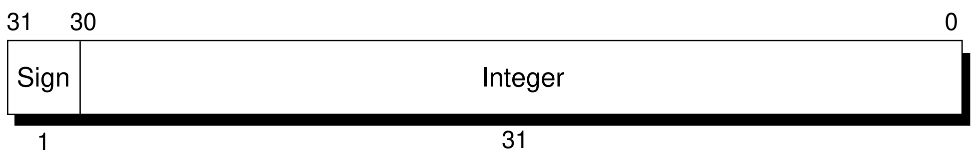


Figure 6-8 Binary Fixed-Point Format

Field assignments of the binary fixed-point format are:

Table 6-8 Binary Fixed-Point Format Fields

Field	Description
sign	sign bit
integer	integer value

6.6 Floating-Point Instruction Set Overview

All FPU instructions are 32 bits long, aligned on a word boundary. They can be divided into the following groups:

- **Load, Store, and Move** instructions move data between memory, the main processor, and the *FPU General Purpose* registers.
- **Conversion** instructions perform conversion operations between the various data formats.
- **Computational** instructions perform arithmetic operations on floating-point values in the FPU registers.
- **Compare** instructions perform comparisons of the contents of registers and set a conditional bit based on the results.
- **Branch on FPU Condition** instructions perform a branch to the specified target if the specified coprocessor condition is met.

In the instruction formats shown in Tables 6-9 through 6-12, the *fmt* appended to the instruction opcode specifies the data format: *S* specifies single-precision binary floating-point, *D* specifies double-precision binary floating-point, *W* specifies 32-bit binary fixed-point, and *L* specifies 64-bit (long) binary fixed-point.

Table 6-9 FPU Instruction Summary: Load, Move and Store Instructions

OpCode	Description
LWC1	Load Word to FPU
SWC1	Store Word from FPU
LDC1	Load Doubleword to FPU
SDC1	Store Doubleword From FPU
MTC1	Move Word To FPU
MFC1	Move Word From FPU
CTC1	Move Control Word To FPU
CFC1	Move Control Word From FPU
DMTC1	Doubleword Move To FPU
DMFC1	Doubleword Move From FPU

Table 6-10 FPU Instruction Summary: Conversion Instructions

OpCode	Description
CVT.S.fmt	Floating-point Convert to Single FP
CVT.D.fmt	Floating-point Convert to Double FP
CVT.W.fmt	Floating-point Convert to 32-bit Fixed Point
CVT.L.fmt	Floating-point Convert to 64-bit Fixed Point
ROUND.W.fmt	Floating-point Round to 32-bit Fixed Point
ROUND.L.fmt	Floating-point Round to 64-bit Fixed Point
TRUNC.W.fmt	Floating-point Truncate to 32-bit Fixed Point
TRUNC.L.fmt	Floating-point Truncate to 64-bit Fixed Point
CEIL.W.fmt	Floating-point Ceiling to 32-bit Fixed Point
CEIL.L.fmt	Floating-point Ceiling to 64-bit Fixed Point
FLOOR.W.fmt	Floating-point Floor to 32-bit Fixed Point
FLOOR.L.fmt	Floating-point Floor to 64-bit Fixed Point

Table 6-11 FPU Instruction Summary: Computational Instructions

OpCode	Description
ADD.fmt	Floating-point Add
SUB.fmt	Floating-point Subtract
MUL.fmt	Floating-point Multiply
DIV.fmt	Floating-point Divide
ABS.fmt	Floating-point Absolute Value
MOV.fmt	Floating-point Move
NEG.fmt	Floating-point Negate
SQRT.fmt	Floating-point Square Root

Table 6-12 FPU Instruction Summary: Compare and Branch Instructions

OpCode	Description
C.cond.fmt	Floating-point Compare
BC1T	Branch on FPU True
BC1F	Branch on FPU False
BC1TL	Branch on FPU True Likely
BC1FL	Branch on FPU False Likely

Floating-Point Load, Store, and Move Instructions

This section discusses the manner in which the FPU uses the load, store and move instructions listed in Table 6-9; Appendix B provides a detailed description of each instruction.

Transfers Between FPU and Memory

All data movement between the FPU and memory is accomplished by using one of the following instructions:

- Load Word To Coprocessor 1 (LWC1) or Store Word From Coprocessor 1 (SWC1) instructions, which reference a single 32-bit word of the FPU general registers
- Load Doubleword (LDC1) or Store Doubleword (SDC1) instructions, which reference a 64-bit doubleword.

