Functional Specification and Simulation of a Floating Point Co-Processor for SPUR [1]

Glenn D. Adams

Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

This report describes the internal organization of the SPUR floating point chip. The primary representation of the FPU microarchitecture is its functional level executable hardware description. This description serves as the primary chip design verification tool at both the functional and the layout levels. The text of this paper gives the operation sequence for the chip's instructions and details its datapath and control structures.

1. INTRODUCTION

The SPUR Floating Point Unit is a single chip, tightly coupled co-processor intended for the SPUR multi-processor workstation[HEL85]. This chip boosts the floating point performance of the SPUR architecture by executing the most common floating point operations directly. The supported arithmetic operations include addition, subtraction, multiplication and division on 80 bit operands. These operations require 4, 4, 8 and 20 processor cycles respectively; these cycles may be overlapped with the central processor. All arithmetic is compatible with the IEEE standard P754[IEE85]. With software support, the SPUR system will achieve a high performance implementation of the complete standard.

This paper focuses on the internal architecture of the SPUR FPU. The primary representation of the FPU microarchitecture is the executable description written in

1 SPUR is sponsored by DARPA under contract order 482427-25840 by NAVALEX. Additional computer resources provided by DARPA (order #4871) monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

Slang, a hardware description language[FoD83]. This description defines the functions the chip will perform and outlines the structure of the microarchitecture implementation. The hardware is described at the abstract logic level with no implementation technology implied. Because it contains both structure and function, the description will assist in simulation at both the functional and transistor levels. Thus, it provides the link between the architectural specification of the chip and its VLSI implementation.

Because the chip is fully specified by its executable description, this paper adds no new information to it. Rather, it clarifies and annotates the hardware description. To begin, an overview of the chip architectural description including the CPU interface specification is given. This section provides the necessary background for understanding the chip internals. The details of the microarchitecture are then presented with separate sections devoted to the chip datapath and control. Finally, the strategies for testing the hardware description are discussed. This paper does assume familiarity with the SPUR multi-processor workstation project; for reference see [HEL85] and [Kat85].

2. CHIP ORGANIZATION OVERVIEW

2.1. Introduction

The work presented in this paper emphasizes the implementation of the SPUR FPU rather than its design rationale. However, an overview of the chip's functional requirements is necessary to understand the design constraints placed on its implementation. For this chip, there are two distinct sources of design constraints. First are those constraints imposed by the purely architectural features of the chip, including its instruction set, supported data types and special capabilities needed for IEEE compatibility. The chip architectural description as given in [Lee86] has had the most profound effect on the internal organization of the chip. Its specifications in large part determined the structure of the datapaths where almost all of the chip's functions are performed. This report assumes familiarity with the FPU architectural description document; its contents will only be summarized here with emphasis on the implications for chip implementation.

In addition, specific constraints are imposed on the chip control by the requirements of the FPU-CPU interface. A detailed specification of these requirements may be found in [HaK86]. A more complete explanation of the chip control organization is given in this paper to supplement the purely functional requirements seen in the interface specification

document. Finally, a brief overview of the Slang hardware description language and simulator [FoD83] provides a background for understanding the coding of the hardware description.

2.2. Architectural Description

The overall goal of the SPUR FPU chip is to increase the performance of IEEE arithmetic as much as possible by implementing the most commonly used operations, data types and data representations in hardware. While only a subset of the full standard can be implemented on a single chip, the hardware provides the primitives for the software implementation of the rest of the standard. The SPUR FPU adds both instructions and data types directly to the SPUR workstation architecture and therefore is known as a tightly coupled co-processor. Consequently, the FPU mirrors the SPUR load/store architecture and in fact uses the same instruction format.

Table 1a shows the complete arithmetic operation set required by the IEEE standard highlighting the operations implemented by the SPUR FPU chip.

Table 1a. IEEE Required Operations and SPUR FPU Instructions

Type of Operation	SPUR FPU Instruction
add, subtract multiply, divide	FADD, FSUB FMUL, FDIV
remainder, square root	———
floating point format conversion	CVTD, CVTS LD_DBL,LD_SGL
round to integer	———
decimal conversion	———
comparison	FP_CMP

The primary arithmetic set for the chip consists of five operations (add, subtract, multiply, divide and compare) implemented in a single data format. The complete instruction set for the SPUR FPU chip is given in Table 1b. Note that two convert instructions and three data transfer operations were added to the primary arithmetic set to enhance the performance of data types and operations not directly supported by the hardware. The memory operations that are supported allow the transfer of all floating point data types directly to and from the chip using addresses generated by the CPU.

The single data format supported by the hardware (known as "extended format") is illustrated in Figure 1 along with the other two memory formats. The extended format

Table 1b. SPUR FPU Instructions

ARITHMETIC and DATA TRANSFER INSTRUCTIONS				
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
FCMP	cond,Rs1,Rs2	FPSW(cond)	<-	result relation
FABS	Rd,Rs1,0	FPU Rd	<-	FPU Rs1 with sign = 0
FNEG	Rd,Rs1,0	FPU Rd	<-	FPU Rs1 with inverted sign
FMOV	Rd,Rs1,0	FPU Rd	<-	FPU Rs1
CVTD	Rd,Rs1,0	FPU Rd	<-	FPU Rs1 rounded to double
CVTS	Rd,Rs1,0	FPU Rd	<-	FPU Rs1 rounded to single

LOAD INSTRUCTIONS				
Instruction Syntax		Instruction Semantics		
LD_SGL	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)
LD_DBL	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)
LD_EXT1	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)
LD_EXT2	Rd,Rs1,RC	FPU Rd	<-	M [(Rs1 + RC)

STORE INSTRUCTIONS				
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)

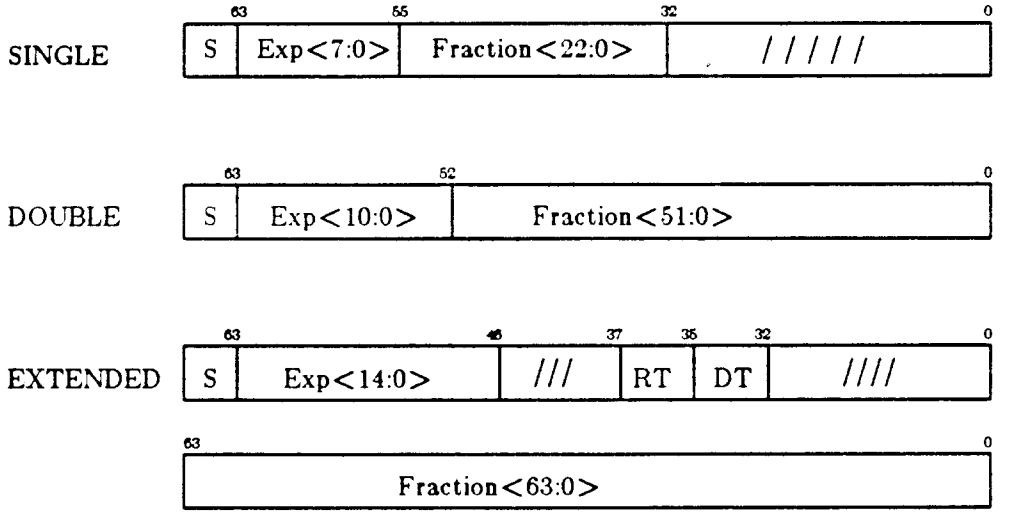


Figure 1. SPUR FPU memory formats.

consists of 87 bits containing three numeric fields (sign, exponent and fraction) and one field each to describe the data and rounding type of the number. Inside the chip, a separate datapath is dedicated to each of the numeric fields. The data type field is used to determine whether a number may be handled correctly by the arithmetic hardware, while the rounding field is used by the support software to maintain rounding precision.

As shown in Table 2, the hardware does not implement operations for certain representations of numbers [2].

Table 2. Exception Detection for FPU Operations

	Operand Exception Type					Result Exception Type		
Operation	Zero	Denorm	Norm	Inf.	NaN	Ovfl.	Undfl.	Inxct.
FADD,FSUB	H	H	H	trap	trap	D	D	D
FMUL	H	trap	H	trap	trap	D	D	D
FDIV	H/trap	trap	H	trap	trap	D	D	D
FP_CMP	H	H	H	trap	trap	-	-	-
CVTS,CVTD	H	H	H	trap	trap	D	D	D
FNEG,FABS	H	H	H	H	H	-	-	-
FMOV	H	H	H	H	H	-	-	-

Legend:
H: Hardware must produce correct result.
trap: Hardware signals exception but does not produce result.
D: Hardware signals exception and produces invalid result.
- : Cannot occur.

In general, the hardware can generate a correct answer for "ordinary" arithmetic operations. Arithmetic exceptions are for the most part simply detected and posted by the hardware; software support is required to generate the correct value for the excepted operation. Subsetting both the data formats and data representations implemented in hardware greatly simplifies the chip design.

Besides new instruction and data types, the FPU adds sixteen registers to the SPUR architecture. Each of these registers can store one operand in the extended format. Fifteen of these registers may be used for floating point operands while one is reserved for the Floating Point Status Word. The FPSW, shown in Figure 2, contains enable bits for exceptions as well as the rounding mode to be used on each floating point operation. In addition, the hardware posts information on each completed arithmetic operation, including the types of the input operands and the type(s) of exceptions that occurred. Keeping this information in the state of the FPU allows arithmetic exceptions to be handled well

2 See the IEEE standard document for a discussion of number representations.

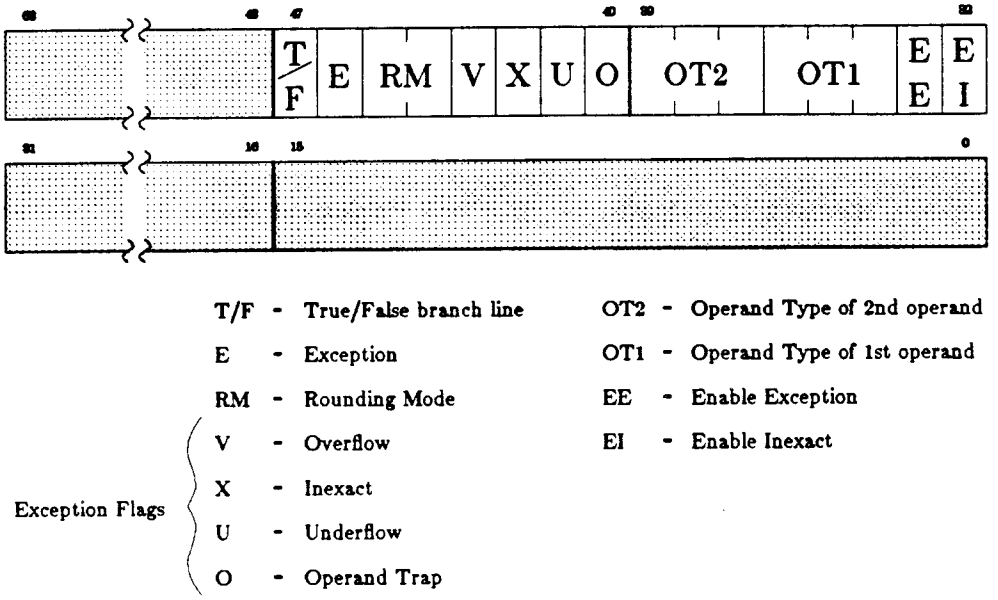


Figure 2. SPUR Floating Point Status Word.

after the instruction that created them has finished.

