SPUR COPROCESSOR INTERFACE DESCRIPTION

Paul M. Hansen

Shing I. Kong

Computer Science Division of EECS
University of California, Berkeley
Berkeley, CA 94720

October 1986

ABSTRACT

This report describes the SPUR coprocessor interface. The interface provides enhanced performance potential by allowing parallel operations between the SPUR processor and SPUR coprocessors. A decoupled control and execution architecture allow data transfers to proceed while coprocessor functions are performed. Implicit and explicit synchronization mechanisms provide the programmer complete control and flexibility. On-chip coprocessor register files and a wide data path between the memory and coprocessor minimize data transfer overhead. An intelligent interface control unit provides parallel decoding of instructions for maximum performance. Other coprocessor functions applicable to performance monitoring, signal processing, image processing, workstation graphics, language coprocessors, and so forth are being considered, but will not be reported here.[†]

[†] Principal funding for the SPUR project is provided by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269. Additional support for this research was provided by the State of California MICRO program, by a Digital Equipment Corporation CAD/CAM grant, by the National Science Foundation under grant DCR-8202591, by equipment donations from Texas Instruments, Inc., and by computer resources provided under DARPA contract N00039-84-C-0089. An overview report on the SPUR architecture is found in [HEL85].

1. INTRODUCTION

The SPUR CPU is a custom VLSI-32 bit general-purpose host targeted to support Lisp and other high-level language software environments. The RISC-like architecture provides high performance for a wide range of applications.

Traditional von Neumann computer architectures have achieved enhanced performance by adding optional hardware to perform tasks that are usually executed in software. These devices are often called *coprocessors* and include attached processors, array processors, floating-point accelerators, data channels, graphics display processors, performance monitors, and so forth. Thus, a coprocessor is an optional piece of hardware that replaces a piece of software for a higher level of performance.

Many peripheral devices as well as more closely coupled coprocessors fall in this general category. It is nevertheless important to recognize a distinction between standard peripheral hardware devices and *tightly coupled* coprocessors: the programming model for the coprocessor differs from that of peripheral devices. Standard peripheral hardware usually appears to the programmer as a set of registers in the memory space of the main processor. The programmer must consider the communication protocol and implement the interface between the peripheral and the device in software.

In contrast to this, the tightly coupled coprocessor adds special instructions to the CPU instruction set that allow the programmer to utilize the coprocessor capabilities. It may also provide additional registers and data types that are not directly supported by the main processor architecture. However, certain interactions needed between the main processor and the coprocessor (i.e., the communications protocol) are implemented in hardware and are transparent to the programmer. Thus, the coprocessor can extend the functions provided to the user without appearing as hardware external to the main processor. This provides a more uniform programming model from a user point of view.

The SPUR system employs an optional special purpose device for floating-point arithmetic. This device will support the IEEE Standard P754 for add, subtract, multiply and divide in single, double, and extended precisions. (Other functions, such as transcendentals, are handled by runtime routines.) We refer to this as the SPUR Floating-point Unit, or simply FPU. For documentation purposes, it would seem logical to refer to all signals and mnemonics related to the coprocessor to be designated "CP". Other applications are being considered besides floating-point arithmetic, but this report will focus on the FPU. Thus, to avoid confusion between the CPU and CP designations, the coprocessor interface signals, blocks, modules and functions will be designated with the "fpu" prefix. The first generation SPUR system will support a floating-point coprocessor (FPU) and a performance monitor (PMC) [Fau86]. Later generations will consider other applications.

Section 2 of this report provides a brief overview of the SPUR coprocessor interface and functions. Section 3 provides a greater degree of detail and timing diagrams for various operations, instructions, and the interaction between the CPU and FPU.

2. FLOATING-POINT COPROCESSOR INTERFACE OVERVIEW

From the assembly language programmers point of view, the SPUR FPU has 15 read/write 87-bit operand registers and one read/write control/status register (nominally 64 bits). The bits in an operand register are defined from left to right (with MSB at left-hand side) as follows:

# of BITS	1	17	64	2	3
FIELD	SIGN	EXPONENT	FRACTION	ROUND TAG	DATA TYPE

The FPU is a load/store architecture. Consequently, all arithmetic operations involve three registers: two source and one destination.

2.1. Instructions

As of this writing, 18 operations are defined for floating-point arithmetic and general coprocessor functions (load, store, etc). They are listed in Table 1. It should be noted that all LOAD instructions have two forms, depending on the cache operation involved: simple read or read with ownership. For example, the two opcodes for loading single precision operands are LD_SGL and LD_SGL_RO.

2.2. Control Flow

The FPU coprocessor has two major functional units: the interface control unit (ICU) and the execution unit (EU). The clocking scheme of the FPU is identical to the CPU: 4 non-overlapping phases per cycle. (Refer to Section 3 for more details.) In phase 3 (phi3) of every cycle, the FPU ICU accepts and decodes the INSTRUCTION BUS fragment issued on *fpuOPCODE_CV3* lines, and initiates operation in the subsequent cycle if it is an FPU operation. This continues until cycle N (N is the number of execution cycles for the particular instruction being executed). The *fpuBusy_C4* signal is disasserted in the last execution cycle (register write) to signal when the FPU EU is done. (See Section 2.6 for complete definitions of signals.)

Under normal circumstances, CPU and FPU instructions execute in parallel. This parallelism is controlled in two possible ways: (1) explicit: the *fpuParallel* bit in the Upsw (user process status word in the CPU) may be set, which will allow overlap of CPU and FPU operation instructions, and (2) implicit: the assertion of

Table 1. SPUR FPU Coprocessor Instructions

ARITHMETIC OPERATIONS, OPERAND CONVERSION, and COMPARE				
Instruction Syntax		Instruction Semantics		
FADD	Rd,Rs1,Rs2	FPU Rd	<-	FPU Rs1 + FPU Rs2
FSUB	Rd,Rs1,Rs2	FPU Rd	<-	FPU Rs1 - FPU Rs2
FMUL	Rd,Rs1,Rs2	FPU Rd	<-	FPU Rs1 * FPU Rs2
FDIV	Rd,Rs1,Rs2	FPU Rd	<-	FPU Rs1 / FPU Rs2
FABS	Rd,Rs1,0	FPU Rd	<-	FPU Rs1 with sign = 0
FNEG	Rd,Rs1,0	FPU Rd	<-	FPU Rs1 with inverted sign
FCMP	cond,Rs1,Rs2	FPSW(cond)	<-	result
CVTS	Rd,Rs1,0	FPU Rd	<-	(convert to single) FPU Rs1
CVTD	Rd,Rs1,0	FPU Rd	<-	(convert to double) FPU Rs1

LOAD COPROCESSOR REGISTERS and FMOV				
Instruction Syntax		Instruction Semantics		
LD_SGL,RO	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)
LD_DBL,RO	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)
LD_EXT1,RO	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)
LD_EXT2,RO	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)
FMOV	Rd,Rs1,0	FPU Rd	<-	FPU Rs1

STORE COPROCESSOR REGISTERS				
Instruction Syntax		Instruction Semantics		
ST_SGL	Rs2,Rs1,SC	FPU Rs2	->	M [(Rs1 + SC)
ST_DBL	Rs2,Rs1,SC	FPU Rs2	->	M [(Rs1 + SC)
ST_EXT1	Rs2,Rs1,SC	FPU Rs2	->	M [(Rs1 + SC)
ST_EXT2	Rs2,Rs1,SC	FPU Rs2	->	M [(Rs1 + SC)

the *fpuBusy_C4* line will prevent the CPU from issuing FPU operation instructions if the FPU is still in the execution phase of a previously issued instruction (as signaled by *fpuBusy_C4*). When overlap is prevented, the CPU always stalls until the *fpuBusy_C4* line is not asserted.

2.3. Data Flow

Data flow between the FPU and the SPUR data cache memory is directly controlled by the CPU. The data path to the cache is 64 bits wide. Double precision operands are loaded in one cycle. Loads may proceed in parallel with FPU operation, since the FPU register file is dual ported. The FPU pipeline is similar to the CPU pipeline: an FPU load requires the instruction fetch, effective addresses calculation, memory access, and register write cycles. Since there is no operand forwarding in the FPU, the load target is not ready for use in the FPU until the third instruction following the load instruction.

Two instructions that allow loading the CPU registers directly from FPU registers, and vice versa without passing through cache memory were included in the initial design, but will not be implemented in the first version. Eventually, these instructions will be useful for transferring integer operands, and control and status information between the CPU and the FPU.

2.4. Performance

Table 2 lists the number of execution cycles needed to complete FPU arithmetic operations. Loads and stores are considered single-cycle operations, and are discussed in Section 3.2.1.

Table 2. SPUR FPU Execution Cycles for Arithmetic Operations

ARITHMETIC OPERATIONS		
Instruction		Cycles (operation only)
FADD	Rd,Rs1,Rs2	3
FSUB	Rd,Rs1,Rs2	3
FMUL	Rd,Rs1,Rs2	8
FDIV	Rd,Rs1,Rs2	20
FABS	Rd,Rs1,0	3
FNEG	Rd,Rs1,0	3
FCMP	cond,Rs1,Rs2	3
CVTS	Rd,Rs1,0	3
CVTD	Rd,Rs1,0	3

Studies comparing the SPUR FPU with commercial microprocessor-based systems employing VLSI floating-point coprocessors indicate that the SPUR-FPU combination can execute several floating-point intensive benchmarks between 5 and 15 times faster than other systems [Han85]. The main performance advantages come from:

(1) the dual ported register file allowing data load/store during FPU operation,

(2) the parallel execution of the FPU and CPU, and

(3) very efficient algorithms and hardware structures for the four operations implemented on-chip: add, subtract, multiply, and divide.

2.5. Programming Interface

The FPU effectively adds new data types, new registers, and new instructions to the CPU. The coordination of the processor-coprocessor operation is handled mostly by the programming languages and coprocessor interface automatically. The FPU architecture is Load/Store, with arithmetic operations between FPU registers. The hardware is invoked directly by program instructions, and no recompilation is necessary for systems that are not equipped with an FPU. Simple link-time command arguments direct the loading of algebraic routines in the absence of the FPU. One bit in the Upsw causes the CPU to trap to algebraic routines when FPU instructions are encountered and an FPU is not available in the system.

2.6. Hardware Interface

Figure 1 shows how the FPU is connected as a coprocessor in the SPUR system. Figure 2 shows the logical interconnections between the CPU and FPU. The CPU and FPU both use a 4-phase non-overlapped clocking scheme as illustrated in Figure 3. The coprocessor interface signals fall into three groups:

(1) instruction: opcode and FPU register specifiers,

(2) control (to FPU): new instruction valid, suspend FPU operation, and

(3) status (from FPU): FPU busy, FPU exception, FPU compare result.

2.6.1. CPU to FPU Signals

Below is a brief description of the signals from the CPU to the FPU.[†] For a more detailed discussion, please refer to Section 3 of this report: Coprocessor Interface Details.

fpuOPCODE_CV3: 7 bits. This specifies the opcode of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

fpuRS1_CV3: 5 bits. This specifies the first source register of the instruction, which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

fpuRS2_CV3: 5 bits. This specifies the second source register of the instruction, which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

[†] If either the CPU or the FPU begins driving a signal at the beginning of phiN, it is assumed that the signal is stable and latchable at the end of phiN at the destination.