These load and store operations are unformatted; no format conversions are performed and therefore no floating-point exceptions can occur due to these operations.

Transfers Between FPU and CPU

Data can also be moved directly between the FPU and the CPU by using one of the following instructions:

- Move To Coprocessor 1 (MTC1)
- Move From Coprocessor 1 (MFC1)
- Doubleword Move To Coprocessor 1 (DMTC1)
- Doubleword Move From Coprocessor 1 (DMFC1)

Like the floating-point load and store operations, these operations perform no format conversions and never cause floating-point exceptions.

Load Delay and Hardware Interlocks

The instruction immediately following a load can use the contents of the loaded register. In such cases the hardware interlocks, requiring additional real cycles; for this reason, scheduling load delay slots is desirable, although it is not required for functional code.

Data Alignment

All coprocessor loads and stores reference the following aligned data items:

- For word loads and stores, the access type is always WORD, and the low-order 2 bits of the address must always be 0.
- For doubleword loads and stores, the access type is always DOUBLEWORD, and the low-order 3 bits of the address must always be 0.

Endianness

Regardless of byte-numbering order (endianness) of the data, the address specifies the byte that has the smallest byte address in the addressed field. For a big-endian system, it is the leftmost byte; for a little-endian system, it is the rightmost byte.

Floating-Point Conversion Instructions

Conversion instructions perform conversions between the various data formats such as single- or double-precision, fixed- or floating-point formats. Table 6-10 lists conversion instructions; Appendix B gives a detailed description of each instruction.

Floating-Point Computational Instructions

Computational instructions perform arithmetic operations on floating-point values, in registers. Table 6-11 lists the computational instructions and Appendix B provides a detailed description of each instruction. There are two categories of computational instructions:

- 3-Operand Register-Type instructions, which perform floating-point addition, subtraction, multiplication, and division
- 2-Operand Register-Type instructions, which perform floating-point absolute value, move, negate, and square root operations

Branch on FPU Condition Instructions

Table 6-12 lists the Branch on FPU (coprocessor unit 1) condition instructions that can test the result of the FPU compare (C.cond) instructions. Appendix B gives a detailed description of each instruction.

Floating-Point Compare Operations

The floating-point compare (C.fmt.cond) instructions interpret the contents of two FPU registers (*fs, ft*) in the specified format (*fmt*) and arithmetically compare them. A result is determined based on the comparison and conditions (*cond*) specified in the instruction.

Table 6-12 lists the compare instructions; Appendix B gives a detailed description of each instruction. Table 6-13 lists the mnemonics for the compare instruction conditions.

Table 6-13 Mnemonics and Definitions of Compare Instruction Conditions

Mnemonic	Definition	Mnemonic	Definition
T	True	**F**	False
OR	Ordered	**UN**	Unordered
NEQ	Not Equal	**EQ**	Equal
OLG	Ordered or Less Than or Greater Than	**UEQ**	Unordered or Equal
UGE	Unordered or Greater Than or Equal	**OLT**	Ordered Less Than
OGE	Ordered Greater Than	**ULT**	Unordered or Less Than
UGT	Unordered or Greater Than	**OLE**	Ordered Less Than or Equal
OGT	Ordered Greater Than	**ULE**	Unordered or Less Than or Equal
ST	Signaling True	**SF**	Signaling False
GLE	Greater Than, or Less Than or Equal	**NGLE**	Not Greater Than or Less Than or Equal
SNE	Signaling Not Equal	**SEQ**	Signaling Equal
GL	Greater Than or Less Than	**NGL**	Not Greater Than or Less Than
NLT	Not Less Than	**LT**	Less Than
GE	Greater Than or Equal	**NGE**	Not Greater Than or Equal
NLE	Not Less Than or Equal	**LE**	Less Than or Equal
GT	Greater Than	**NGT**	Not Greater Than

6.7 FPU Instruction Pipeline Overview

The FPU provides an instruction pipeline that parallels the CPU instruction pipeline. It shares the same eight-stage pipeline architecture with the CPU (see Chapter 3).

Instruction Execution

Figure 6-9 illustrates the 8-instruction overlap in the FPU pipeline.