2.3. CPU Interface Requirements

As mentioned above, the SPUR FPU is a tightly coupled co-processor. This means that the instruction and state of the FPU are accessed in a manner transparent to the assembly language programmer. The implementation of the hardware interface of the co-processor also involves tight coupling with the SPUR CPU. The interface is synchronous, with the FPU decoding in parallel each instruction fetched by the CPU to determine whether that instruction is to be executed by the FPU. The interface control structure allows the CPU to execute instructions in parallel with FPU arithmetic operations. Consecutive FPU arithmetic operations are serialized, however these operations may execute in parallel with FPU load and store operations as well as with CPU instructions. Finally, on FPU memory operations the CPU actually generates the address for the instruction; the FPU need only transfer the data itself. Thus, the interface provides enhanced performance of co-processor functions at a minimum overhead.

Figure 3 illustrates the signals used to implement the co-processor interface. Because the CPU fetches instructions from an internal buffer, the instruction currently fetched by the CPU must be sent on dedicated wires (fpuOPCODE, fpuRS1, fpuRS2, fpuRD) to the

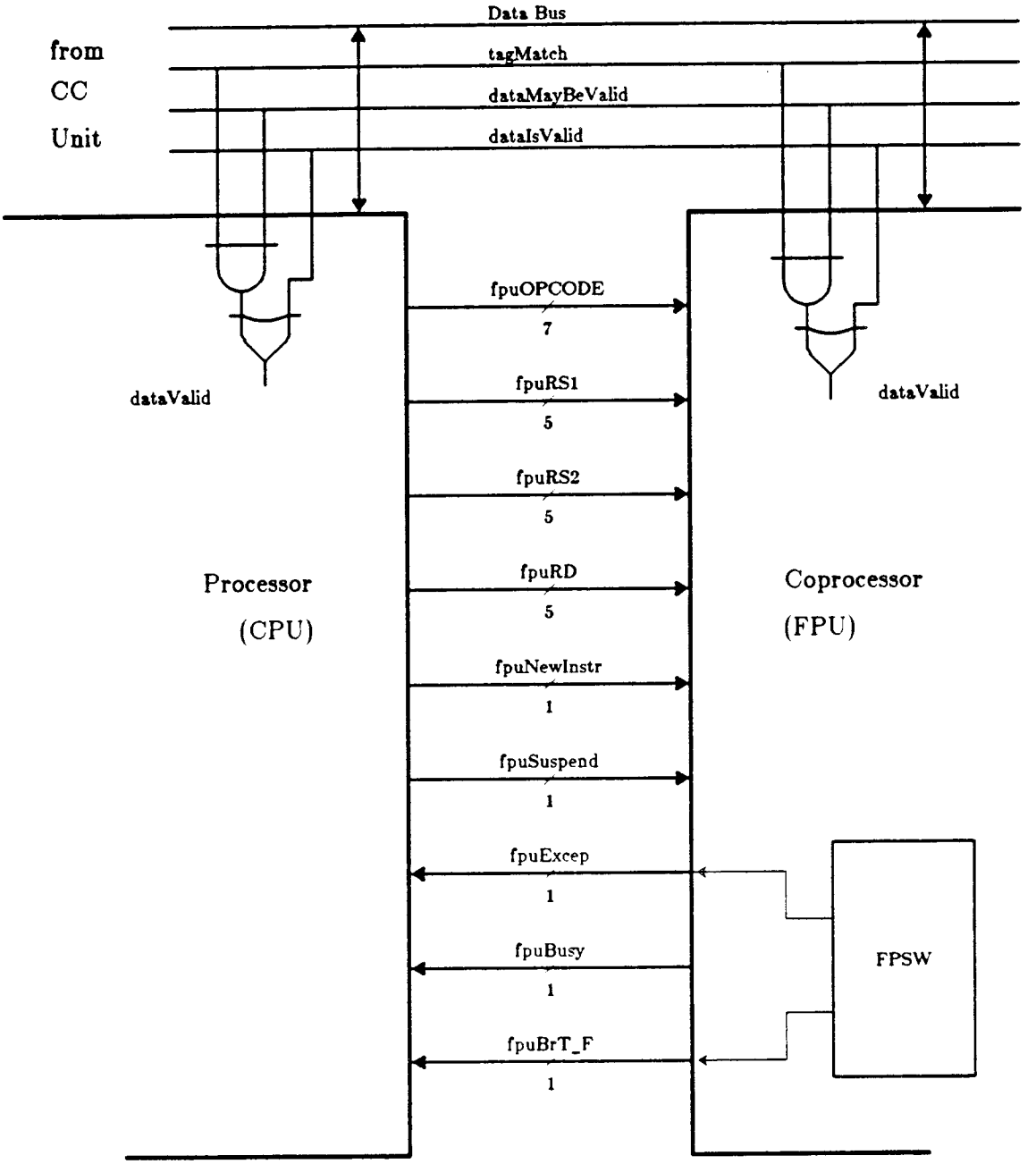


Figure 3. Signals for Co-processor Interface.

co-processor. In addition, the CPU sends one line (fpuNewInstr) to indicate the validity of the instruction sent. The CPU may not assert fpuNewInstr for several reasons, including a pipeline stall due to consecutive co-processor instructions. Another signal from the CPU (fpuSuspend) indicates a pipeline stall specifically due to a miss in the processor cache. This signal not only halts the issue of new FPU instructions but also may prevent

instructions from completing.

Because a cache miss may result in a page fault trap, instructions that were issued after the offending load or store must not change the state of the processor. This requirement imposes timing constraints on the execution of arithmetic instructions. Each arithmetic instruction must contain at least two execute cycles and a distinct write cycle. This is because FPU instructions may be suspended and possibly killed during the two execute cycles just like CPU instructions. This makes the semantics of FPU suspensions and traps the same as those for the CPU.

Whereas the CPU signals are provided for control, the FPU returns status information for its portion of the interface. The fpuBusy signal, for example, is asserted only while the FPU is executing an arithmetic operation. This signal causes the CPU to stall if another FPU arithmetic instruction is fetched while it is asserted. The other two status lines indicate exception and branch conditions for the last completed FPU instruction. These two lines are attached directly to bits in the FPSW, allowing the support software to set or clear them directly.

None of the precise timing of the above signals has been given here. A large number of possible timing sequences arise from the parallel execution of CPU and FPU with the possibility of traps and/or pipeline suspensions. Also, although the discussion of this interface has been in terms of the SPUR FPU, the interface is intended for general application co-processors. A discussion of the timing sequences for this interface as well as the extensions required for generality are beyond the scope of this paper. For a more complete discussion of the SPUR co-processor interface, please see [HaK86].

2.4. Slang Overview

As mentioned above, the hardware description for the SPUR FPU was written in Slang. The term Slang may refer either to the hardware description language or its corresponding simulator. Slang the language is actually an extension of Lisp: i.e. a set of Lisp macros and functions that support constructs for describing hardware. These constructs include functions for describing the operation of the circuit as well as constructs for describing its partitioning. Slang the simulator is a set of functions that provide a purely event driven simulation of a Slang circuit model.

In Slang, a circuit model consists of a collection of objects called *nodes*. Each node contains an *update* function and a list of zero or more nodes grouped into a *depends* function. In the simulator, a node corresponds to a single event whose value is evaluated by executing its update function every time one of the nodes listed in its depends function changes. Thus, a circuit is built from nodes by describing its components using the node update function and its interconnection network using the depends function.

Because Slang is an extension of Lisp, it has several capabilities not ordinarily found in hardware description languages. For example, although nodes may evaluate to tri-state logic values, they are not limited to that type of representation. In fact, a node may contain arbitrary Lisp functions in its update clause and have any Lisp data object as its value. Thus, hardware may be described at different levels in Slang; differing amounts of detail may exist even in the same circuit description. Users of Slang have the full capability of the Lisp interpreter for debugging and interaction. Furthermore, they may load arbitrary Lisp macros or functions along with the hardware description and have those functions execute as if they were built into Slang. These capabilities make Slang a very flexible system suited for many different kinds of hardware simulation. As will be seen, much of this flexibility was used in generating the description of the SPUR FPU.

3. FPU DATAPATH ORGANIZATION

3.1. Overview

As seen in Figure 4, the entire chip is split into separate sections for memory and arithmetic operations. The load/store portion consists of data packing and multiplexing logic and contains separate units for load and store operations. The arithmetic logic is broken up into a section for addition/subtraction of floating point fractions (called the "fraction box"), a section for fraction multiplication and division (the "multiply/divide section"), and a section for handling exponent, sign and type fields of operands ("exponent section" for short). Within the arithmetic section, the fraction, exponent, sign and type datapaths all operate concurrently, exchanging information at several points in the computation.