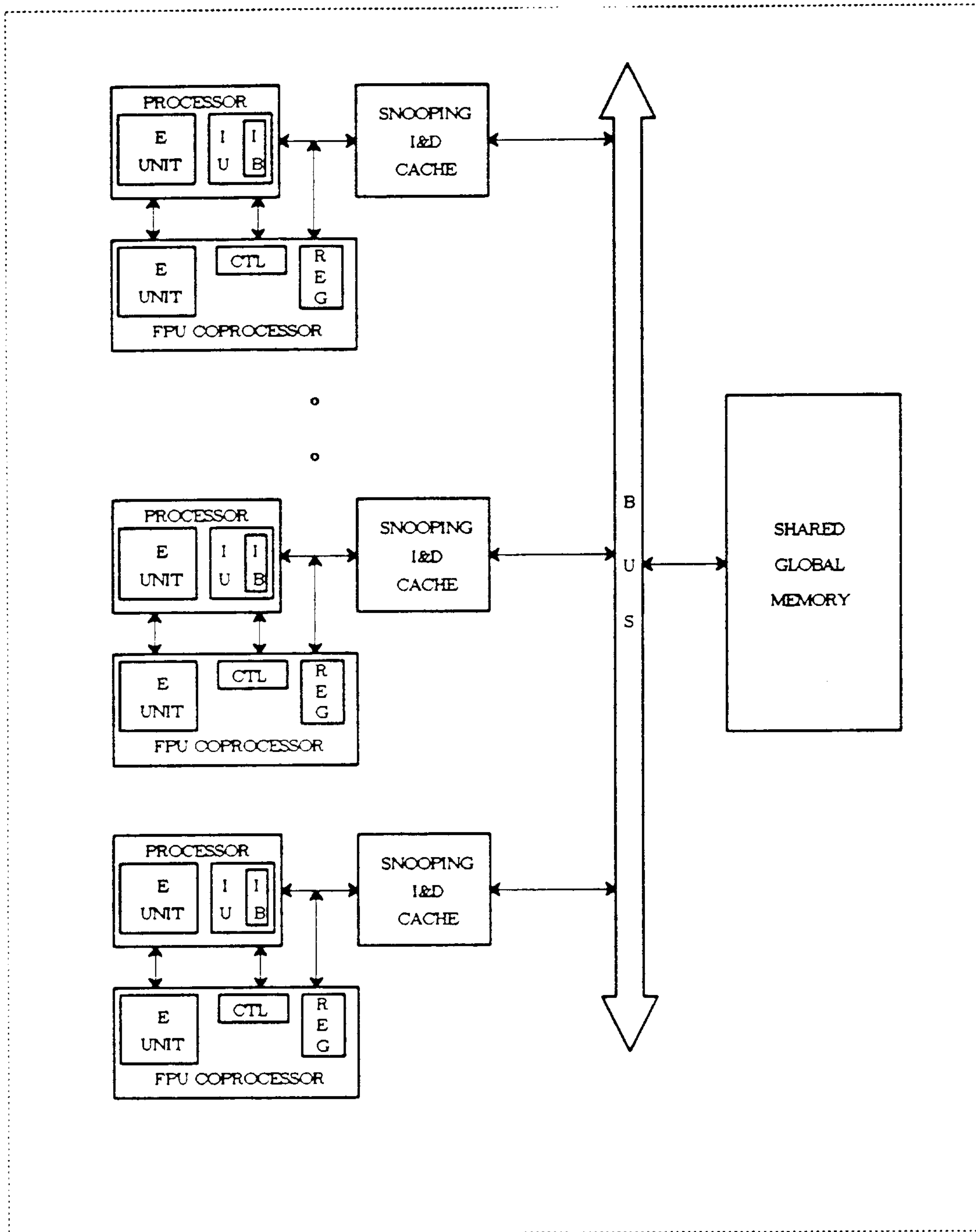


Figure 1. The UC Berkeley SPUR multiprocessor system.

The processor is a single chip with an on-board instruction buffer (IB). The floating-point unit (FPU) is tightly coupled to the CPU via the local processor bus. The cache controller is integrated on one chip with off chip tag and data RAMs. The caches work together to implement a cache consistency protocol, described in [KEP]. Shared memory and I/O devices are accessible through the system bus.

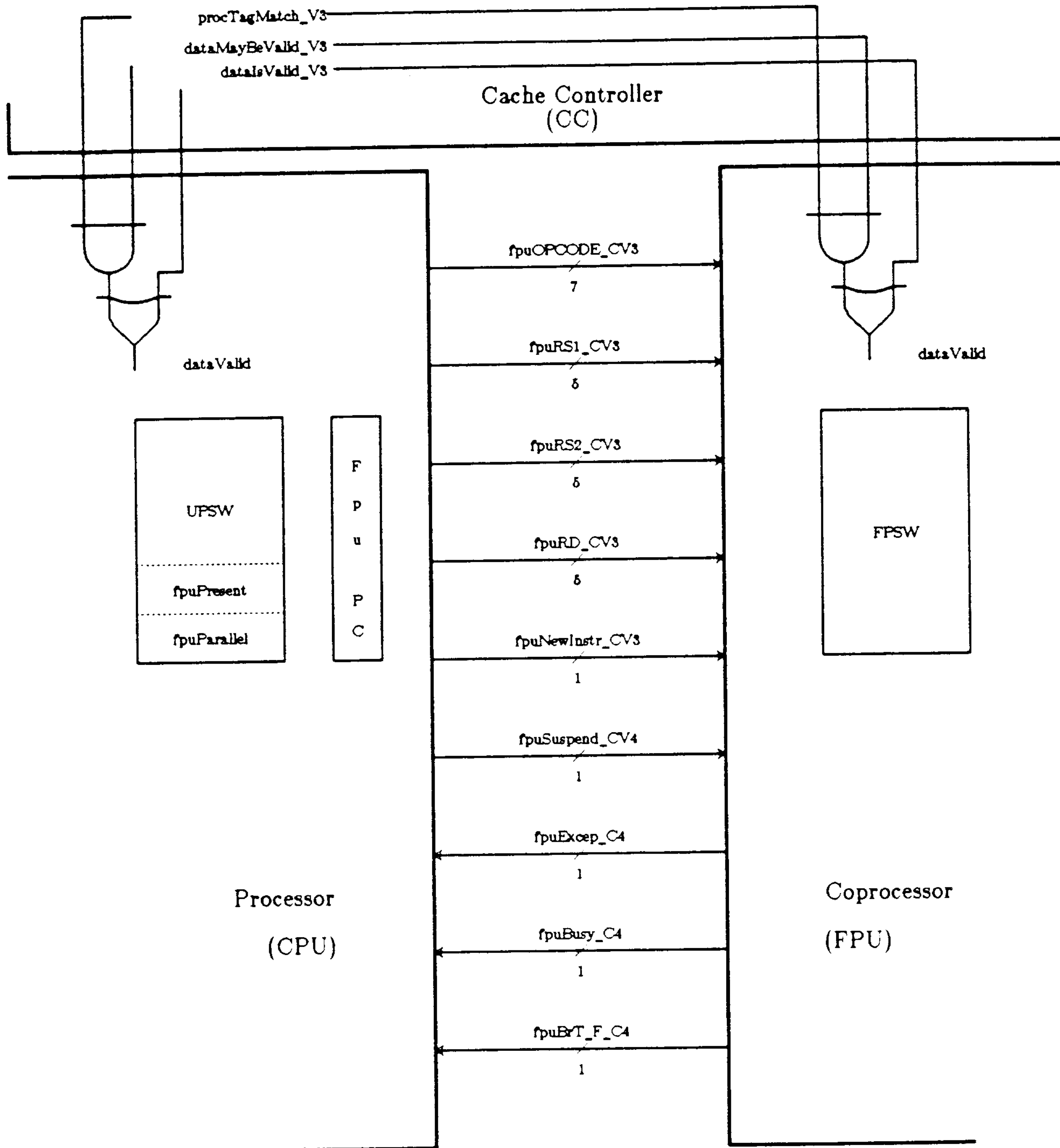


Figure 2. The SPUR coprocessor interface (FPU).
Signals *dataMayBeValid_V3*, *tagMatch_V3*, and *dataIsValid_V3* come from the cache controller to both the CPU and FPU.

fpuRD_CV3: 5 bits. This specifies the Destination register of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

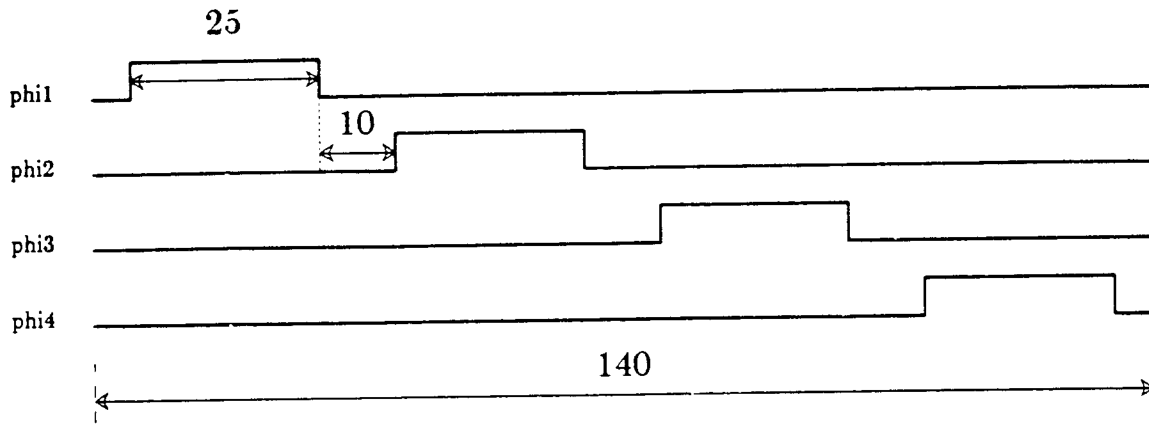


Figure 3. The SPUR system 4-phase clocking scheme.

Clocking is 4-phase, non-overlapping, with 25 nsec high levels, 10 nsec underlap low levels yielding a 140 nsec total cycle time.

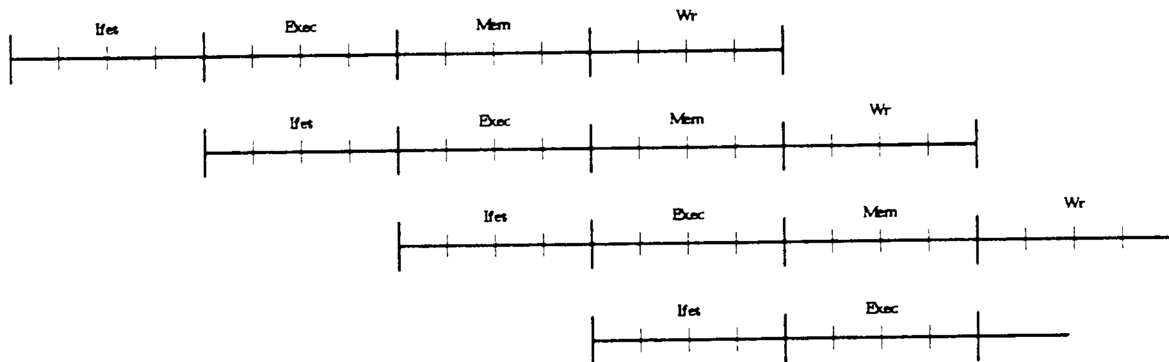


Figure 4. CPU pipeline Stages.