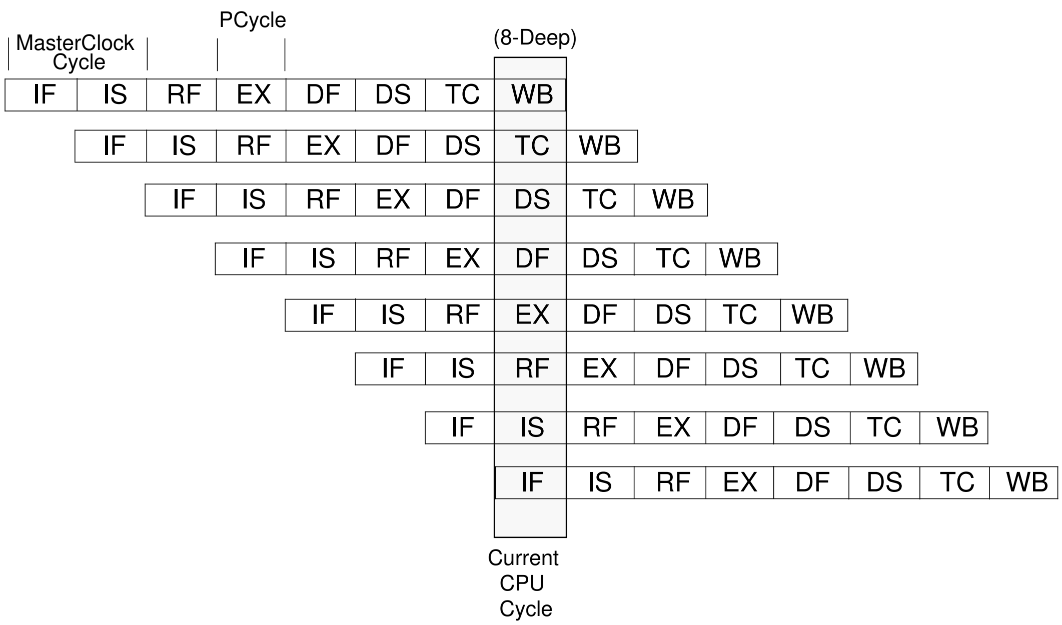


Figure 6-9 FPU Instruction Pipeline

Figure 6-9 assumes that one instruction is completed every PCycle. Most FPU instructions, however, require more than one cycle in the EX stage. This means the FPU must stall the pipeline if an instruction execution cannot proceed because of register or resource conflicts.

Figure 6-10 illustrates the effect of a three-cycle stall on the FPU pipeline.

IF	IS	RF	EX	DF	stall	stall	stall	DS	TC	WB				
	IF	IS	RF	EX	stall	stall	stall	DF	DS	TC	WB			
		IF	IS	RF	stall	stall	stall	EX	DF	DS	TC	WB		
			IF	IS	stall	stall	stall	RF	EX	DF	DS	TC	WB	
				IF	stall	stall	stall	IS	RF	EX	DF	DS	TC	WB

Figure 6-10 FPU Pipeline Stall

To lessen the performance impact that results from stalling the instruction pipeline, the FPU allows instructions to overlap so that instruction execution can proceed as long as there are no resource conflicts, data dependencies, or exception conditions. The following sections describe the timing and overlapping of FPU instructions.

Instruction Execution Cycle Time

Unlike the CPU, which executes almost all instructions in a single cycle, more time may be required to execute FPU instructions.

Table 6-14 gives the minimum latency, in processor pipeline cycles, of each floating-point operation for the currently implemented configurations. These latency calculations assume the result of the operation is immediately used in a succeeding operation.

Table 6-14 Floating-Point Operation Latencies

Operation	Pipeline Cycles				Operation	Pipeline Cycles			
	S	D	W	L		S	D	W	L
ADD.fmt	4	4	(a)	(a)	**CVT.[W,L].fmt**	4	4	(a)	(a)
SUB.fmt	4	4	(a)	(a)	**C.fmt.cond**	3	3	(a)	(a)
MUL.fmt	7	8	(a)	(a)	**BC1T**	(b)	1	(b)	(b)
DIV.fmt	23	36	(a)	(a)	**BC1F**	(b)	1	(b)	(b)
SQRT.fmt	54	112	(a)	(a)	**BC1TL**	(b)	1	(b)	(b)
ABS.fmt	2	2	(a)	(a)	**BC1FL**	(b)	1	(b)	(b)
MOV.fmt	1	1	(a)	(a)	**LWC1**	(b)	3	(b)	(b)
NEG.fmt	2	2	(a)	(a)	**SWC1**	(b)	1	(b)	(b)
ROUND.[W,L].fmt	4	4	(a)	(a)	**LDC1**	(b)	3	(b)	(b)
TRUNC.[W,L].fmt	4	4	(a)	(a)	**SDC1**	(b)	1	(b)	(b)
CEIL.[W,L].fmt	4	4	(a)	(a)	**MTC1**	(b)	3	(b)	(b)
FLOOR.[W,L].fmt	4	4	(a)	(a)	**MFC1**	(b)	3	(b)	(b)
CVT.S.fmt	(a)	4	6	7	**CTC1**	(b)	3	(b)	(b)
CVT.D.fmt	2	(a)	5	4	**CFC1**	(b)	2	(b)	(b)