This split between the load/store and arithmetic units is more than physical; the two section operate concurrently and share only the register file between them. The register file may be shared without conflict because it is dual ported; two operands may be read or

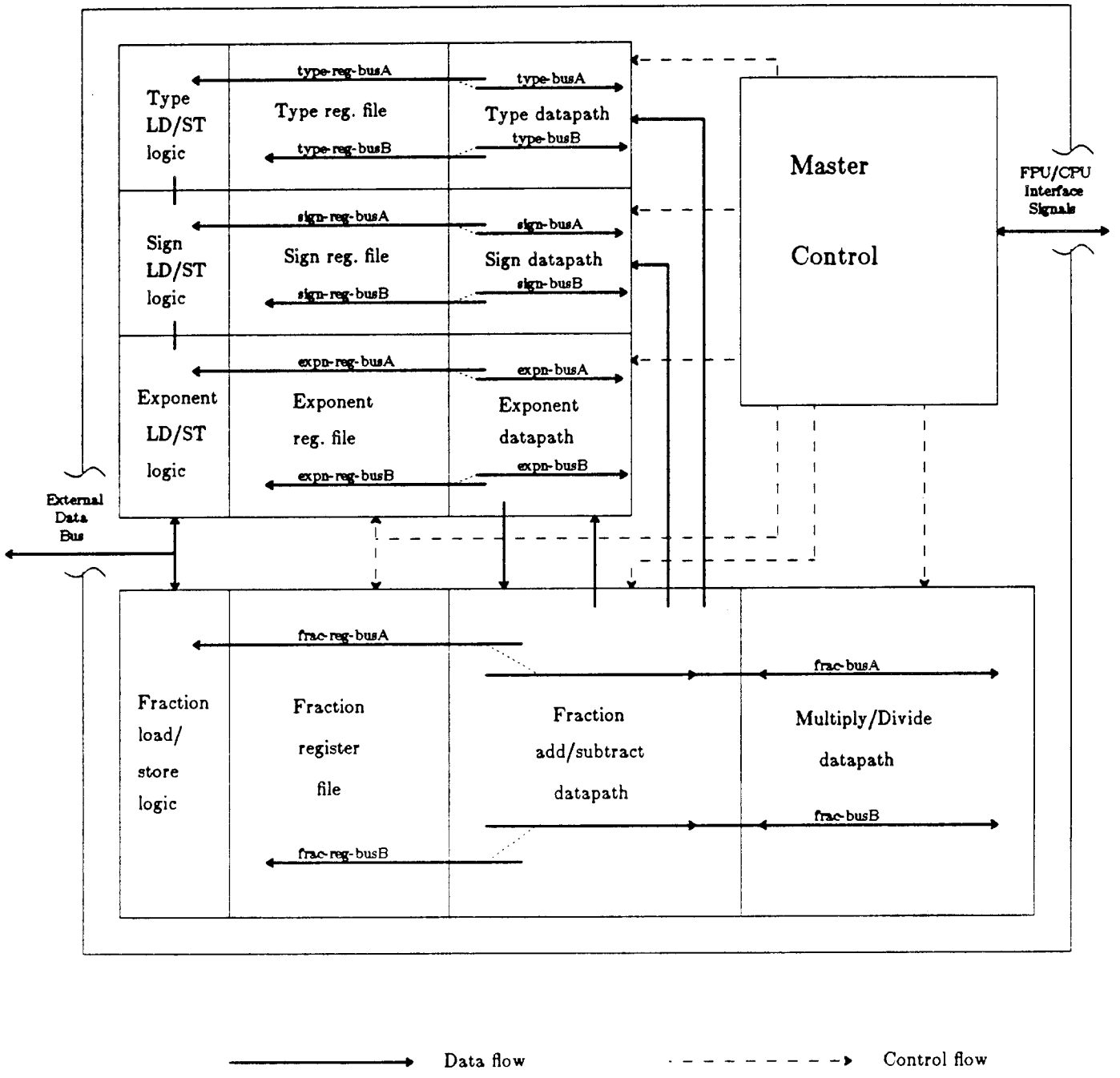


Figure 4. FPU Chip Organization

written simultaneously. Since all reads occur at the beginning of an instruction, there is never a conflict between reads for stores and reads for arithmetic operations. Although two writes may occur simultaneously, the load/store and arithmetic sections use different busses for writing. Also, the A/B busses in the arithmetic datapath are different from the

busses in the register file. This allows the memory and arithmetic sections to operate independently between data transfers to the register file.

In this section of the paper, the chip datapaths will be analyzed from both the operation and implementation point of view. First the flow of operands for each instruction through the various datapath components will be detailed, outlining the major datapath components and the interconnections between units. Then, the implementation of each chip datapath component is fully detailed.

3.2. Instruction Flow

Although many FPU instructions share components of the chip's datapaths, not all operations use these components in the same way. Table 3 repeats the set of instructions implemented by the FPU, taxonomizing them according to the path taken by operands through the chip.

Table 3. SPUR FPU Instruction Classes

Arithmetic Instructions		
uses add/subtract datapath only	uses multiply/divide datapath	uses data transfer hardware only
FADD FSUB CVTS CVTD FP_CMP	FMUL FDIV	FMOV FNEG FABS

Memory Instructions	
Needs alignment/packing only	Needs exponent, type field handling
LD_EXT1, LD_EXT2 ST_EXT1, ST_EXT2	LD_DBL,LD_SGL ST_DBL,ST_SGL

Note that there are several variations of data routes even within the set of memory and arithmetic operations. Each of these routes (indicated as instruction subsets) requires a substantial portion of logic dedicated to their implementation. Within the subsets, however, the differences are primarily in the control of the datapath rather than the datapath components themselves. Note that this section provides a complete but not fully detailed description of instruction sequences, a full description of the exact signals involved is given in Appendix A of this report.

3.2.1. Memory Operations

All load and store operations follow a path going directly between the chip data pins and the register file via bus A. However, there is a significant difference in the amount of logic required to process data in extended format versus single or double memory format. As seen in Figure 5, the extended format data can be field extracted for loads and field packed for stores since the SPUR extended memory format is isomorphic to the FPU internal register format. Since only 64 pins are supplied to the FPU, two instructions are needed to load (or store) an 87 bit extended format value. The LD_EXT1 instruction loads the exponent, sign and type fields, while LD_EXT2 loads the fraction. Finally, both loads and stores require separate master/slave latches to handle consecutive memory operations.

Figure 6 illustrates the logic required to convert from the other two SPUR memory formats to register format and vice versa. Whereas the sign bit needs no conversion, the fraction portion requires alignment to the most significant bit positions of the register file bus. Also, the bit to the left of the binary point (called the hidden bit) must be made explicit at this time. Conversion of the biased exponent to internal format involves a complement and sign extend operation from the proper bit position. Also, the case of a zero input exponent must be handled specially. This exponent corresponds either to a zero or to a denormalized number, depending on whether the fraction is zero or not. Flipping the LSB is sufficient for denormalized number conversion; a special exponent value must be produced for the number zero. An "all ones" detector along with the fraction and exponent zero detection circuitry are needed to produce the data type bits for the number. Note that these are not present in single or double memory format. This conversion is only complete for ordinary, zero and denormalized numbers, however only these numbers ever will be processed by the arithmetic section. Numbers that are not handled by the arithmetic section (infinity and NaN) will have their type fields set correctly.

Single and double memory format stores are simplified greatly because the correct operand data type field is present already. Only an exponent conversion for zero or denormalized numbers (as indicated by the data type field) is required. The other data portions for the number only need to be selected and packed into the proper bit positions.

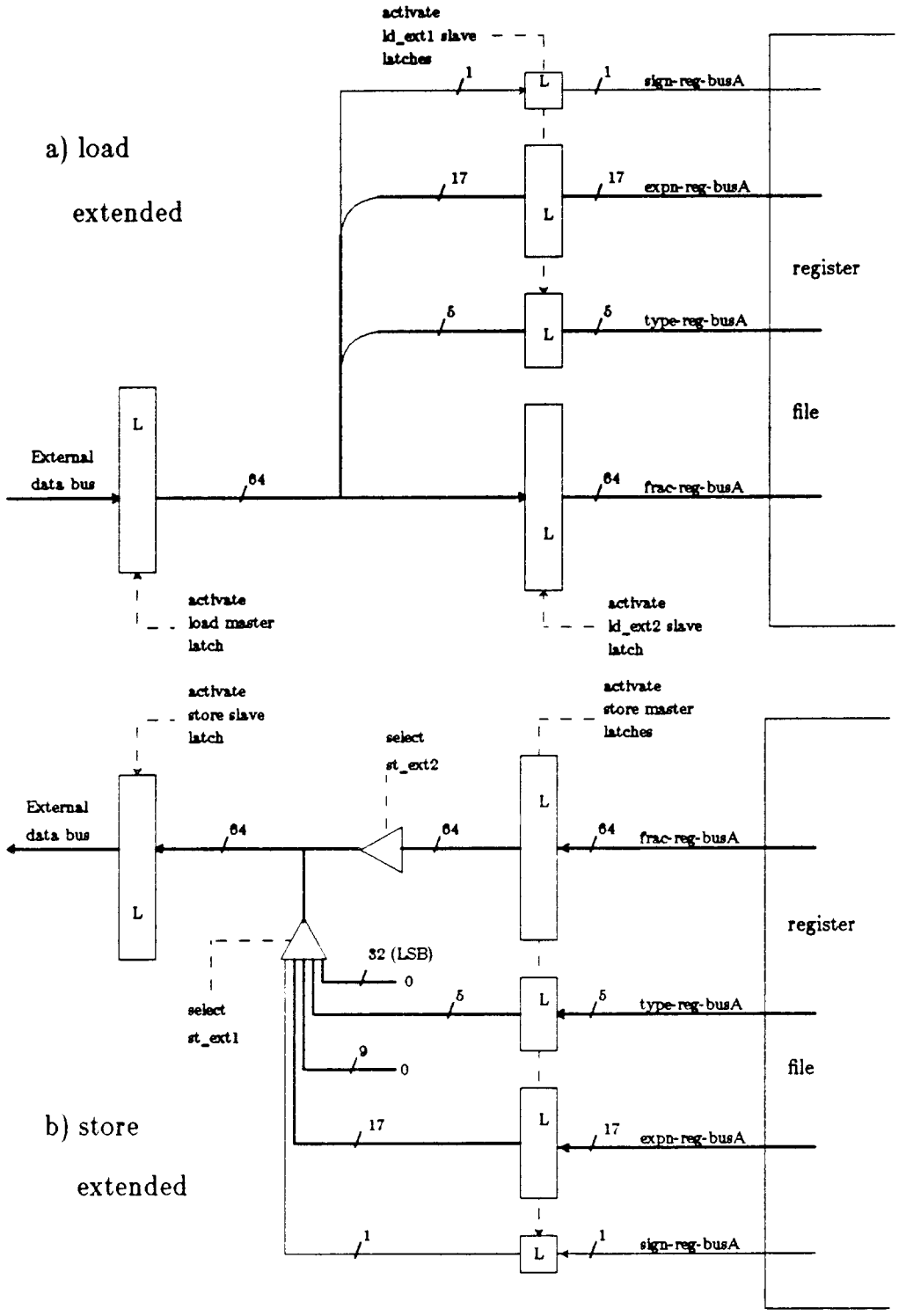


Figure 5. Extended format memory operations.