Register operations do not use the Mem cycle, permitting instruction prefetching or operand access during those times. Although it is theoretically possible to issue one instruction per cycle (as shown above), cache misses and/or a busy coprocessor (cannot begin execution of a second instruction until the first is done) will increase the number of effective cycles per instruction completed to more than 1.0. Simulations indicate that between 1.6 and 2.0 clock cycles per instruction are necessary.

fpuNewInstr_CV3: 1 bit. Asserted by the CPU whenever a valid instruction of any variety is issued. The CPU starts driving this signal at the beginning of phi3.

fpuSuspend_CV4: 1 bit. Asserted by the CPU during any pipeline suspension, except when the pipeline is suspended due to *fpuBusy_C4*. (This effectively is a signal to stall FPU register writes.) The CPU starts driving

this signal at the beginning of phi4.

2.6.2. FPU to CPU Signals

The signals between the FPU and CPU which provide status are described next. Many of the details of operation are contained in Section 3 of this report: Coprocessor Interface Details. Note: If the FPU begins driving a signal at the beginning of phiN, it is assumed that the signal is stable and latchable at the end of phiN on the CPU.

fpuBusy_C4: 1 bit. Asserted by the active coprocessor (the FPU) to indicate that it is busy. The FPU starts driving this signal at the beginning of phi4. The CPU latches it at the end of phi1.

fpuExcept_C4: 1 bit. Notifies the CPU that an error condition exists in the coprocessor. The FPU starts driving this signal at the beginning of phi4. The CPU latches it at the end of phi1.

fpuBrT_F_C4: 1 bit. A signal coming from the FPU Fpsw which indicates the result of the last FCMP instruction. The FPU starts driving this signal at the beginning of phi4. The CPU latches it at the end of phi1.

Table 3 summarizes the signals between the CPU and FPU and indicates the phase in which the signals change (driven either by the CPU or FPU) and are latched (by either the CPU or FPU).

2.6.3. CPU UPSW and FPU PC Registers

Besides the signals above, the CPU must maintain the following information in the Upsw to support coprocessors:

fpuParallel: 1 bit. When asserted, it enables parallel operation of the CPU and the FPU. If this bit is not set, parallel operation is prohibited (forcing sequential mode). Parallel operation is described later.

fpuEnable: 1 bit. When asserted, it indicates to the CPU that an FPU device is available in the system. When not asserted, the CPU traps to runtime routines to emulate floating-point hardware operations.

The parallel execution of CPU and FPU instructions requires that the CPU maintain a copy of the last FPU instruction's address which is needed in the event of an exception. Exception handling routines must determine what action is necessary based on the instruction that faulted. However, with parallel operation between the CPU and FPU, the program counter inside the CPU may not be pointing at the FPU instruction that causes the FPU exception. This implies that the CPU must have a special register that stores the address of the last FPU instruction the CPU issued. This special register is the *FpuPC*, which is loaded

Table 3. SPUR Coprocessor Interface Signals - Read/Write Timing.

	CLOCK EDGE of CYCLE							
SIGNAL	phi1:LE	phi1:TE	phi2:LE	phi2:TE	phi3:LE	phi3:TE	phi4:LE	phi4:TE
fpuOPCODE_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuRS1_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuRS2_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuRD_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuNewInstr_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuSuspend_CV4	- -	- -	- -	- -	- -	- -	C(cpu) -	S L(fpu)
fpuBusy_C4	- -	- L(cpu)	- -	- -	- -	- -	C(fpu) -	S -
fpuExcept_C4	- -	- L(cpu)	- -	- -	- -	- -	C(fpu) -	S -
fpuBrT_F_C4	- -	- L(cpu)	- -	- -	- -	- -	C(fpu) -	S -
dataMayBeValid_V3 *(pulse)*	- - -	- - -	- - -	- - -	C(cc) - -	S L(cpu) L(fpu)	- - -	- - -
dataIsValid_V3 *(pulse)*	- - -	- - -	- - -	- - -	C(cc) - -	S L(cpu) L(fpu)	- - -	- - -
procTagMatch_V3 *(pulse)*	- - -	- - -	- - -	- - -	C(cc) - -	S L(cpu) L(fpu)	- - -	- - -
interrupts	-	L(cpu)	-	-	-	-	-	-

All signals are levels, except as indicated.
All signals shown "-" are assumed stable or unasserted at those times.

phiN	clock phase N.
LE	leading edge.
TE	trailing edge.
_L	signal is asserted low. All other signals are asserted high by default.
_CN	signal changes only during phi N, where N = 1, 2, 3, or 4.
_VN	signal is valid only during phi N, where N = 1, 2, 3, or 4.
_CVN	signal changes only during phi N and is valid by the end of phi N, where N = 1, 2, 3, or 4.
_N	signal has valid and non-zero value only during phi N, where N = 1, 2, 3, or 4.
cpu	SPUR central processing unit
fpu	SPUR floating-point unit
cc	SPUR cache control unit
C(xxx)	the phase-edge where the sender (CPU or FPU) begins CHANGING the signal.
S	the phase-edge where the signal is first assumed STABLE.
L(xxx)	the phase-edge where the receiver (CPU or FPU) LATCHES the signal.

for all FPU operations.

2.7. Floating-point Unit Micro-Architecture

The description of the internal architecture and structure of the floating-point unit is beyond the scope of this report, and is discussed in [Kat85]. The following sections describe in detail the interaction between the CPU and FPU during normal processing, exception and interrupt handling, and parallel operation.

3. FLOATING-POINT COPROCESSOR INTERFACE DETAILS

3.1. Sending Instructions to Coprocessors

As illustrated in Figure 2, 22 bits of every instruction the CPU fetches from its internal instruction buffer is broadcasted to all coprocessors via *fpuOPCODE_CV3, fpuRS1_CV3, fpuRS2_CV3*, and *fpuRS3*. The coprocessor decodes every instruction issued by the CPU to determine if the instruction is intended for it. This is an *instruction tracker* paradigm. The FPU instructions (see Table 1 for details) include:

(1) FPU arithmetic operations (add, subtract, multiply, divide, etc.) and compare,

(2) FPU memory and register operations (move between registers, or load/store FPU registers from/to memory), and

(3) a synchronization instruction.

As mentioned earlier, the CPU and FPU are both load/store architectures. All floating-point arithmetic operations involve three registers: two sources and one destination. The CPU pipeline consists of four stages each of which has four phases, corresponding to the 4-phase clock. Figure 4 illustrates the cycle overlap between CPU pipeline stages. The duration of each instruction is represented by a horizontal bar. The darker vertical bars splitting the horizontal bar show the cycle boundaries and the smaller/lighter vertical bars show the phase boundaries. Time progresses from left to right.

A typical CPU instruction is composed of the (1) instruction fetch cycle, (2) execution cycle (ALU operation, shift, effective address calculation, etc.), (3) memory reference cycle (or null cycle for register operations), and (4) register write cycle. An FPU instruction is similar, differing only in that it can have more than one execution stage.

3.2. Timing Analysis: Load & Store, FPU Operation, FPU Compare

The following sections discuss and illustrate with diagrams the basic timing relationships between the CPU and FPU for some of the instructions shown in Table 1, and clarify the operation of the CPU-FPU interaction at the hardware level. The simplest functions include loading and storing operands to and from the FPU. The more complicated floating-point operations include the algebraic functions (+,-,*,/), operand conversion, and comparison. Many complicated situations arise due to cache misses, floating-point exceptions, overlapped operation, pipeline suspension, etc. and these will be discussed in Section 3.4. In this section, only simple cases are considered.

3.2.1. Timing of CPU and FPU Load and Store Instructions

The sequence of events during a CPU load (with no data cache miss) are as follows:

cycle 1 (Ifet): CPU fetches the instruction.

cycle 2 (Exec): CPU calculates the effective address and sends it out during phi4.

cycle 3 (Mem): CPU latches the data input pins at the end of phi3.

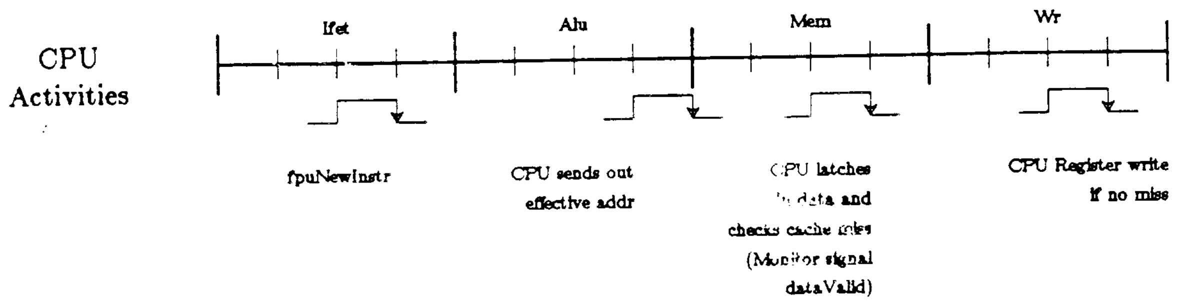
cycle 4 (Wr): CPU writes the operand to the CPU register file during phi3.

This is illustrated in Figure 5a. Figure 5b shows the timing of the FPU load instruction. The only difference between this and the regular CPU load (Figure 5a) is that the FPU latches in the data instead of the CPU.

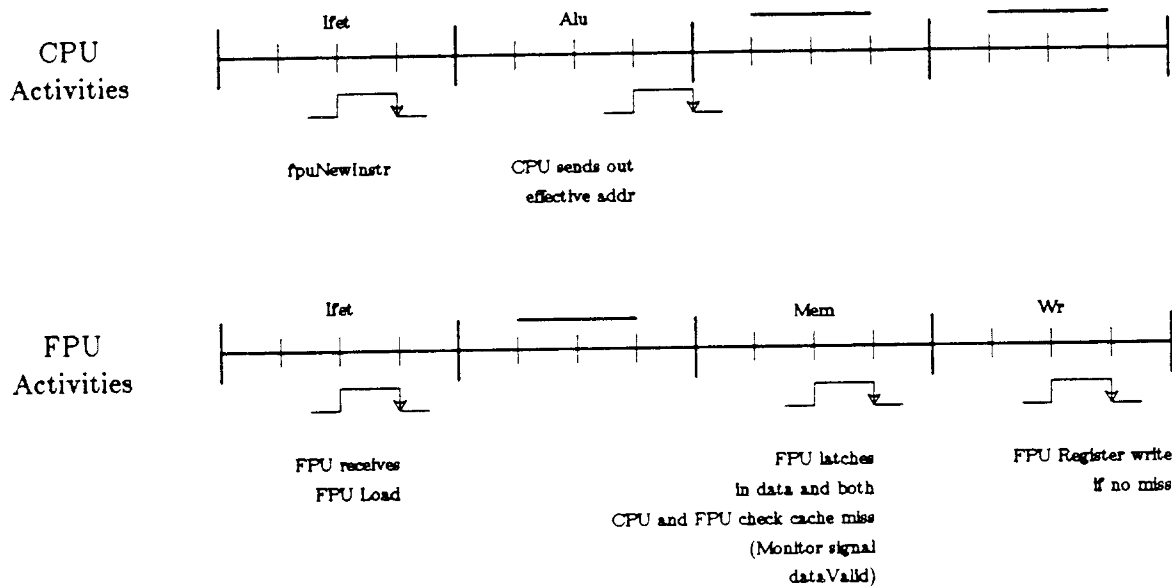
Similarly the only obvious difference between a regular CPU store (Figure 6a) and the FPU store (Figure 6b) is that the FPU sends out the data instead of the CPU. The goal here is to make the FPU behavior during an FPU load/store identical to the CPU behavior during a CPU load/store: receive input data and send output data during the same times the CPU would receive or send data. This essentially makes the FPU transparent to the cache controller. If there is no cache miss, the FPU load and store instructions are both four cycle instructions - the same as all other CPU instructions. The dual port register file on the FPU allows FPU loads and stores to be overlapped with other instructions (the same as CPU instructions). As mentioned previously, there is no internal forwarding on the FPU, and results for an FPU load cannot be used until the third instruction after the load.