(a) .... These operations are illegal.
(b) .... These operations are undefined.

Scheduling FPU Instructions

The floating-point architecture permits the overlapping of floating-point load, store, and move instructions with some of the other processor operations.

To permit this, the FPU coprocessor implements three separate operation (op) units:

- divider
- multiplier
- adder (for remaining operations)

The multiplier and divider can overlap adder operations; however, they use the adder on their final cycles, which imposes some limitations.

The multiplier can begin a new double-precision multiplication every four cycles, and a new single-precision multiplication every three cycles. The adder generally begins a new operation one cycle before the previous cycle completes; therefore, a floating-point addition or subtraction can start every three cycles.

The FPU coprocessor pipeline is fully bypassed and interlocked.

FPU Pipeline Overlapping

Each of the three op units is controlled by an FPU resource scheduler, which issues instructions under constraints described in the following section. Table 6-15 lists the pipe stages used in each of the op units (although not all stages are used by each unit).

Table 6-15 FPU Operational Unit Pipe Stages

Stage	Description
A	FPU Adder Mantissa Add stage
E	FPU Adder Exception Test stage
EX	CPU EX stage
M	FPU Multiplier 1st stage
N	FPU Multiplier 2nd stage
R	FPU Adder Result Round stage
S	FPU Adder Operand Shift stage
U	FPU Unpack stage

Instruction Scheduling Constraints

The FPU resource scheduler is kept from issuing instructions to the FPU op units (adder, multiplier, and divider) by the limitations in their micro-architectures. If any of the following constraints are violated, the op unit assumes the outstanding instruction in its pipe is discarded, and then continues operation on the most recently issued instruction.

FPU Divider Constraints

The FPU divider can handle only one non-overlapped division instruction in its pipe at any one time.

FPU Multiplier Constraints

The FPU multiplier allows up to two pipelined MUL.[S,D] instructions to be processed as long as the following constraints are met:

- Two idle cycles are required after a MUL.S instruction (as shown in Figure 6-11).
- Three idle cycles are required after MUL.D instruction (as shown in Figure 6-12).

These figures are not meant to imply that back-to-back multiplications are allowed. Rather, as shown in Figure 6-11, instructions **I2** and **I3** are illegal and **I5**, **I6**, **I7**, and **I8** are successive stages of **I4**, referenced to **I1**.

Figure 6-12 is similar, in that **I6**, **I7**, and **I8** are successive stages of **I5**.