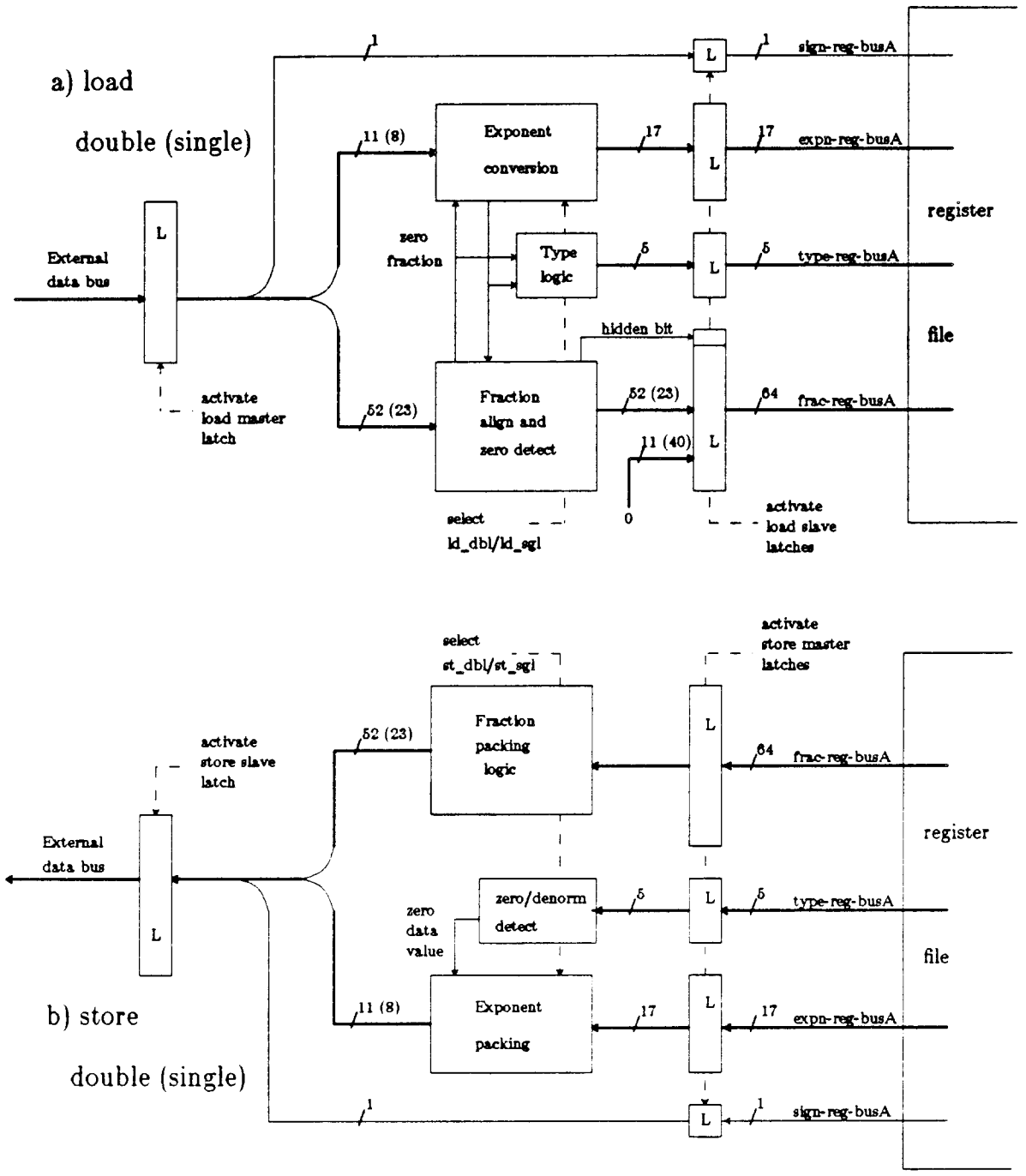


Figure 6. Double/single format memory operations.

3.2.2. Arithmetic Operations

All arithmetic instructions must pass through the exponent, sign and type handling logic as well some portion of the fraction box to produce a result. The three classes of

instructions (add/subtract, multiply/divide and data transfer) differ in the manner and extent to which they use these datapath components. The largest class of instructions, consisting of add, subtract, compare and two converts, uses all of the above mentioned components. The data transfer instructions need only the register read/write logic in these datapaths to accomplish their tasks. The multiply and divide instructions generate partial results in a completely dedicated datapath. Even these instructions require the fraction datapath to produce a final result, however.

3.2.2.1. Add/Subtract

Figure 7 illustrates the sequence of operations for add, subtract, compare and convert operations. Although the basic sequence for each datapath is simply a single pass, the exponent and fraction units must operate concurrently and exchange information. For adds and subtracts, this sequence begins with the reading and latching of the input vectors from the register file. The operand fractions must first be aligned to compensate for the differences in exponents. This is done by computing the difference in the exponent datapath and sending the result to a multiplexor and shifter in the fraction datapath. The operand with the lesser exponent is selected and passed through a right shifter. Each operand is then fed to the fraction adder/subtractor for generation of the intermediate result. In parallel with the result calculation, the greater exponent is selected and fed to the exponent adder/subtractor.

Once the intermediate result has been computed, it must be rounded and normalized to form the final result. The normalization step involves both shifting and zero detection with the shift amount and zero detect signal being sent to the exponent adder/subtractor. For a non-zero fraction, the result exponent is then computed by adding or subtracting the greater exponent by the normalizing distance. A special value exponent is inserted into the destination register for a zero exponent. The fraction zero detect signal is also sent along with the greater fraction selection signal to the sign determiner. The result sign is a simple combination of these signals and the input signs; it is computed in parallel with the result exponent. The result data type value also is generated from the fraction zero detect signal. Finally, the result exponent is checked for underflow or overflow. Only after this operation has finished are the various results written back to the register file.

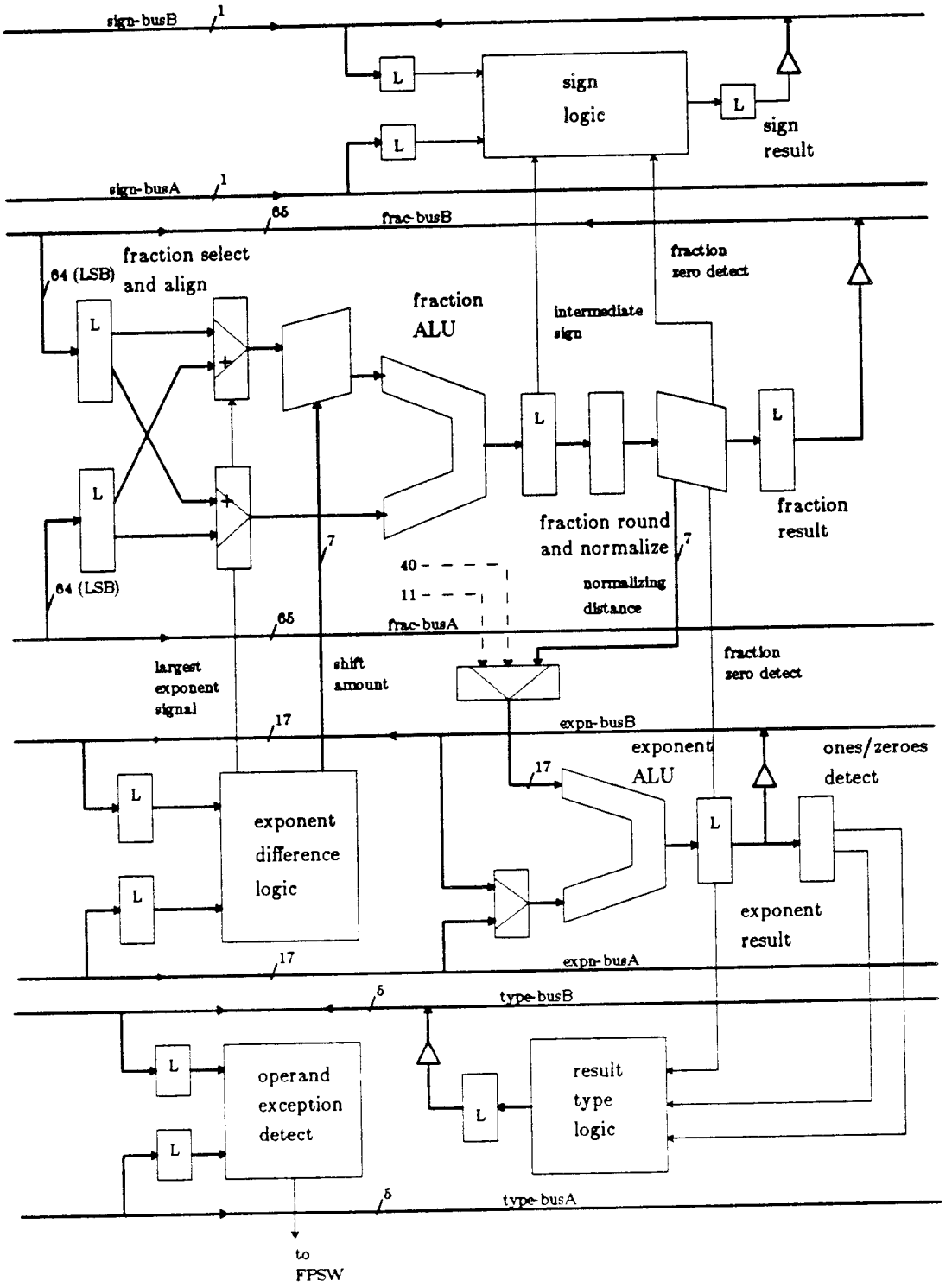


Figure 7. Add, subtract, compare and convert operations.