3.2.2. Timing of FPU Operation Instructions

The timing of an FPU operation that is neither an FPU load nor an FPU store instruction is as follows (for example, a 4-cycle FADD):



Part a. CPU LOAD



Part b. FPU LOAD

Figure 5. CPU and FPU load instruction timing.

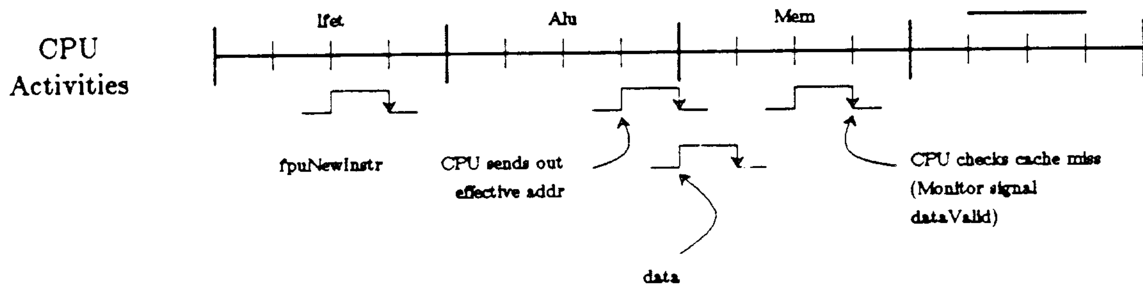
cycle 1 (Ifet): CPU fetches the instruction, FPU latches it.

cycle 2 (Exec1): FPU begins operation, asserts *fpuBusy_C4* at beginning of phi4.

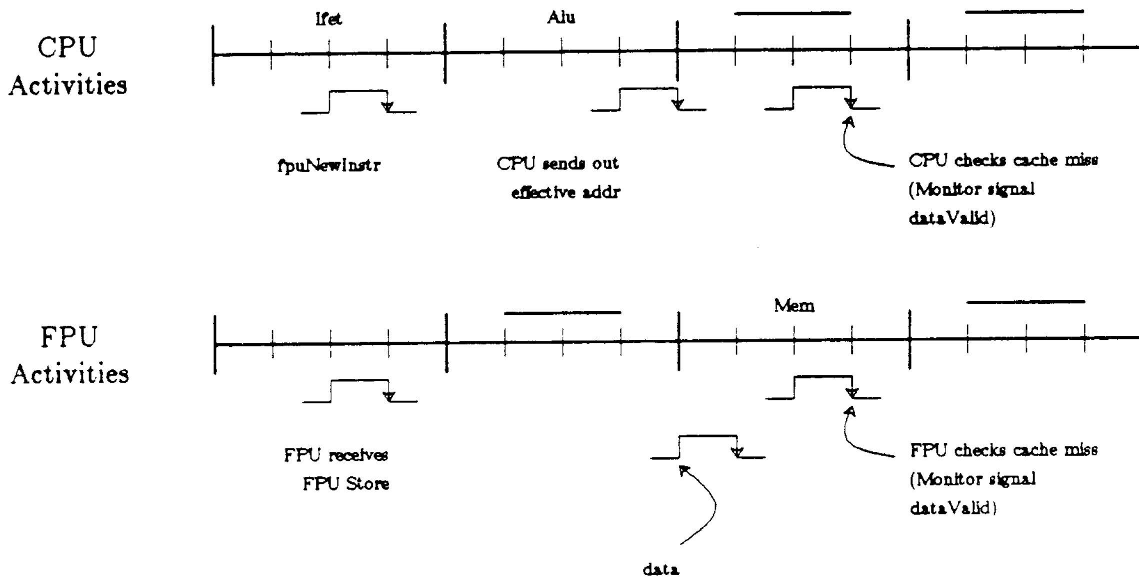
cycle 3 (Exec2): FPU continues operation, CPU updates *FpuPC* during phi1.

cycle 4 (Exec3): FPU continues operation.

cycle 5 (Wr): FPU ends operation, writes FPU register file during phi3, disasserts *fpuBusy_C4* at beginning of phi4, and asserts *fpuExcep* in phi4 if such a condition occurs.



Part a. CPU Store



Part b. FPU Store

Figure 6. CPU and FPU Store instruction timing.

This is illustrated in Figure 7, with back to back floating-point multiply instructions. If the FPU operation had been FADD or FDIV, the only difference would have been fewer (3 total for FADD) or more (20 total for FDIV) Exec cycles before the Wr(FPU) cycle.

As shown in the timing diagram, the CPU recognizes that the FPU is busy only after the CPU has already issued the instructions I0 and I1. † The CPU internal pipeline is then suspended and the FPU interface control unit has

† From the view point of the FPU, I2 has not yet been issued because *fpuNewInstr* is zero during the CPU Ifet cycle.

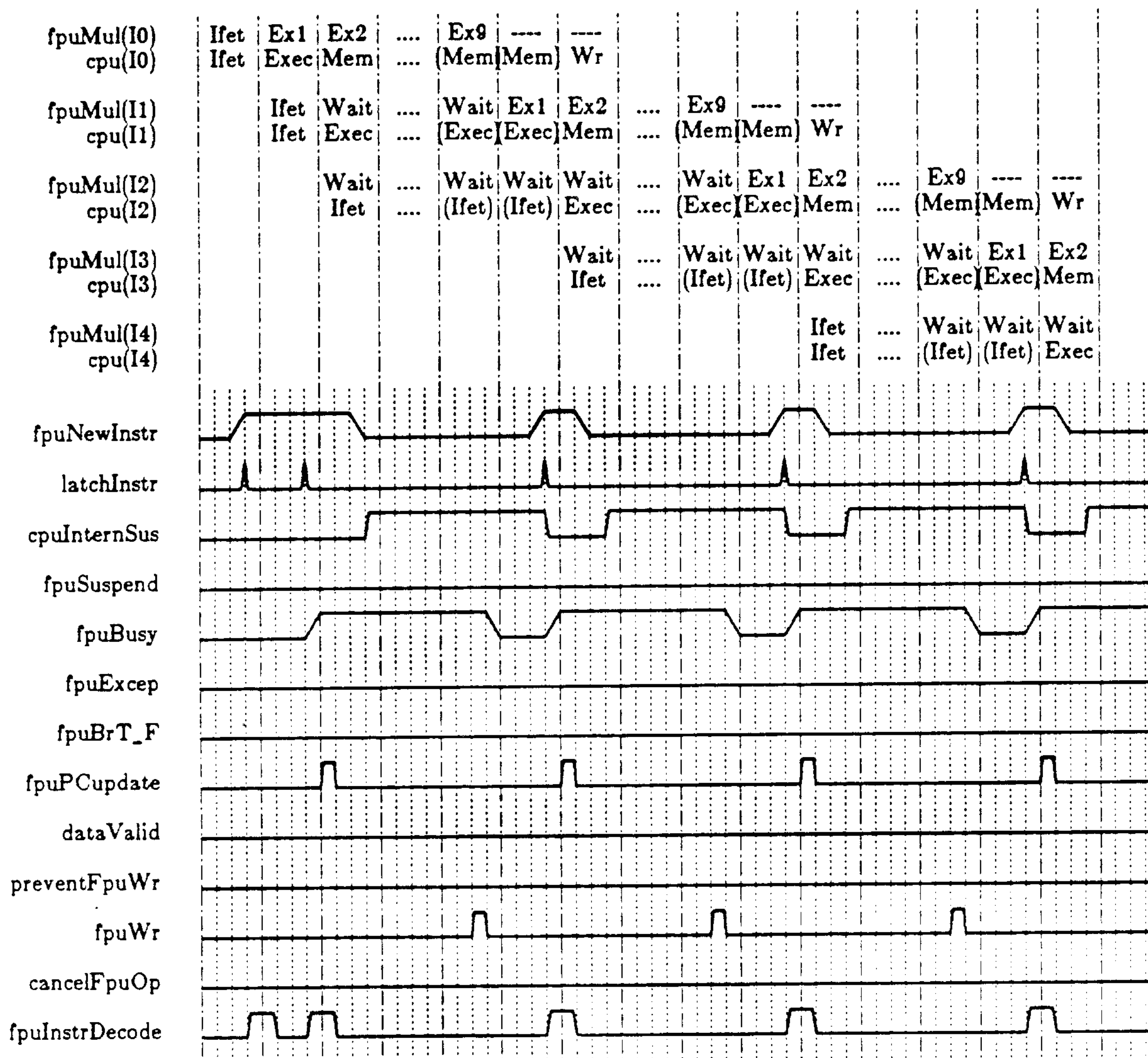


Figure 7. FPU operation instruction timing for back-to-back FPU Operations

Note: Signals that are shown to make the transition from low to high or vice-versa in less than one phase time are intra-chip signals. Those shown taking one phase time are inter-chip signals.

buffered instruction I1 while the FPU execution unit is working on I0. The CPU updates the *FpuPC* register to instruction I0 during the Mem cycle (Exec2) for I0. The CPU internal pipeline remains suspended until one full cycle after the *fpuBusy_C4* line is disasserted. Before that, however, the *fpuNewInstr_CV3* is issued so that the FPU will latch the *fpuOPCODE_CV3* and register specifier lines for I2.

The sequence is the same for all other instructions:

(1) The *fpuBusy_C4* signal is asserted at the beginning of a new execution sequence on the FPU.

(2) The CPU suspends its internal pipeline, not issuing any more instructions until the FPU is no longer busy.

(3) The CPU updates the *FpuPC* register during its Mem cycle (Exec2 for the FPU).

(4) The FPU register file is written during phi3 of the last execution cycle, and the *fpuBusy_C4* signal is disasserted in phi4.

(5) The values on the *fpuOPCODE_CV3*, *fpuRS1_CV3*, *fpuRS2_CV3*, and *fpuRD_CV3* lines are significant only during phi3 of any cycle while *fpuNewInstr_CV3* is asserted.

3.2.3. Timing of the FPU Compare Instruction

The FPU compare instruction's timing is similar to that of FADD and is illustrated in Figure 8.