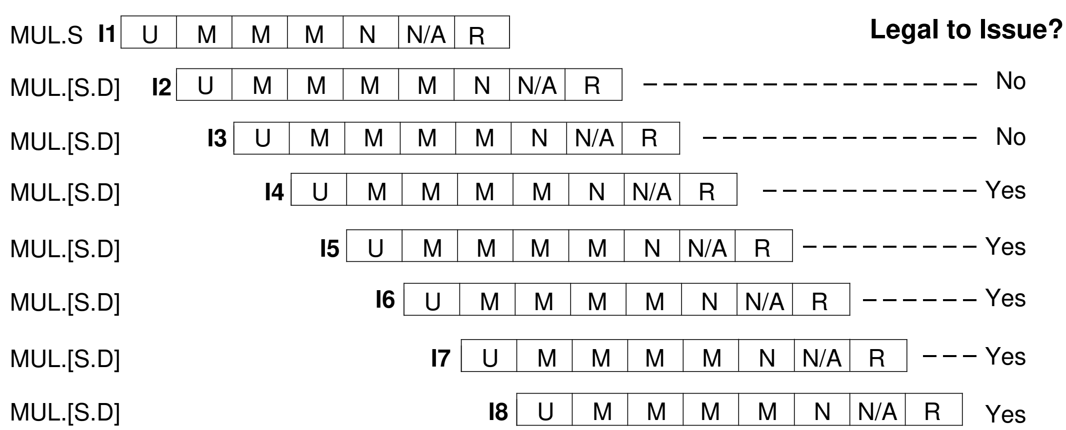


Figure 6-11 MUL.S Instruction Scheduling in the FPU Multiplier

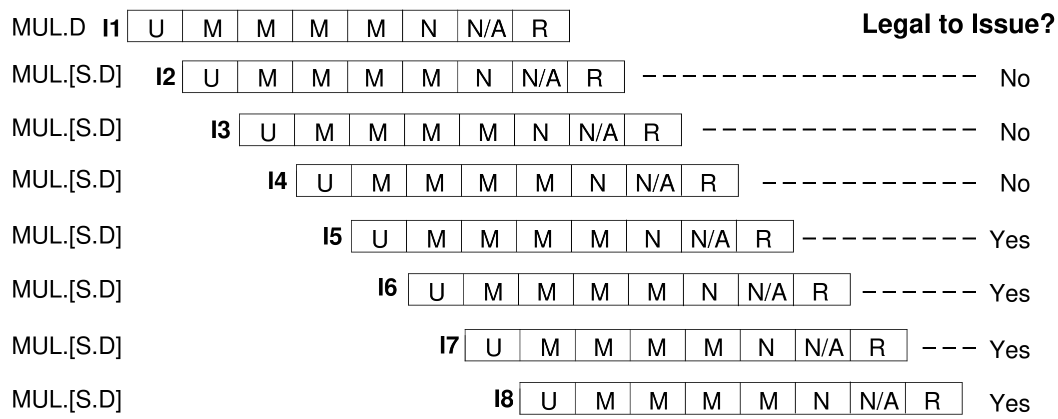


Figure 6-12 MUL.D Instruction Scheduling in the FPU Multiplier

FPU Adder Constraints

Following are the constraints that must be met in the FPU adder op unit.

Cycle Overlap. The adder op unit must allow a clock cycle overlap between each newly issued instruction and the instruction being completed, as shown in Figure 6-13.

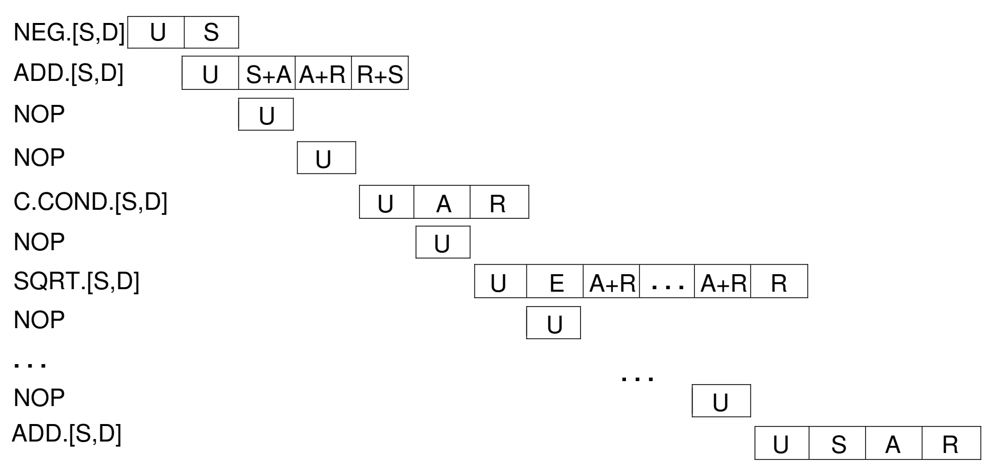


Figure 6-13 Instruction Cycle Overlap in FPU Adder

Resource Conflict. The adder must allow the cleanup stages (A, R) of a multiplication instruction to be pipelined with the execution of an ADD.[S,D], SUB.[S,D], or C.COND.[S,D] instruction, as long as no two instructions simultaneously attempt to use the same A and R pipe stages. For instance, Figure 6-14 shows a resource conflict between the mantissa add (A, stage 7) of instructions 1, 5, and 6. This figure also shows the resource conflict between result round (R), stage 8, of instructions 1, 5, and 6. The multiplication cleanup cycles (A, R) can neither overlap nor pipeline with any other instruction currently in the adder pipe.

Figures 6-14 through 6-17 show these constraints.