Although the above mentioned datapath flow has been described exclusively in terms of the add and subtract instructions, very little needs to be added to describe the compare and convert instructions. The compare instruction behaves exactly like a subtract operation except that the result write is inhibited. Instead the result sign and zero fraction detect signals are combined to set the proper result condition in the FPSW. The fraction computation for a convert instruction differs only in that the active operand is routed to the alignment shifter and shifted by a fixed amount. Since the other operand of a convert instruction is always zero [3], the rest of the fraction datapath may be used without modification. The result exponent requires an addition of the fixed alignment amount along with the adjustment for normalization. This is done with two passes through the exponent adder/subtractor; the result of the first computation is fed back via bus B to its left hand input. Finally, the sign and type determining logic are bypassed on a convert operation since the result sign is always the sign of the active operand.

3.2.2.2. Multiply/Divide

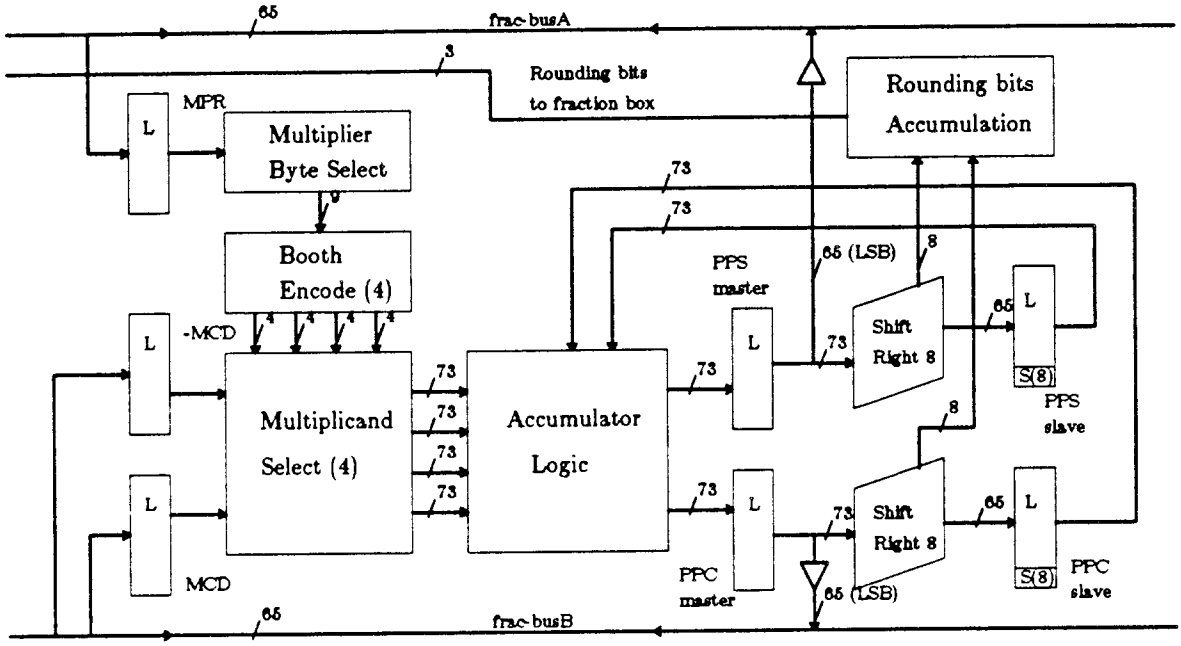
As seen in Figure 8, the flow of operands for the multiply instruction requires the coordination of the fraction box and the multiply/divide datapath at the beginning and the end of the operation. The multiplier is able to load two of its input operands directly from the register file, however it also requires the complement of one of the operands. This complementation is accomplished by subtracting the input operand from zero using the fraction box adder/subtractor and sending the result to the multiplier one cycle after the uncomplemented version.

The actual multiplication step is carried out using an 8-bit version of Booth's algorithm. This involves selecting and encoding one byte from the multiplier then using that byte to select four instances of the multiplicand or its complement. These instances are added to the previous partial product to form the new partial product for that step. The two parts of this loop are pipelined; a new byte is selected and encoded while the multiplicands selected by the previous byte are being accumulated.

To increase performance, the partial product accumulation is done using carry save addition. Thus, the partial products are encoded as a sum and carry vector during the

3 This is because register Zero (R0) is hardwired to zero.

a) Multiply



b) Divide

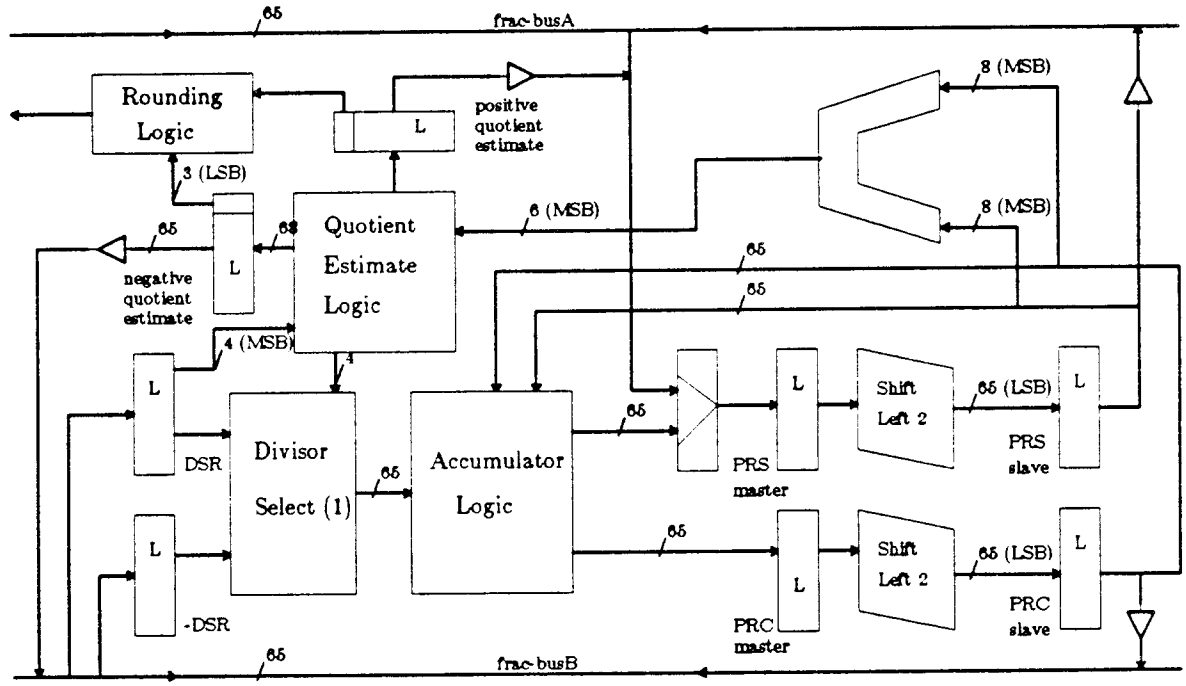


Figure 8. Multiply and divide operations.

entire multiply operation. At the end of the operation, the final sum and carry vector must be added together to produce the final result. These vectors are therefore sent directly to the inputs of the fraction box adder/subtractor. From then on, the final result is formed by addition, rounding and normalization, exactly as it would be for the add instruction.

Although the divide instruction uses some of the hardware mentioned above, the sequence of operations is quite different. First, the divisor is loaded directly from the register file into the same latches occupied by the multiplicand and multiplier on multiplies. Like the multiplicand, the divisor must be complemented in the fraction box and sent to the divide unit later. The dividend, however, is loaded into the fraction box input latch and is held there for one-half cycle. It is then transferred not to the multiplier latch but to one of the latches that previously held the partial product; a zero is placed in the other partial product latch.

The main divide loop uses a repeated bit quotient estimation scheme described in [Tay85]. On a given loop iteration, a two bit estimate is made of the entire quotient. This estimate is a function of the six most significant bits of the partial remainder (which for the first step is the dividend) and the four most significant bits of the divisor. Because the quotient may have been overestimated on a previous iteration, a given estimate may have the range ($-2 < X < 2$). The quotient estimate is encoded as a three bit signed magnitude value, positive and negative estimates are kept in different latches. The quotient estimate also selects an instance of the divisor or its complement which will be added to the previous partial remainder to form a new one. This selection is made such that the partial remainder (whether positive or negative) always is reduced in magnitude by this operation. Thus, with each step the precision of the quotient estimate increases and the partial remainder converges to zero.

At the end of the final quotient estimate step, the negative quotient estimate vector must be subtracted from the positive quotient estimate vector to obtain the final result. To preserve accuracy, however, the sign of the final remainder must be included in the subtraction. To do this, the partial remainder vectors must be sent to the fraction box for an addition to form the final remainder. The final remainder itself remains at the output of the fraction box adder/subtractor; only its sign is fed back to the divider. Only after this first addition has taken place can the quotient estimate be sent to the fraction box. As with the result of a multiply, the final quotient is rounded and normalized before

being written to the register file.

The sequence of operations for computing the other data portions (sign, exponent and type) of the result is exactly the same for multiplies and divides. The exponent computation is a two step process similar to that used in the convert instructions. The initial result is formed by either adding (for multiply) or subtracting (for divide) the two input exponents. This result is then fed back to the exponent adder/subtractor via bus B and adjusted by the normalizing distance of the final fraction result. The result sign for both instructions is the exclusive-OR of the input signs; it is latched early in the operation. Finally, the data type of the result is computed in the manner already described for the add and subtract instruction.

3.2.2.3. Data Transfer

The FPU data transfer instructions (fmov, fneg, fabs) have a very simple operation sequence since only the sign of the input operand is ever modified. The single operand is read from the register file and latched directly into the destination latches of the various datapaths. A small piece of logic is placed in the path of the sign bit to compute the correct sign; the other parts of the operand are latched directly from the bus. The values of the destination latches are then written back to the register file without further manipulation. Thus, no extra datapath hardware is required to implement these instructions.