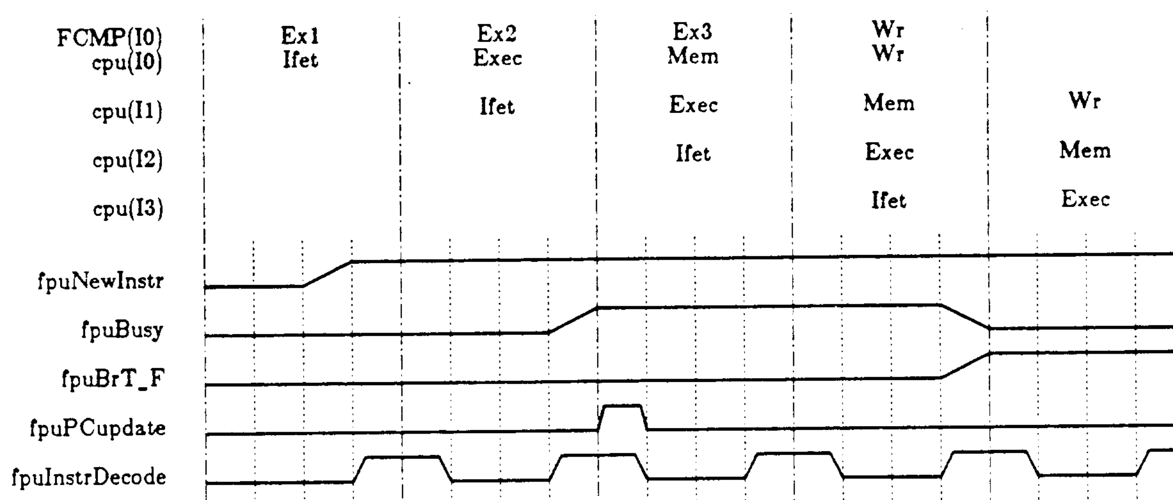


Figure 8. FPU compare instruction timing

Like any other FPU operation, the *fpuBusy_C4* signal is asserted until the comparison is completed and the result (T or F) is stored in a bit in the Fpsw. This bit is continuously available to the CPU as the *fpuBrT_F* signal. The CPU looks at the *fpuBrT_F* signal whenever a regular CMP_BR_DELAYED instruction is executed with the *fpuBrT_F* bit specified by the COND field of the instruction. The compiler must insert a SYNC instruction (or several CPU instructions) after the FCMP instruction but before the *fpuBrT_F* signal is tested to guarantee that the *fpuBrT_F* signal is valid, and allow possible resulting exceptions to be

handled. As mentioned before, the *fpuBusy_C4* signal provides an indication to the CPU when the FPU execution unit is busy.

3.3. FPU Suspension and Parallel CPU-FPU Operation

The CPU is designed to issue a new instruction each cycle. However, the CPU pipeline may occasionally be suspended, resulting in no new instructions being issued. From the FPU's point of view, there are two possible causes for pipeline suspension:

(1) Reasons that are related to FPU operation. When the FPU is busy with a previous arithmetic operation, the CPU must wait until it becomes free before it can issue a new FPU instruction (as discussed in Section 3.2.2.

(2) Reasons that are NOT related to FPU operation. For example, this happens when a cache miss occurs.

Also, the CPU and the FPU have been designed to allow parallel operation. The following sections explain in greater detail the timing relationships between the CPU and FPU to make parallel operation possible.

3.3.1. CPU and FPU Pipeline Suspension

Figure 7 illustrated the internal pipeline suspension of the CPU due to *fpuBusy_C4* being asserted (reason 1 stated above). The CPU issued an FPU instruction in I0 and wanted to issue another FPU instruction immediately afterward. However, the FPU execution unit can only handle one instruction at a time. The CPU responds to the *fpuBusy_C4* signal when another FPU instruction is decoded by suspending the CPU pipeline. This results in no new instructions being issued. It is important to note that the FPU operation cannot be suspended during such a condition. Otherwise, a deadlock may result (since *fpuBusy_C4* would never be disasserted and the CPU would wait forever).

On the other hand, the CPU internal pipeline can be suspended for reasons not related to the FPU operation. At the interface level, the only indication that the CPU internal pipeline is suspended comes when the *fpuSuspend_CV4* signal is asserted. In this case, certain FPU activities must be suspended to prevent it from advancing to an inconsistent state if a trap occurs. (By definition, all instructions issued before the trapping instruction must be allowed to finish, while the victim of the trap and all later instructions must be restarted after the trap is serviced.) This implies the FPU operation can be suspended by the CPU with the *fpuSuspend_CV4* signal. (Note: The FPU does not have to suspend everything as soon as it receives the *fpuSuspend_CV4* signal. It must simply remain in a state where instructions that were issued after the instruction which eventually causes the trap can be killed if a trap occurs, and before writing to FPU internal

registers.) This is illustrated in Figure 9, which shows that the CPU asserts the *fpuSuspend_CV4* signal at the beginning of phi4 before the first suspended cycle and disasserts the *fpuSuspend_CV4* signal at the beginning of phi4 before the first normal cycle after suspension.

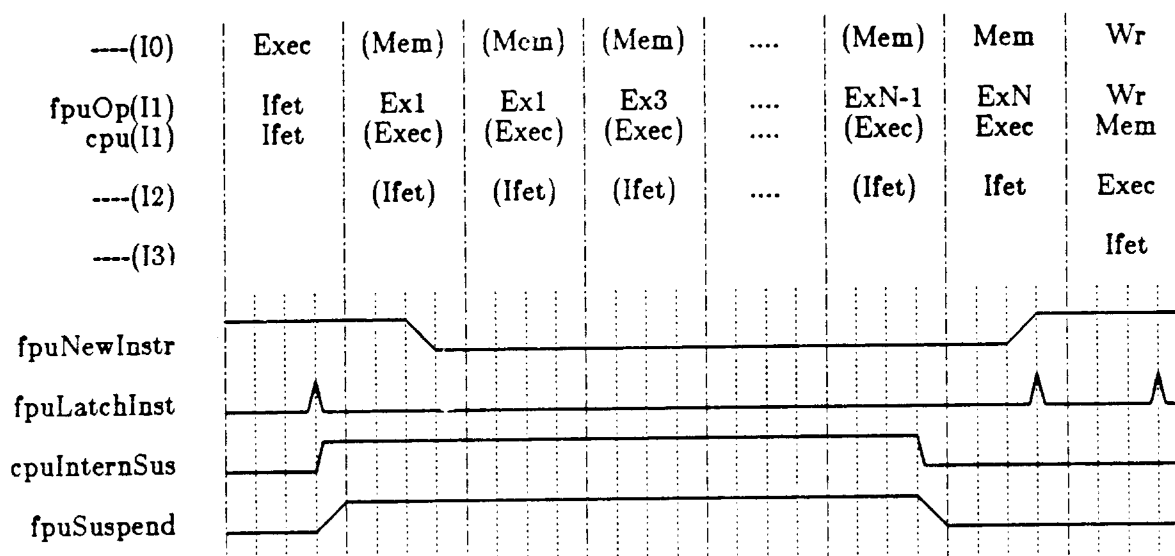


Figure 9. Illustration of fpuSuspend_CV4 signal.

In Figure 9, the cycles are identified for both the FPU and the CPU, as indicated by Ifet, Exec (or ExN, for the FPU), Mem or Wr. Also, depending on whether I1 is a CPU instruction or an FPU instruction, the Exec cycle of I1 can either be in the CPU or the FPU.

To summarize: the CPU asserts the signal *fpuNewInstr_CV3* to inform the FPU that new values for *fpuOPCODE_CV3*, *fpuRS1_CV3*, *fpuRS2_CV3*, and *fpuRS3* are available and the FPU latches them at the end of phi3. During CPU pipeline suspension, no new instructions are issued to the FPU and the signal *fpuNewInstr_CV3* is unasserted (zero) as illustrated in Figure 9.

3.3.2. CPU and FPU Parallel Operation

The CPU and FPU can function in two different modes: parallel and sequential. In parallel mode, the *fpuParallel* bit in the CPU's Upsw is set and parallel operations of the CPU and FPU are allowed. Otherwise, all FPU operation instructions are serialized with CPU instructions. As mentioned earlier, the FPU is a load/store architecture and the instructions include: FPU arithmetic operations (add, subtract, multiply, divide, compare, etc.) and FPU memory and register operations (move between registers, or load/store FPU registers from/to memory). The parallel operation of the CPU and FPU can be summarized as

follows:

(1) After an FPU memory operation is issued, the CPU and FPU can continue to execute either CPU or FPU instructions. (Note: Software must guarantee that the target of an FPU operation is valid before an FPU store instruction tries to write it to memory. Otherwise, a SYNC instruction must be inserted before all FPU store instructions to make sure that any operation which could potentially change the store target is completed.)

(2) After an FPU arithmetic operation is issued, the CPU can continue to execute CPU instructions, or FPU load or FPU store instructions. No new FPU operation instruction is allowed to begin until the previous FPU operation instruction is done.

If the *fpuParallel* bit is not set, the FPU and CPU operate in sequential mode in which all FPU operation instructions must be executed serial fashion; ie, after the CPU issues an FPU operation instruction, the CPU cannot issue or continue to execute *any* instructions until the FPU instruction is done (*fpuBusy_C4* is unasserted). Figure 8 in Section 3.2.3 illustrates parallel execution. While the FPU performs the compare operation of I0, the CPU continues to execute I1, I2, and I3 where I1, I2, and I3 are assumed to be CPU instructions. Figure 10 illustrates the case where FPU and CPU instructions are mixed in sequence, and the operation is assumed to be sequential.

3.4. How Special Cases Are Handled By This Interface

Many things can happen during the normal execution of programs that need special attention at the hardware level to guarantee correct results. Among them are exceptional conditions (associated with integer or floating-point arithmetic), cache misses, page and bus faults, etc. The following sections of this report explain many of the most common "unusual" events and illustrate how they are handled.

3.4.1. The Effects of Cache Misses on CPU and FPU Pipelines

A cache reference immediately followed by another cache reference (for any combination of CPU or FPU LD/ST instructions) can be handled by the SPUR system as long as both the CPU and FPU monitor the *dataValid* signal (a composite of 3 other signals) and the FPU monitors the *fpuSuspend_CV4* signal.

3.4.1.1. CPU Load Followed By FPU Load - No Cache Miss

Figure 11 shows the simplest case where neither instruction I0 nor I1 causes a cache miss. The FPU will not be misled by the first assertion of the *dataValid* signal as long as the FPU knows when to start checking the *dataValid* signal. This

	1	2	3	4	5	6	7	8	9	10	11	12	13
fpuAdd(I0)	Ifet	Ex1	Ex2	Ex3	Wr	----	----						
cpu(I0)	Ifet	Exec	Mem	(Mem)	(Mem)	(Mem)	Wr						
fpu(I1)		Ifet	----	----	----	----	----	----					
cpuADD(I1)		Ifet	Exec	(Exec)	(Exec)	(Exec)	Mem	Wr					
fpu(I2)			Ifet	----	----	----	----	----	----				
cpuADD(I2)			Ifet	(Ifet)	(Ifet)	(Ifet)	Exec	Mem	Wr				
fpuADD(I3)							Ifet	Ex1	Ex2	Ex3	Wr		
cpu(I3)							Ifet	Exec	Mem	(Mem)	(Mem)	(Mem)	Wr
fpu(I4)								Ifet	----	----	----	----	----
cpuADD(I4)								Ifet	Exec	(Exec)	(Exec)	(Exec)	Mem
fpu(I5)									Ifet	----	----	----	----
cpuADD(I5)									Ifet	(Ifet)	(Ifet)	(Ifet)	Exec

Figure 10. Sequential mode: FPU operation followed by CPU operation

is a requirement even for the simplest case when an FPU load or store instruction is not preceded by any other cache reference instruction. As indicated in Figure 5 and Figure 6, without any pending cache miss, the FPU does not start checking the *dataValid* signal until its Mem cycle, which is two cycles (third cycle overall) after it receives the FPU load or store instruction.