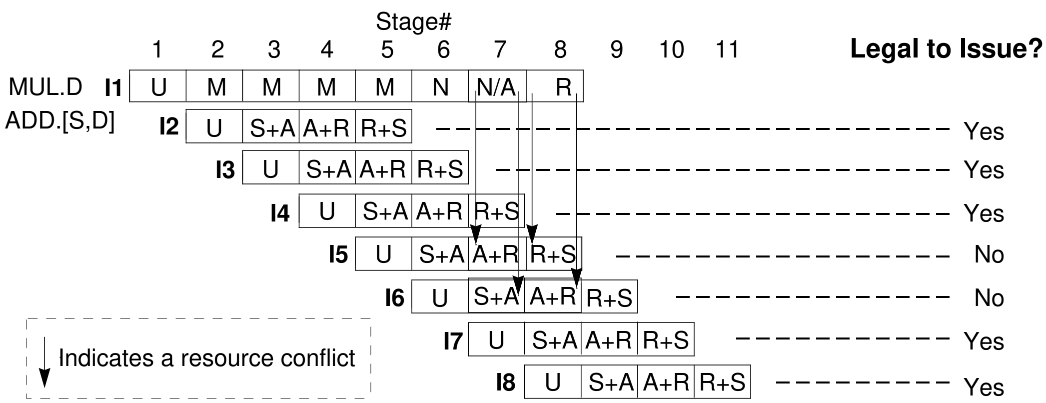


Figure 6-14 MUL.D and ADD.[S,D] Cycle Conflict in FPU Adder

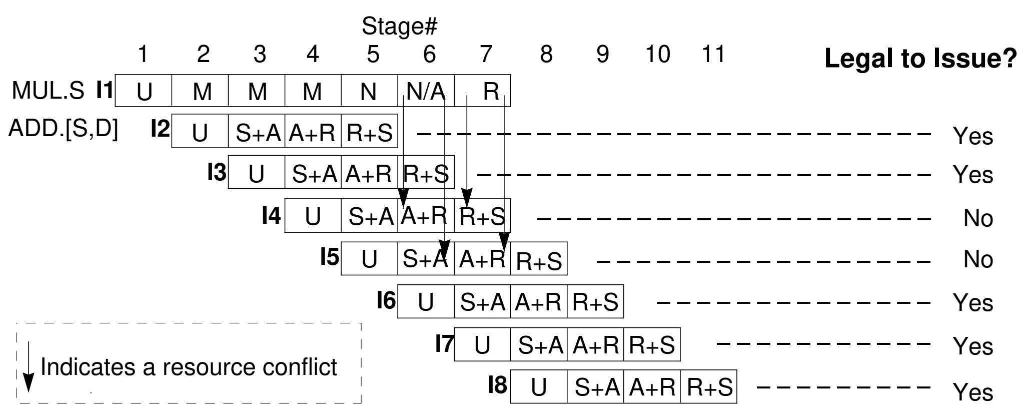
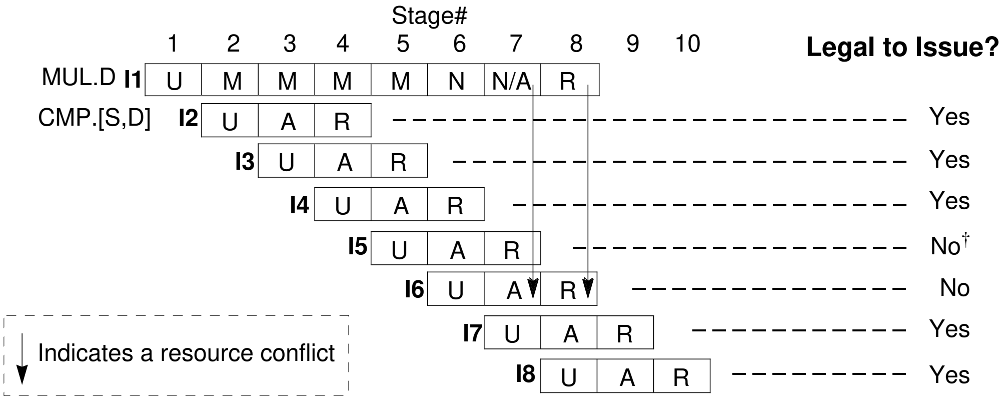
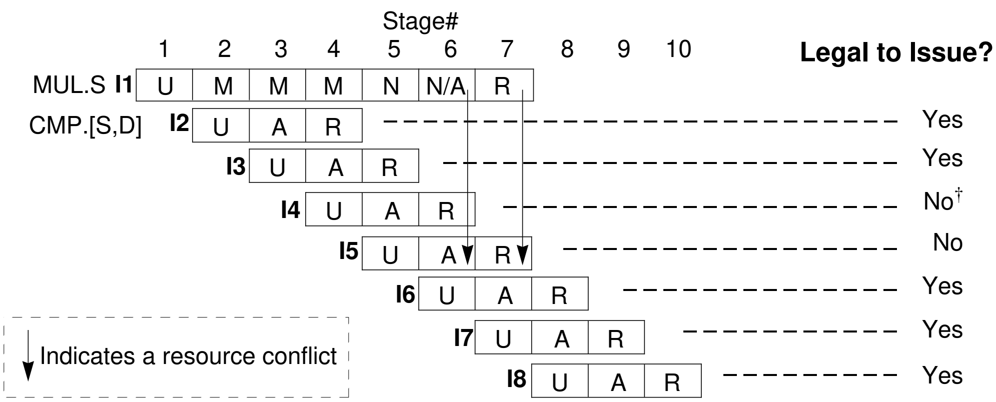