3.3. Implementation

3.3.1. Fraction Box

The use of the fraction datapath is not limited to the add and subtract instructions; the unit produces the result fraction for all arithmetic operations. This unit is also used to complement one of the input operands for the multiply and divide instructions and to compute the final remainder of a division. To perform these functions, the fraction datapath contains the following four components:

1) Operand binary point alignment

2) Adder/Subtractor

3) Normalization and Rounding

4) Exponent Adjustment

These components correspond to the basic operations required to complete a floating point add or subtract. The busses and main datapath components are 65 bits wide; in addition three bits must be maintained separately to ensure correct rounding of the result. Figure 9 gives a detailed view of this unit.

The operand alignment datapath is used exclusively for the add and subtract instructions. The alignment operation involves routing the fraction with the lesser exponent through the right side of the datapath and a right shifter. The shift amount and fraction routing is controlled by the exponent difference of the two operands. This right shift generates the rounding bits for the add and subtract instructions. The two most significant bits shifted out of the operand become the guard (G) and round (R) rounding bits. The rightmost round bit is generated by ORing the rest of the bits that were shifted out; it is called the sticky (S) bit. Because the bits shifted out of the operand can be condensed, the aligning shifter need only handle a shift amount of up to 66 bits. Also, the exponent shift amount signals are condensed into a 7 bit shift vector and a single signal for all shift amounts greater than 127.

The aligned fractions are latched at the input to the main adder/subtractor in phi1 of the second execute cycle. Since phi1 of the first execute cycle is dedicated to reading the input operands from the register file, a little more than two phases are allocated for operand alignment. The exponent box produces the control signals for alignment; the shifter must wait one phase for these results. Thus, the actual fraction shifting is done in phi3 and phi4 of the first execute cycle.

At this point the operands are ready for the actual addition or subtraction operation. Note that the operands generated by multiplication and division have the same magnitude, hence they are multiplexed into the unit directly at this point. Subtraction in this unit is performed by complementing the left operand and adding it to the right operand. Since the three rounding bits are always associated with the right operand, the size of the main adder/subtractor is reduced because the rounding bits do not participate in the subtraction.

Since both addition and subtraction are performed, the output of this section is a 69 bit two's complement value. The value consists of a sign bit, two bits to the left of the binary point, 63 bits to the right of the binary point and three rounding bits. The result

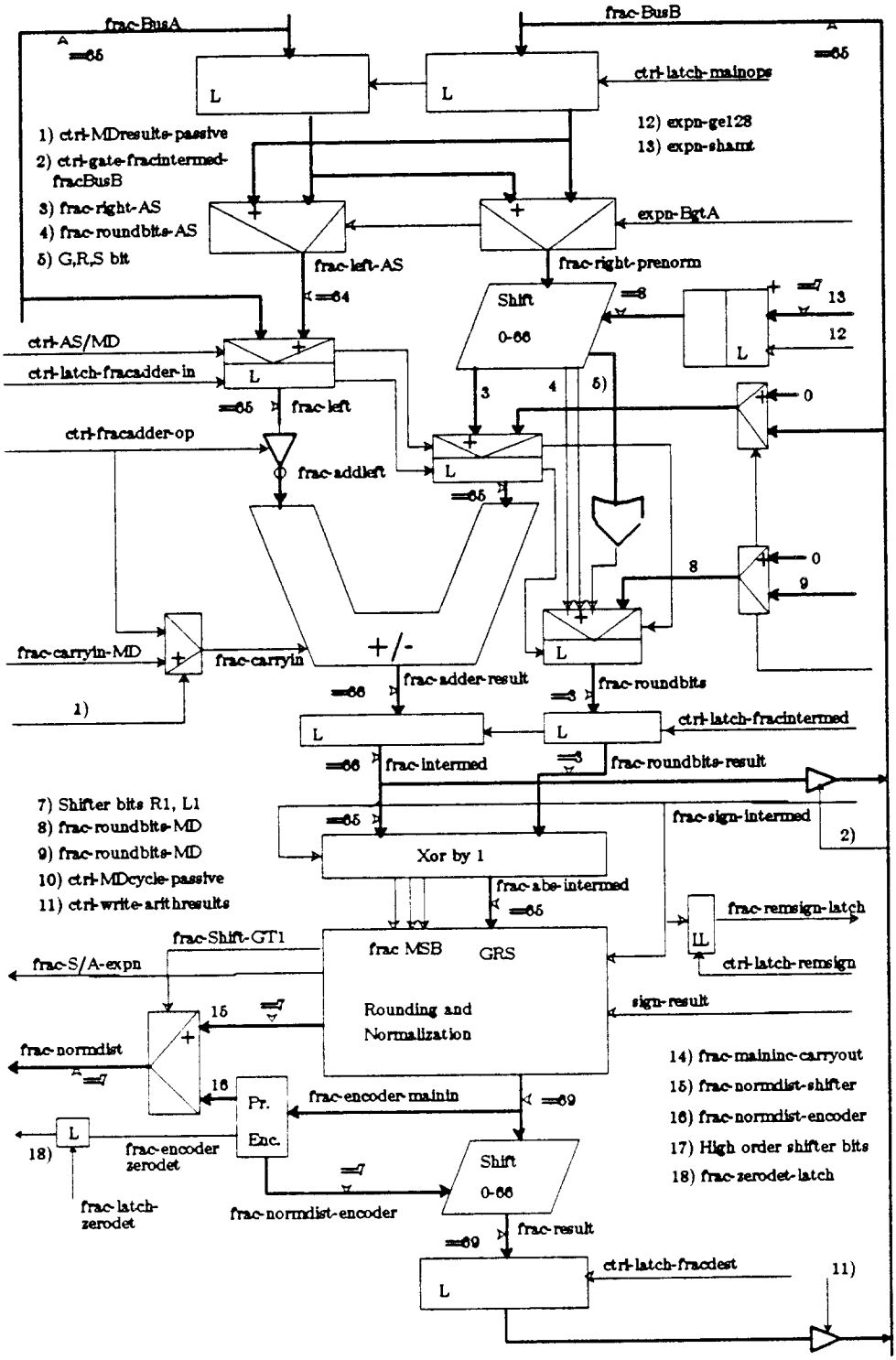


Figure 9. Detailed view of fraction box

is immediately returned to positive magnitude form by exclusive-ORing all of the 68 rightmost bits with the intermediate sign bit. Thus, the final output of this unit is a 68 bit positive number. The number is correct if the result of the adder/subtractor was positive; otherwise the value must be incremented to finish complementing it. The entire addition/subtraction operation is estimated to take two clock phases.

The basic operations that remain to be performed by this unit include rounding, normalization and incrementing. Normalization here means bringing the uppermost one in the number to the bit position just to the left of the binary point. This requires that the value be shifted right by one or left by an amount ranging from zero to 66 bits. Since rounding must be done to the normalized number, it would seem that normalization would be the first thing to do in this section. That is however NOT correct because the number may still need to be incremented to finish the complementation; this could in turn change the normalization shift amount [4]. Furthermore, the rounding may also require that the number be incremented even if it was originally positive. Given these constraints, the primary design goals for this section of the datapath are to use one incrementer for both rounding and complementation and to make only one adjustment to the exponent.

The solution that satisfies the above design criteria exploits the fact that different rounding and normalizing actions are taken based on the value of the input. Only those numbers that require a shift amount of zero or one (left or right) need to be rounded. Although it should be clear that shifting left by more than two clears the rounding bits, the case of shifting left by two is not obvious. Numbers in that range ($0.25 < x < 0.5$) must have resulted from a subtraction where the lesser operand was shifted right by no more than one. Hence all significant bits are shifted out of the rounding bits by the normalization shift left of two. These numbers are therefore treated differently from those numbers that require a normalizing left shift of greater than one.

As seen in Figure 10, the input to this section is first tested to see if the required shift amount for normalization is right by one (R1), zero (Pass), left by one (L1) or left by greater than one (GT1). Numbers with shift amount greater than one are passed without shifting to the incrementer; these numbers are incremented only for complementation.