3.4.1.2. FPU Load Followed By FPU Load - Cache Miss

For all memory reference instruction (either CPU or FPU load or store), signals *dataMayBeValid_V3*, *procTagMatch_V3*, and *dataIsValid_V3* must be monitored by both the CPU and FPU to form the *dataValid* signal (see Figure 2). This is necessary because a cache miss caused by an FPU load or store requires: (1) the CPU to suspend its pipeline, and (2) the FPU to repeat its memory access cycle and suspend all FPU activities that are related to FPU instructions received after the FPU memory access instruction. This is illustrated in Figure 12, which shows an FPU load that results in a cache miss, and is followed by another FPU load instruction.

As stated above, the FPU must suspend all FPU activities that are related to FPU instructions received after the FPU memory access instruction, until the

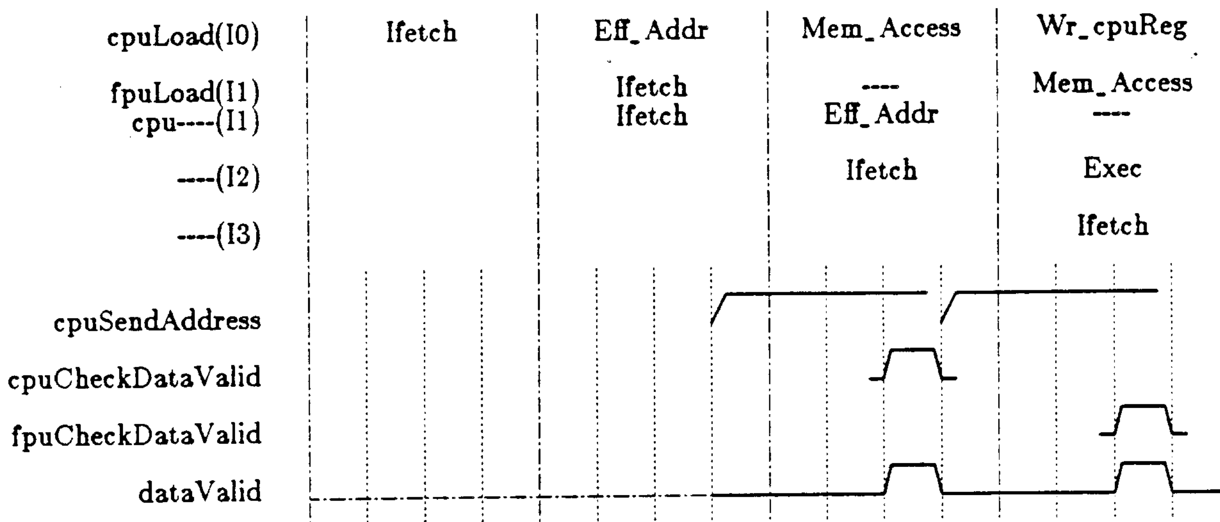


Figure 11. Timing for CPU load followed by FPU load or store.

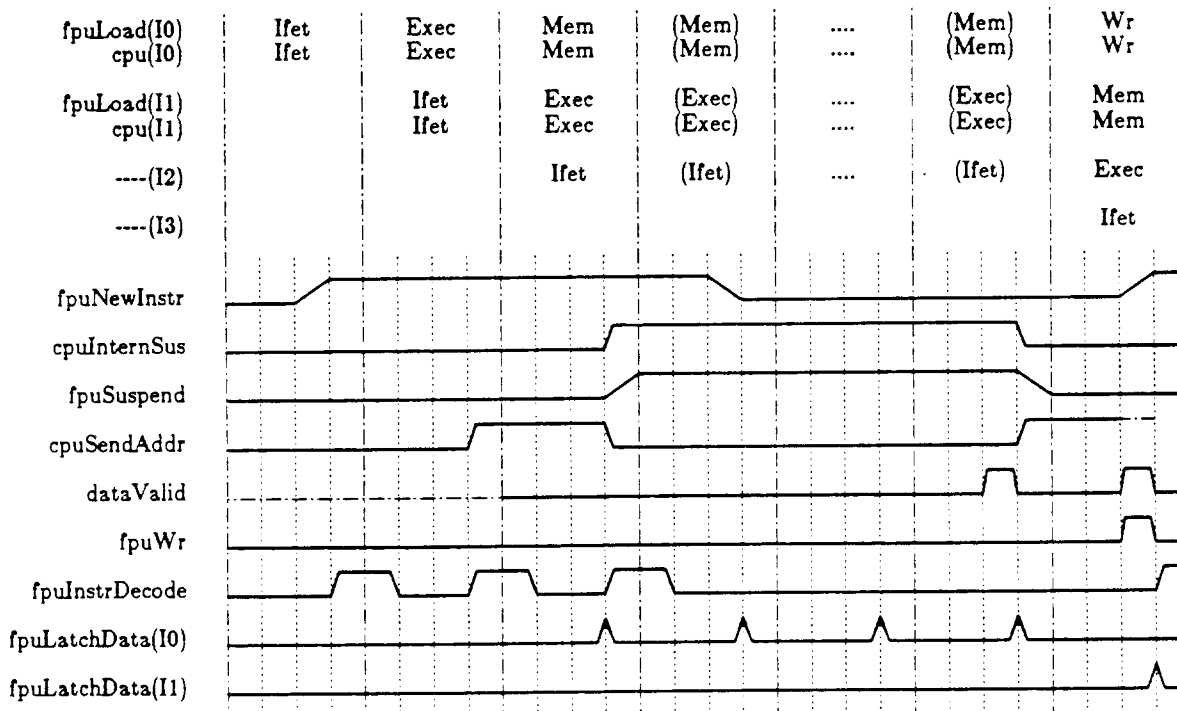


Figure 12. FPU cache miss followed by FPU load or store

pending cache miss is serviced. For example in Figure 12, I1 (the second FPU load instruction, is suspended until valid data is received for I0. So, the *fpuSuspend_CV4* signal is asserted. However, all FPU instructions issued prior to the time I0 was fetched can continue execution. The signal *fpuBusy_C4* remains unasserted and FPU load and store are handled like CPU load and store. Although both the CPU and FPU see the *dataValid* signal, only the CPU has to monitor the *fault* or *error* lines. This will be explained later in this section.

3.4.1.3. CPU Load Followed By FPU Load - Cache Miss

The coprocessor interface also takes care of what might appear to be a much more complicated case: a CPU memory reference instruction that misses in the cache followed by an FPU memory reference instruction.

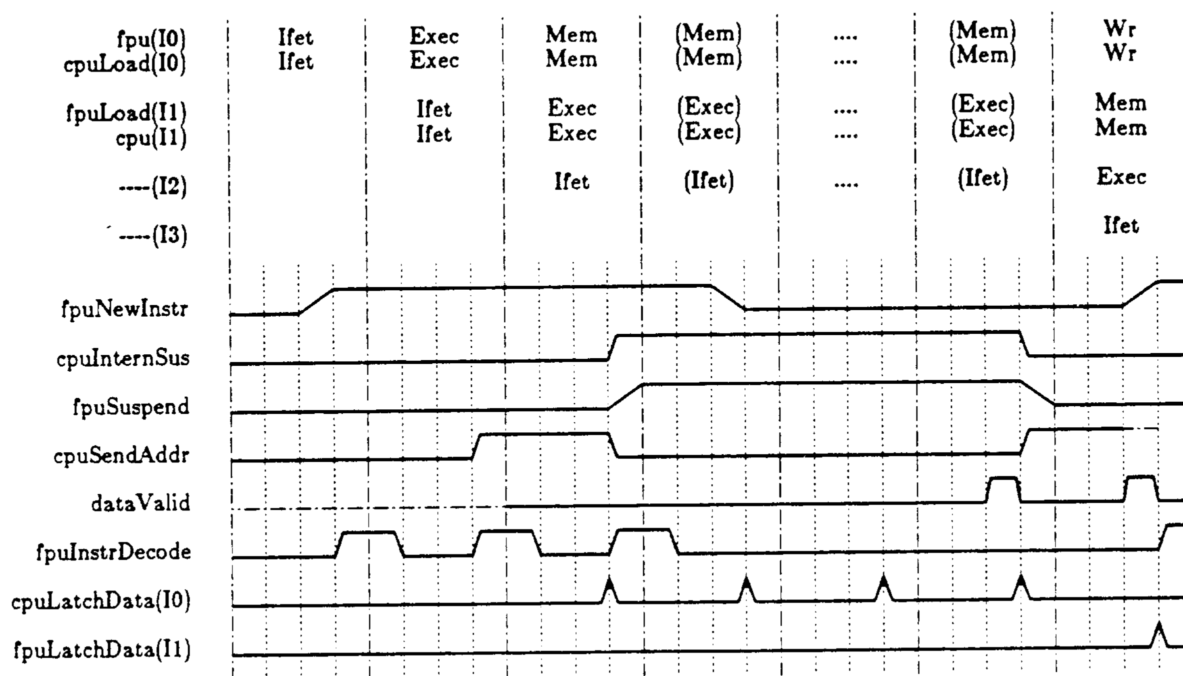


Figure 13. CPU cache miss followed by FPU load or store

The FPU cannot start looking at the *dataValid* line for its data until the cache miss caused by the CPU load has been satisfied. This problem is easily solved by using the *fpuSuspend_CV4* signal. The CPU pipeline is suspended due to the cache miss caused by I0. And the FPU will not begin looking for the *dataValid* signal until *fpuSuspend_CV4* is disasserted.

By comparing I1 in Figure 11 and I1 in Figure 13, it is obvious that the *fpuSuspend_CV4* signal stops the FPU from getting into I1's memory access cycle (Mem) until the pending cache miss for I0 is serviced. Since I1 is not in its memory access cycle, the FPU does not look at the *dataValid* signal and will not

be misled by the first assertion of the *dataValid* signal.

Finally, if I1 in Figure 13 causes a cache miss, the CPU pipeline will again be suspended. The FPU controller must repeat the memory access cycle of I1 until valid data is received (*dataValid* = 1) and suspend state-changing activities in the FPU. By using the *fpuSuspend_CV4* signal, the FPU does not have to understand the CPU load or store instruction. Also, the FPU does not have to match each assertion of *dataValid* with each cache reference.

In summary:

(1) If *fpuSuspend_CV4* is asserted, suspend FPU execution unit activities. (The FPU does not have to suspend everything as soon as it receives the suspend signal, but must guarantee that FPU operations that are in progress can be killed if a trap occurs (see Section 3.6) before the internal state of the FPU is changed). Otherwise,

(2) Start looking for valid data by monitoring the *dataValid* signal during the memory access cycle of an FPU load or store.

(3) If *dataValid* is not asserted, repeat the memory access cycle and suspend all FPU activities that are related to the FPU instruction received after the FPU memory access instruction until *dataValid* is asserted.

3.4.1.4. Cache Miss Resulting in Fault

Certain error conditions in the CPU and FPU result in a trap. Some of these have been mentioned previously. The FPU finds out the CPU is taking a trap by decoding the CPU internal TRAP_CALL instruction. This section describes what the FPU must do in response to receiving the TRAP_CALL instruction.

Inside the CPU, there is a slight difference between a trap caused by a page fault or bus fault and a trap caused by something else: a page fault or bus fault causes a trap during CPU pipeline suspension while the other conditions can only cause a trap when the CPU pipeline is not suspended. How the FPU should respond to a TRAP_CALL depends on whether the page or bus fault is caused by an FPU cache access, or by a CPU cache access. Therefore, there are three different cases to consider:

(1) The CPU takes a trap because the CPU cache reference results in a page or bus fault,

(2) The CPU takes a trap because the FPU cache reference results in a page or bus fault,

(3) The CPU takes a trap for reasons other than 1 or 2 above. This will be referred to as "regular trap" for the rest of this section.