Figure 6-15 MUL.S and ADD.[S,D] Cycle Conflict in FPU Adder



†While there is no resource conflict in issuing this CMP.[S,D] instruction, the hardware does not allow it.

Figure 6-16 MUL.D and CMP.[S,D] Cleanup Cycle Conflict in FPU Adder



†While there is no resource conflict in issuing this CMP.[S,D] instruction, the hardware does not allow it.

Figure 6-17 MUL.S and CMP.[S,D] Cleanup Cycle Conflict in FPU Adder

Prep and Cleanup Cycle Overlap. The adder does not allow the preparation (U stage) and cleanup cycles (N, A, R) of a division instruction to be pipelined with any other instruction; however, the adder does allow the last cycle of preparation or cleanup to be overlapped one clock by the following instruction's U stage (the CPU EX cycle). Figure 6-18 shows this process.

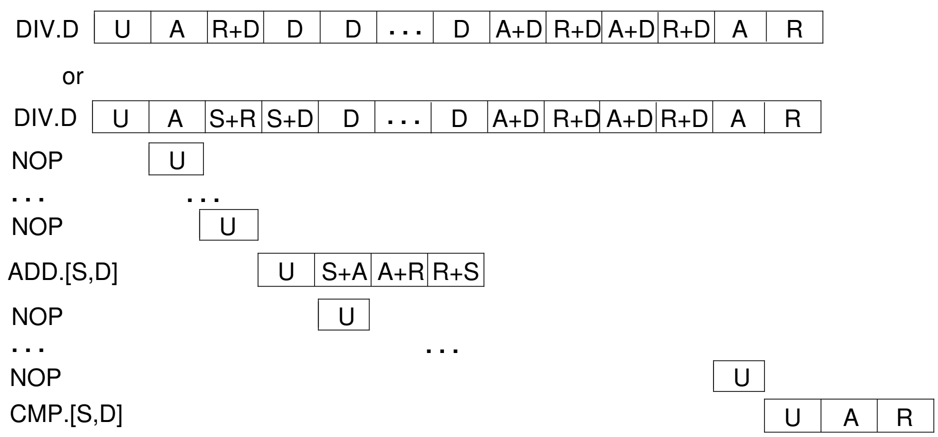


Figure 6-18 Adder Prep and Cleanup Cycle Overlap

Instruction Latency, Repeat Rate, and Pipeline Stage Sequences

Table 6-16 lists the latency and repeat rate between instructions, together with the sequence of pipeline stages for each instruction. For example, the latency of the ADD.[S,D] is 4, which means it takes four processor cycles to complete. The Repeat Rate column indicates how soon an instruction can be repeated; for example, an ADD.[S,D] can be repeated after the conclusion of the third pipeline stage.