4 To see this, considering incrementing 0.11111.

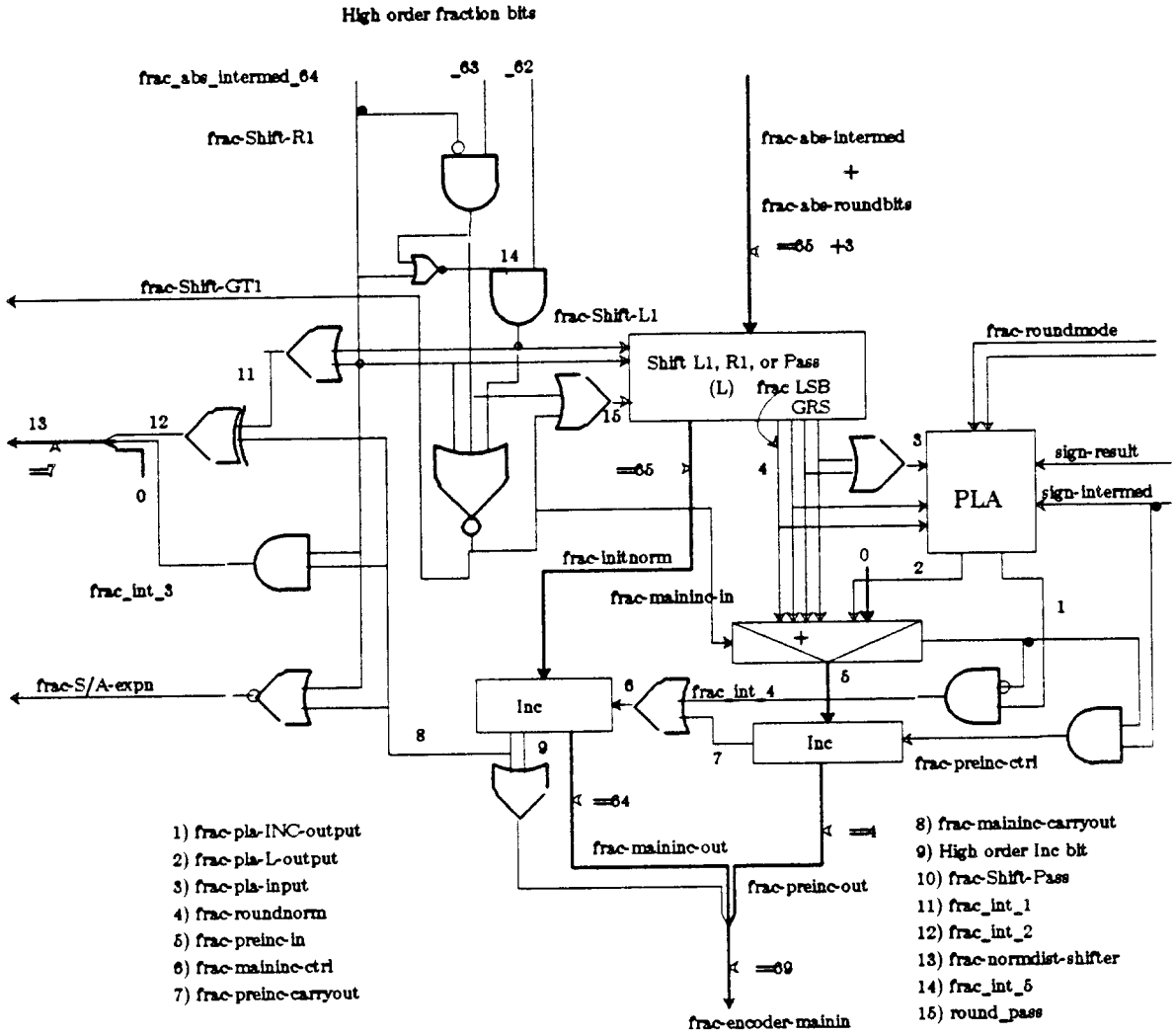


Figure 10. Details of fraction rounding and normalization.

Otherwise, the number is normalized immediately and only the uppermost 63 bits of this result are sent to the incrementer; the four least significant bits (called the L, G, R, and S bits) are sent to the rounding pla. The pla is also sent the intermediate sign of the number so that the need for a complementation increment can be taken into account. This pla returns the new value of the L bit, increments the uppermost 63 bits if necessary and clears the rounding bits. The uppermost 63 bits of any number will never be incremented both for complementation and for rounding, thus only one incrementer is needed. Incrementer overflow is handled by ORing the carry output of the incrementer into the most significant output bit and by modifying the exponent adjustment amount.

After the incrementing is complete, the number consists of at most 67 bits and is ready for the final normalization step. The number is passed to a priority encoder; this will generate a left shift amount ranging from zero to 66. The encoder generates the shift amount in both a "one-hot" encoding that controls the shifter and a binary encoding that is used to adjust the result exponent. Also, the priority encoder must detect an all zero input and latch that signal for the exponent and type datapaths. The normalizing shifter generates the final result for the fraction unit. Currently, the entire rounding and normalization process is allocated only two phases. This estimate is probably too optimistic from a layout implementation point of view.

In addition to generating the final result, the normalizing unit must also adjust the result exponent to reflect all normalization that was done. Exponent adjustment information comes from three places of this section, the initial shifter (for the R1, Pass and L1 cases), the incrementer overflow or the priority encoder. The first two sources are mutually exclusive of the third; the correct exponent adjustment will come from the priority encoder only in the case where the shift amount before the increment is greater than one. Thus, the incrementer overflow and initial normalization information is gathered into a single shift amount by combinational logic. The shift GT1 signal is then used to choose between that shift amount and the value from the priority encoder. The results of this section consist of the magnitude of the shift amount and a signal that indicates whether to add or subtract that value from the other exponent.

3.3.2. Multiplier and Divider

As mentioned before, the floating point unit uses a version of Booth's algorithm for multiplication and a repeated quotient estimation scheme for division. Although the principles of these algorithms are different, each requires a loop where bits from one operand (the multiplier or divisor) are repeatedly selected to control the selection of instances of the other operand (multiplicand or dividend) or its complement. On each iteration these operands are added to the previous partial results to form the new partial results. Because of the similarity between the algorithms, the entire accumulator loop (consisting of operand selection multiplexors, carry save adder tree and partial result latches) is common to both the multiplier and divider. This sharing of hardware for multiplication and division was absolutely necessary to meet the area and performance goals for this datapath section.

3.3.2.1. Multiplier

For multiplication, the operand selection portion of the loop mentioned above consumes eight multiplier bits per iteration. As seen in Figure 11, this requires four sets of Booth encoders and multiplicand selectors, each of which consumes two multiplier bits. To feed these encoders and selectors, each iteration requires an entire byte of the multiplier plus the high order bit from the previous byte for processing. The nine bits are arranged into four groups of three lines, with the high order bit of each group forming the low order bit of the next group. These lines are then recoded into four groups where each group contains four bits in a "one hot" encoding. Each group then controls a multiplexor that selects one of four versions of the multiplicand; these being the multiplicand or its complement taken as is or shifted left by one. Since each of these partial results is generated from two multiplier bits, the multiplicand in each multiplexor in the group is shifted left by two bits from its predecessor.

The outputs of the four multiplexors are then added to the two results of the previous step to form two new results. A tree of four carry-save adder rows is required for this task. Since four two bit partial results are generated at once, the carry-save adders must be 73 bits wide. Although four rows of carry-save adders are required, the order of addition has been arranged so that the critical path through the tree contains only three adders. This is accomplished by placing the six input operands into two groups of three and adding them in parallel. The four vectors that result are then accumulated in sequence to generate the two final result vectors. Once the result vectors have been generated, they are latched into the master of the partial product accumulator latch. A new loop begins on the next phase when the slave result latches are loaded.

Up to now, no mention has been made of the timing of the multiply loop. Since the generation of multiplicands and the accumulation of partial product vectors are independent, these functions are pipelined. As seen in Figure 12 the shifting and rounding of the partial product vectors is done in parallel with multiplicand selection. Each pipelined function is estimated to take two clock phases, with the master and slave cycles occurring on even and odd clock phases respectively. The entire accumulation process requires 8.5 loops. The extra half loop in the multiplication is required to finish the evaluation of the leftmost bit of the multiplier. The partial product master is latched on even phases so that the result of the multiplication (from the last half loop) may be sent to the fraction box on the following phase.

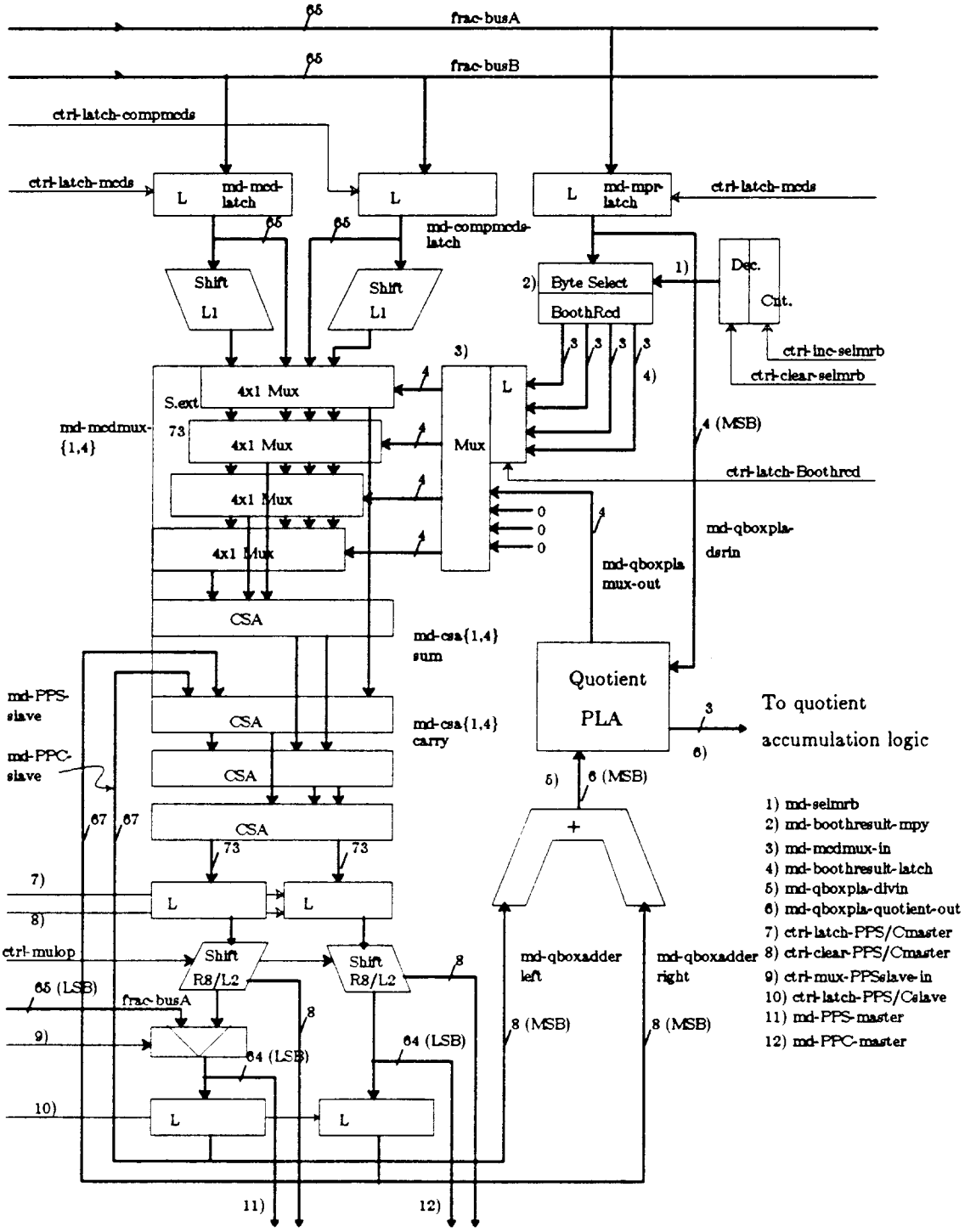


Figure 11. Multiply/Divide accumulator loop.