Case 3 described above is the simplest and is illustrated in Figure 14. When the FPU receives the internal instruction TRAP_CALL, all it has to do is check

whether the last instruction it received (I1 in Figure 14) before receiving TRAP_CALL was an FPU instruction. If that is the case, then that instruction must be killed. Otherwise, no action has to be taken.

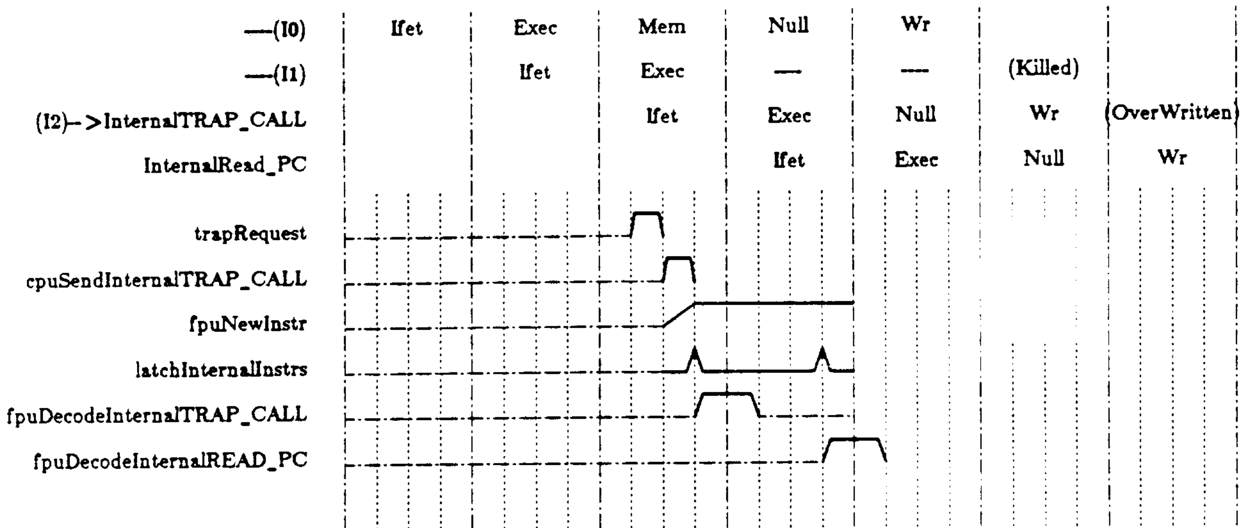


Figure 14. A regular TRAP timing sequence.

3.4.1.4.1. CPU Cache Miss Fault

Figure 15 illustrates Case 1 in which the CPU takes a trap because a CPU cache reference results in a page or bus fault. As discussed in Section 3.3, whenever the CPU pipeline is suspended for a CPU instruction, FPU operation is also suspended for instructions issued after the CPU instruction that caused the suspension. (FPU operations that were started before the CPU instruction causing the CPU suspension must finish, and the *fpuBusy_C4* signal will guarantee that no other FPU operations will be started.) In other words, during CPU pipeline suspension, no later FPU instruction can complete its execution even if the CPU-FPU pair operates in parallel mode. I1, which is shown in Figure 15 as an FPU instruction, will not finish its execution during the CPU pipeline suspension and can be easily killed when the FPU receives the TRAP_CALL instruction. Any FPU instruction received before I0, however, must be allowed to finish.

Notice that in Figure 15 the *fpuSuspend_CV4* signal is still asserted when the FPU receives the TRAP_CALL instruction. As discussed previously, as long as *fpuSuspend_CV4* is asserted, the Ifet cycle of the I2 is being repeated by the FPU. Consequently, when the FPU receives the TRAP_CALL, the FPU still considers I2 to be in its Ifet cycle. As far as the FPU is concerned, instruction I2 is overwritten by the TRAP_CALL instruction. The last instruction the FPU received before receiving the TRAP_CALL is, in effect, still I1. Therefore the

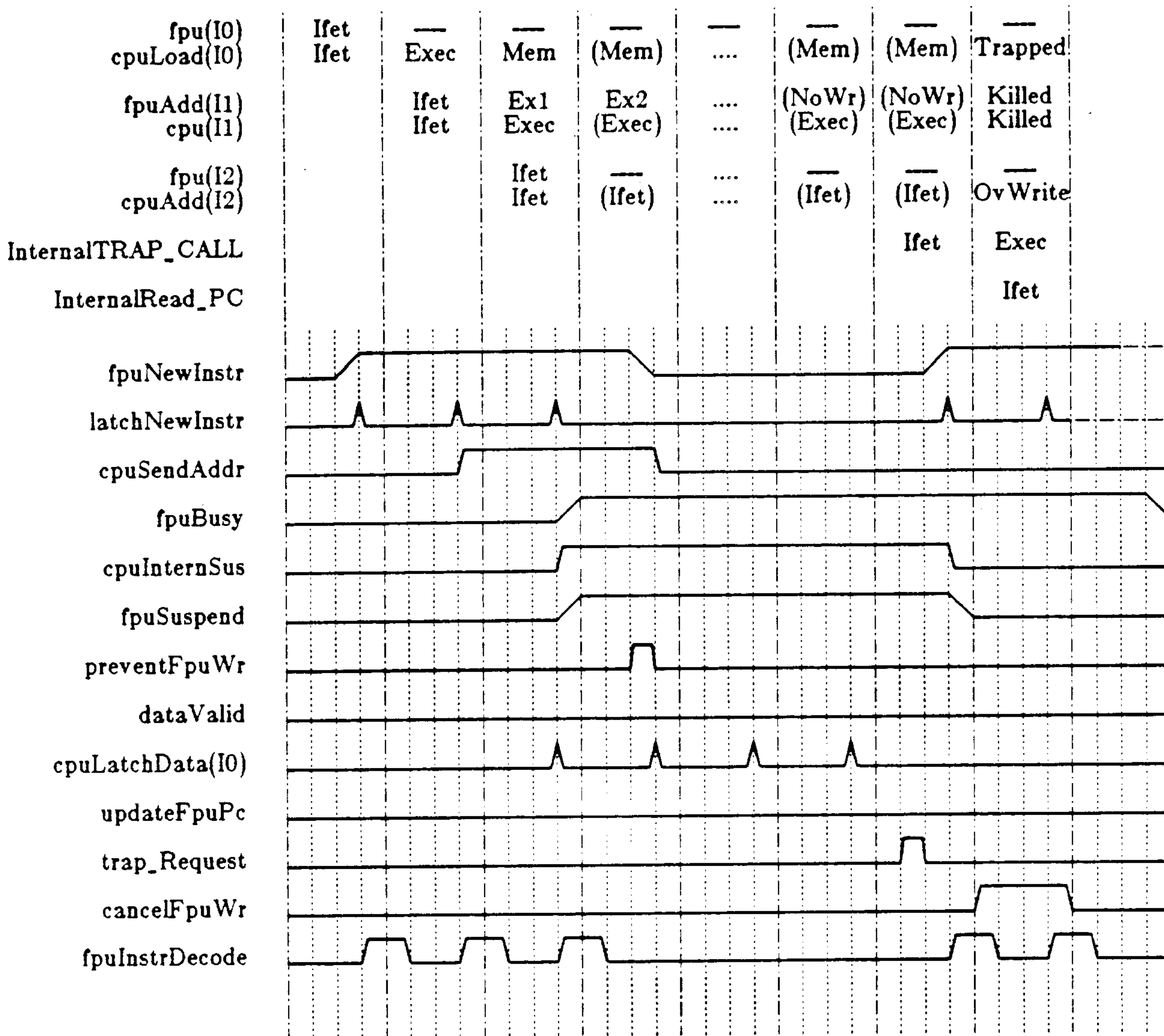


Figure 15. Timing of trap caused by CPU page or bus fault.

same hardware, which is used in Case 3 to kill I1 in Figure 14, can be used here for Case 1 as long as the hardware ignores the empty cycles when the FPU is suspended by the *fpuSuspend_CV4* signal.

3.4.1.4.2. FPU Cache Miss Fault

Figure 16 illustrates Case 2 in which the CPU takes a trap because the FPU cache reference results in a page or bus fault. The CPU pipeline is suspended due to the cache miss, and the FPU must suspend all execution activities related to this instruction (no effect on previously issued instructions). Consequently, during an FPU cache miss, no FPU instruction issued after I0 can complete its execution even if the CPU-FPU pair operates in parallel mode. Thus, the FPU instruction

I1 shown in Figure 16 will not finish its execution (ie, write its result to the destination register) during an FPU cache miss and can be killed easily when the FPU receives the TRAP_CALL instruction. The FPU Exec cycles during which the destination register would be written are usually shown at *Wr* in all figures. If the FPU write is cancelled, it is designated a *NoWr(FPU)* cycle.

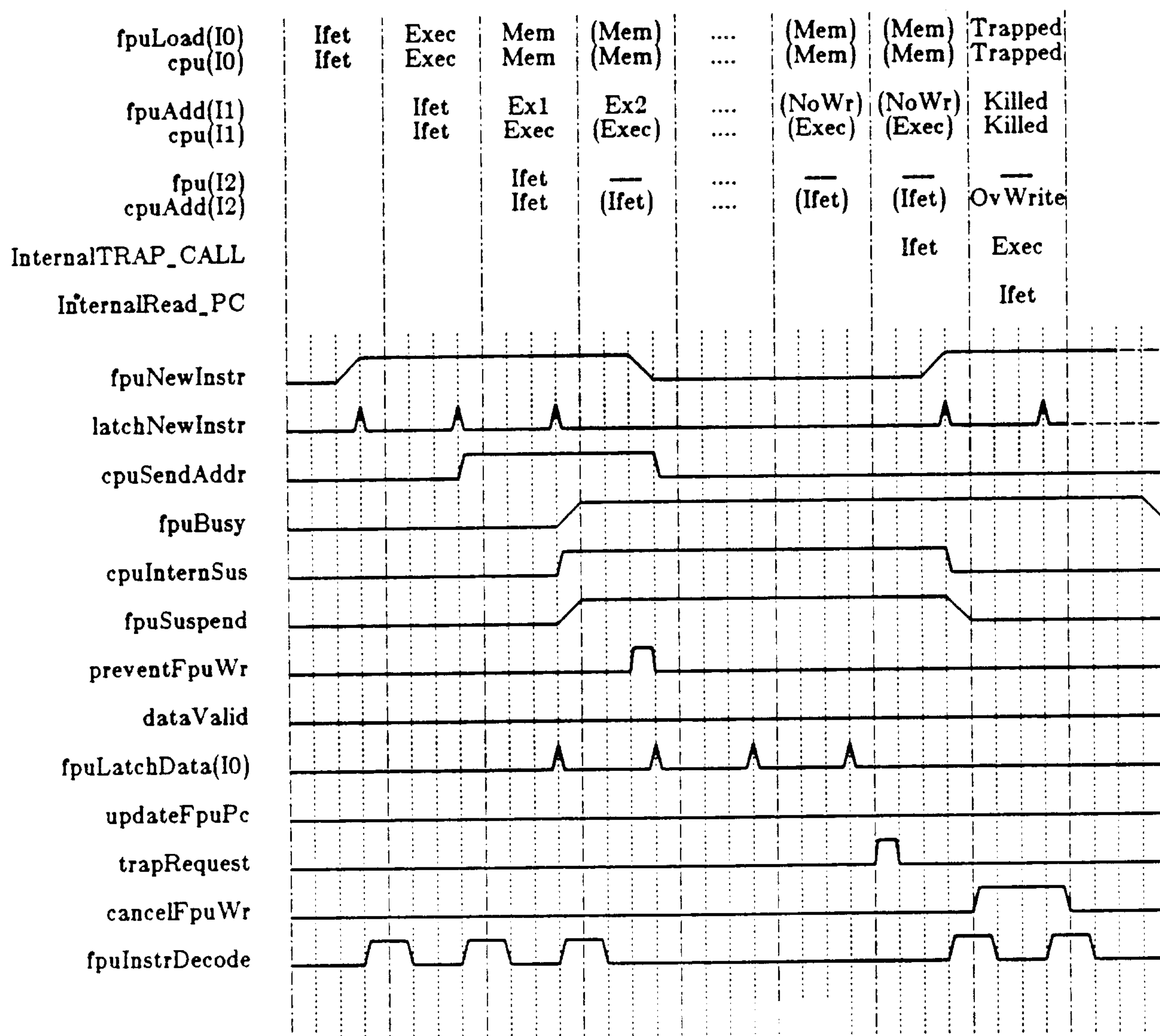


Figure 16. Timing of trap caused by FPU page or bus fault.