Table 6-16 Latency, Repeat Rate, and Pipe Stages of FPU Instructions

Instruction Type	Latency	Repeat Rate	Pipeline Stage Sequence
MOV.[S,D]	1	1	EX
ADD.[S,D]	4	3	U→ S+A→ A+R→ R+S
SUB.[S,D]	4	3	U→ S+A→ A+R→ R+S
C.COND.[S,D]	3	2	U→ A→ R
NEG.[S,D]	2	1	U→ S
ABS.[S,D]	2	1	U→ S
CVT.S.W	6	5	U→ A→ R→ S→ A→ R
CVT.D.W	5	4	U→ S→ A→ R→ S
CVT.S.L	7	6	U→ A→ R→ S→ S→ A→ R
CVT.D.L	4	3	U→ A→ R→ S
CVT.D.S	2	1	U→ S
CVT.S.D	4	3	U→ S→ A→ R
CVT.[W,L].[S,D] *or* ROUND.[W,L].[S,D] *or* TRUNC.[W,L].[S,D] *or* CEIL.[W,L].[S,D] *or* FLOOR.[W,L].[S,D]	4	3	U→ S→ A→ R
MUL.S	7	3	U→ E/M→ M→ M→ N→ N/A→ R
MUL.D	8	4	U→ E/M→ M→ M→ M→ N→ N/A→ R
DIV.S	23	22	U→ S+A→ S+R→ S→ D...D→ D/A→ D/R→ D/A→ D/R→A→R
DIV.D	36	35	U→ A→ R→ D...D→ D/A→ D/R→ D/A → D/R→ A→ R
SQRT.S	2–54	2–53	U→ E→ A+R→...→ A+R→ A→ R
SQRT.D	2–112	2–111	U→ E→ A+R→...→ A+R→ A→ R

Resource Scheduling Rules

The FPU Resource Scheduler issues instructions while adhering to the rules described below. These scheduling rules optimize op unit executions; if the rules are not followed, the hardware interlocks to guarantee correct operation.

DIV.[S,D] can start only when all of the following conditions are met in the RF stage:

- The *divider* is either idle, or in its second-to-last execution cycle.
- The *adder* is either idle, or in its second-to-last execution cycle.
- The *multiplier* is either idle, or in its second-to-last execution cycle.

Idle means an operation unit—adder, multiplier or divider—is either not processing any instruction, or is currently in its last execution cycle completing an instruction.

MUL.[S,D] can start only when all of the following conditions are met in the RF stage:

- The *multiplier* is one of the following:
  - idle, or in its second-to-last execution cycle.
  - not within the first two execution cycles (EX, EX+1) if the most recent instruction in the multiplier pipe is MUL.S
  - not within the first three execution cycles (EX...EX+2) if the most recent instruction in the multiplier pipe is MUL.D
- The *adder* is one of the following:
  - idle, or in its second-to-last execution cycle.
  - not processing the first execution cycle (EX) of CVT.S.L
- The *adder* is not processing a square root instruction
- The *divider* is one of the following:
  - idle, or in its second-to-last execution cycle.
  - in the first 8 execution cycles (EX...EX+7) of a DIV.S
  - in the first 21 execution cycles, except for the second execution cycle, (cycles EX, EX+2...EX+20) of a DIV.D)

SQRT.[S,D] can start only when all of the following conditions are met in the RFstage:

- The *divider* is either idle, or in its second-to-last execution cycle.
- The *adder* is either idle, or in its second-to-last execution cycle.
- The *multiplier* is either idle, or in its second-to-last execution cycle.

CVT.fmt, **NEG.[S,D]** or **ABS.[S,D]** instructions can only start when all of the following conditions are met in the RF stage:

- The *adder* is either idle, or in its second-to-last execution cycle.
- The *multiplier* is either idle, or in its second-to-last execution cycle.
- The *divider* is one of the following:
    - idle, or in its second-to-last execution cycle.
    - in the third through eighth execution cycle (EX+2...EX+7) of a DIV.S
    - in the third through twenty-first execution cycle (EX+2...EX+20) of a DIV.D

ADD.[S,D], **SUB.[S,D]** or **C.COND.[S,D]** can only start when all of the following conditions are met in the RF stage:

- The *adder* is either idle, or in its second-to-last execution cycle.
- The *multiplier* is one of the following:
  - idle, or in its second-to-last execution cycle.
  - not in the third or fourth execution cycles (EX+2...EX+3) if the most recent instruction in the multiplier pipe is MUL.S
  - not in the fourth or fifth execution cycles (EX+3...EX+4) if the most recent instruction in the multiplier pipe is MUL.D
- The *divider* is one of the following:
  - idle, or in its second-to-last execution cycle.
  - in the third through eighth execution cycle (EX+2...EX+7) of a DIV.S
  - in the third through twenty-first execution cycle (EX+2...EX+20) of a DIV.D