Since the FPU is not looking at the page fault or bus fault line, it does not know a fault has occurred until the FPU receives the TRAP_CALL instruction. Therefore in Figure 16, when the FPU receives the TRAP_CALL instruction, the FPU still considers I2 to be in its Ifet cycle. Once again, the TRAP_CALL instruction has overwritten instruction I2 and I1 is still, in effect, the last

instruction the FPU received before receiving the TRAP_CALL. Therefore the same hardware, which is used Case 1 to kill I1 in Figure 15, can be used here for Case 2 as long as the hardware ignores the empty cycles when most of the FPU activities are suspended due to an FPU cache miss.

3.5. Internal MISS Instruction

Besides the internal TRAP_CALL instruction, the FPU must also understand the internal MISS instruction, used for instruction buffer misses. The FPU should not do anything as a result of receiving this instruction. In other words, the FPU must treat this instruction the same way it treats all others CPU (integer) instructions.

3.6. Exception Handling

Figure 17 shows a most critical case, where two FPU instructions are in series and the first one causes an arithmetic exception. Here, the CPU must respond to the exception before the CPU updates the FpuPC incorrectly. When the CPU responds to the exception by taking a trap, the FpuPC must contain the address of I0, the FPU operation instruction that caused the exception.

Figure 17 should be compared with Figure 7, which shows the case where several FPU instructions are in series but none causes an exception. The following two assumptions are made in Figure 17:

(1) The CPU looks at the *fpuExcep* continuously.

(2) The CPU, if no exception has occurred and the *fpuBusy_C4* signal is not asserted, updates the FpuPC during phi1 of the second execution cycle of an FPU instruction (the equivalent Mem cycle for the CPU).

In Figure 17, instruction I1 is killed, as described in the previous examples shown in Figure 15 and Figure 16. I2 is overwritten by the internal TRAP_CALL.

3.6.1. Exceptions During Cache Miss and Subsequent Fault

Since the FPU can operate in parallel mode, an exception resulting from an operation may not be posted until long after the instruction was issued. In the meantime, the CPU can issue an instruction that results in a cache miss, and subsequently a page fault. What happens then?

First, the exception detection logic on the CPU is disabled during suspension due to a cache miss. Thus, if a reference results in a cache miss, the CPU will suspend everything until (1) the reference is either satisfied (the *dataValid* signal is asserted) and the CPU continues operation, or (2) a page fault results. In the first case, once normal operation resumes, the exception will be detected (since the

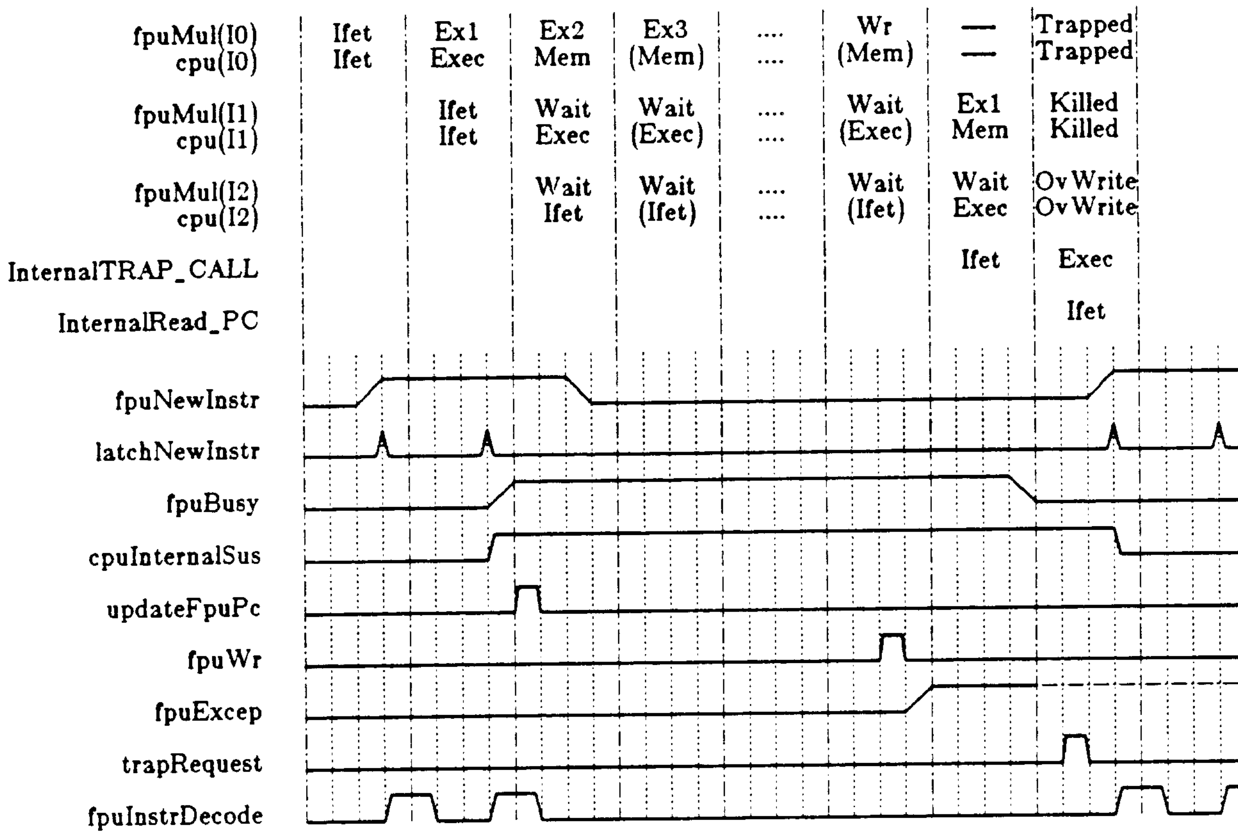


Figure 17. Timing of FPU exception.

logic is no longer disabled because of suspension) and handled much like an interrupt, resulting in a TRAP_CALL. In the second case, the trap handler is called to resolve the page fault first, and any instructions issued after the one that caused the miss/fault are killed. The page-fault trap has higher priority than the FPU exception trap, so the exception will be handled later. (Note: to guarantee that the state of the FPU is preserved to allow proper exception handling, the FPU must either not be used by any routines in the trap handling code, or the exception must be handled during the trap handler.)

Table 4 shows which exceptions are generated by the hardware (FPU execution unit) for each of the FPU instructions. It should be remembered that the fpuPC register maintained on the CPU is for the latest FPU operation. Loads and stores are not "operations" in that sense. Also, since the SYNC instruction is only a busy-test, it falls in the non-operation category. In general, FPU instructions that can cause exceptions are the only ones for which the CPU must save the fpuPC.

Table 4. SPUR FPU Exceptions

INSTRUCTION	EXCEPTION			
	Operand Trap	Result Overflow	Result Underflow	Result Inexact
LD_SGL	no exceptions generated			
LD_DBL	no exceptions generated			
LD_EXT1	no exceptions generated			
LD_EXT2	no exceptions generated			
ST_SGL	no exceptions generated			
ST_DBL	no exceptions generated			
ST_EXT1	no exceptions generated			
ST_EXT2	no exceptions generated			
FMOV	no exceptions generated			
FABS	no exceptions generated			
FNEG	no exceptions generated			
FP_CMP	yes	no	no	no
FADD	yes	yes	yes	yes
FSUB	yes	yes	yes	yes
FMUL	yes	yes	yes	yes
FDIV	yes	yes	yes	yes
CVTS	yes	yes	yes	yes
CVTD	yes	yes	yes	yes
SYNC	NONE			

1. FUTURE COPROCESSORS

Many factors influenced the SPUR coprocessor interface design. Initially, we limited ourselves to approximately 100 signal pins as a packaging constraint. We currently expect about double that number. Some of the things we left out, changed, or limited because of the pins constraint are:

(1) generalized coprocessor ID's. A coprocessor ID code would have been included in the instruction format (or in a CPU control register) to allow more than one coprocessor to exist in a system.

(2) generalized exception handling.

(3) multiple coprocessor support. Coprocessor PC's would need to be saved on the CPU chip for exception handling, etc. Multiple enable bits would need to exist in the Upsw to indicate which coprocessors are present, which can run in parallel, and so forth.

(4) local-bus master capability. Language coprocessors would need to manipulate memory. For example, a Prolog coprocessor (see [BCD86]) that provides hardware support for unification will need to compare entire data structures to determine if two patterns

match. This implies being able to read from and write to memory and being able to handle such things as page faults, etc.

Other applications are being considered for the SPUR workstation environment, involving second-generation capabilities. However, details are beyond the scope of this paper.

2. ACKNOWLEDGMENTS

We are grateful to Glenn Adams, Gaetano Borriello, BK Bose, Reese Faucette, Randy Katz, and Dave Patterson for reading early versions of this paper and providing useful suggestions.

3. REFERENCES

[BCD86] G. Borriello, A. Cherenson, P. Danzig and M. Nelson, "Special or General-Purpose Hardware for Prolog: A Comparison", Computer Science Division Technical Report UCB/Computer Science Dpt. 87/314, University of California, Berkeley, October 1986.

[Fau86] R. Faucette, "SPUR Performance Monitor Coprocessor (PMC)", Computer Science Division Internal Technical Report, University of California, Berkeley, October 1986.

[Han85] P. M. Hansen, "Coprocessor Architectures for VLSI", Phd Qualifying Examination Proposal, University of California, Berkeley, CA 94720, May 3, 1985.

[HEL85] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", Computer Science Division Technical Report UCB/Computer Science Dpt. 86/273, University of California, Berkeley, December 1985.

[Kat85] "SPUR Architecture Design Rationale", *Proc. of CS292i: Implementation of VLSI Systems*, September 1985.

[KEP] R. H. Katz, S. Eggers, C. Perkins, R. Sheldon and D. Wood, "Implementing a Cache Consistency Protocol", 12th Annual Symposium on Computer Architecture, Boston, MA, (June 1985).