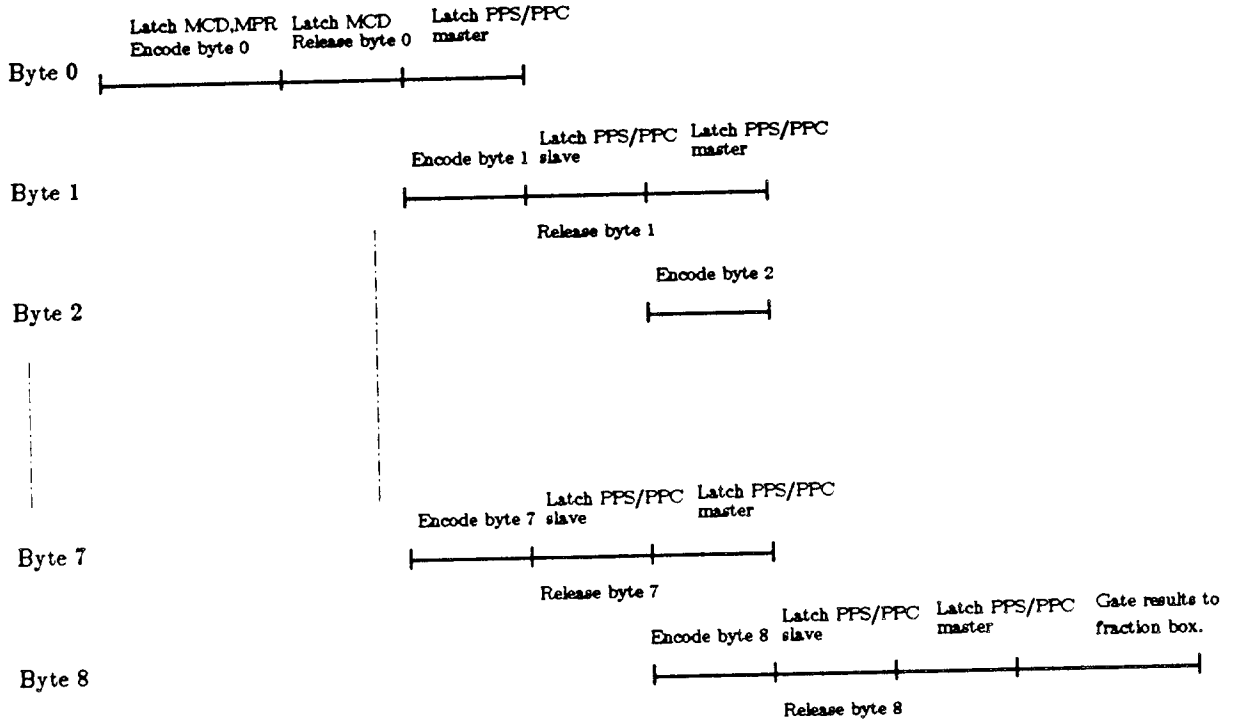


Figure 12. Timing of multiplier pipeline.

Note that the partial product vectors must be shifted to the right by eight bits before they can be returned to the carry-save adder for the next loop. The eight bits shifted out of each result vector are the least significant bits; these must be condensed to form the rounding bits for the operation. This is done using the adder and OR gate shown in Figure 13. On each iteration, the "partial" sticky bit is fed back to the rounding adder while the carry output of the last addition forms the least significant bit of the partial carry for the next iteration.

On the final half cycle of the multiply, this circuit must furnish the rounding bits to the fraction box. The mapping of these bits is complicated by the fact that the leftmost bit of the rounding adder actually belongs in the L (bit 63) position of the fraction unit datapath, not in the rounding bits. Thus, the second from leftmost position of the adder corresponds to the guard bit of the fraction unit. Furthermore, the multiply unit does NOT produce a round (R) bit; using bit five from the adder is not correct. Instead, a zero is placed in the round position and the rightmost six bits of the adder are ORed with the previous value of the sticky bit to form the final sticky bit. Finally, the carry output

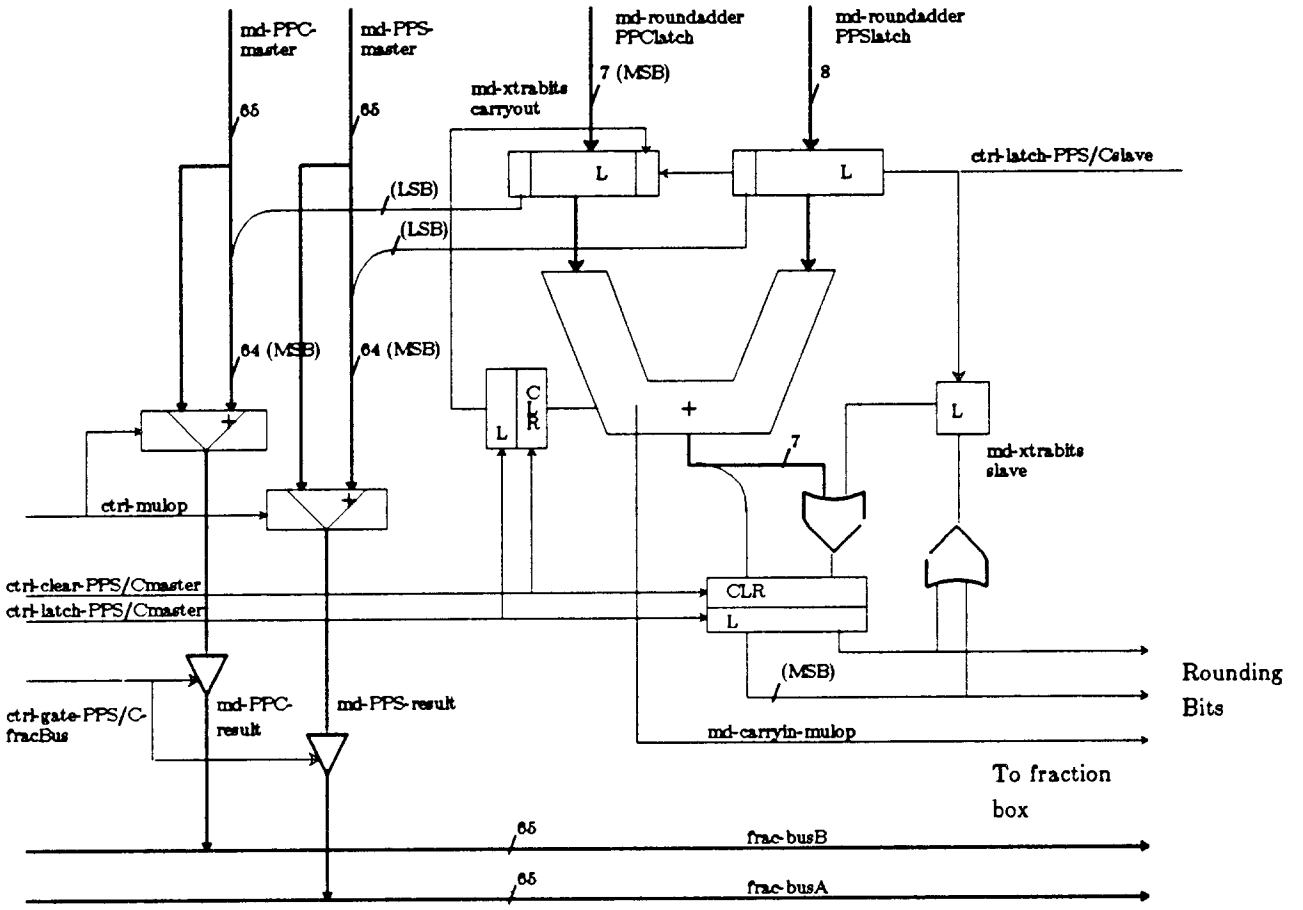


Figure 13. Rounding bit accumulation in multiplier.

from bit position six of this adder must be sent to the fraction box to form the carry input of the final result accumulation.

3.3.2.2. Divider

As denoted in Figure 11 the divide instruction actually does not require much additional hardware beyond that needed for the multiply instruction. As mentioned previously, the divider uses the same accumulator/selector loop as the multiplier. Since only two quotient bits are generated on each loop, only the topmost of the four selection multiplexors is used for divisor selection. Only a slight modification is needed to use the partial product latches as partial remainder latches; they are loaded with the dividend and zero respectively at the beginning of the operation. Also, the new partial remainder vectors are shifted left by two bits before taking part in the next quotient estimation. Thus, the

divider loop on each iteration consumes two bits of the partial remainder and generates two bits of quotient.

Only the quotient estimate hardware is unique to the division instruction. This hardware consists of the quotient estimation pla and an eight bit adder for the most significant bits of the partial remainder. The topmost six bits of the adder result constitute an estimate of the entire partial remainder at each step. Doing eight bits of addition rather than just six significantly increases the precision of this estimate and does not increase execution time. The remainder estimate is then fed along with the four most significant divisor bits into the quotient estimation box. The pla output is a number in the range (-2 <= X <= 2) encoded into sign magnitude format to be stored in the quotient estimate latches. This value must also be decoded into a "one hot" format to be used in the divisor selection unit.

Unlike the multiply instruction, the divider loop is not pipelined; the quotient estimator and dividend accumulator operate as a single piece of logic. Two full clock phases are required to traverse the entire loop. The loops are controlled by the partial product master latch and therefore are timed to end on even clock phases. Each quotient estimate vector is a 68 bit quantity consisting of 65 result bits plus three rounding bits for each estimate. Thus 34 iterations of the divider loop are required to generate the full quotient vectors.

Note that the loop count figure shown above does not take into account the time required to convert the quotient estimate vectors into the final result. As mentioned previously, a two step process is required to obtain the final quotient. First, the two 65 bit final remainder vectors are sent to the fraction datapath to be subtracted in order to find the sign of the remainder. Then the quotient estimate vectors are sent to the fraction box to be subtracted. As seen in Figure 14, one cannot send all 68 bits of both quotient vectors to the fraction unit; the unit expects rounding bits from one operand only. Rather than enlarge the fraction box adder/subtractor for this case, the three rounding bits are subtracted before being sent to the fraction unit. The rounding bits result should also be decremented if the sign of the final remainder was negative. This is accomplished by using a two's complement adder for the subtraction and the complement of the partial remainder sign bit as the carry input. Thus, to compute the final result the fraction unit is sent two 65 bit operands, the three rounding bits and the carry output of the three bit subtractor. The carry output of the three bit subtractor becomes the carry input of the

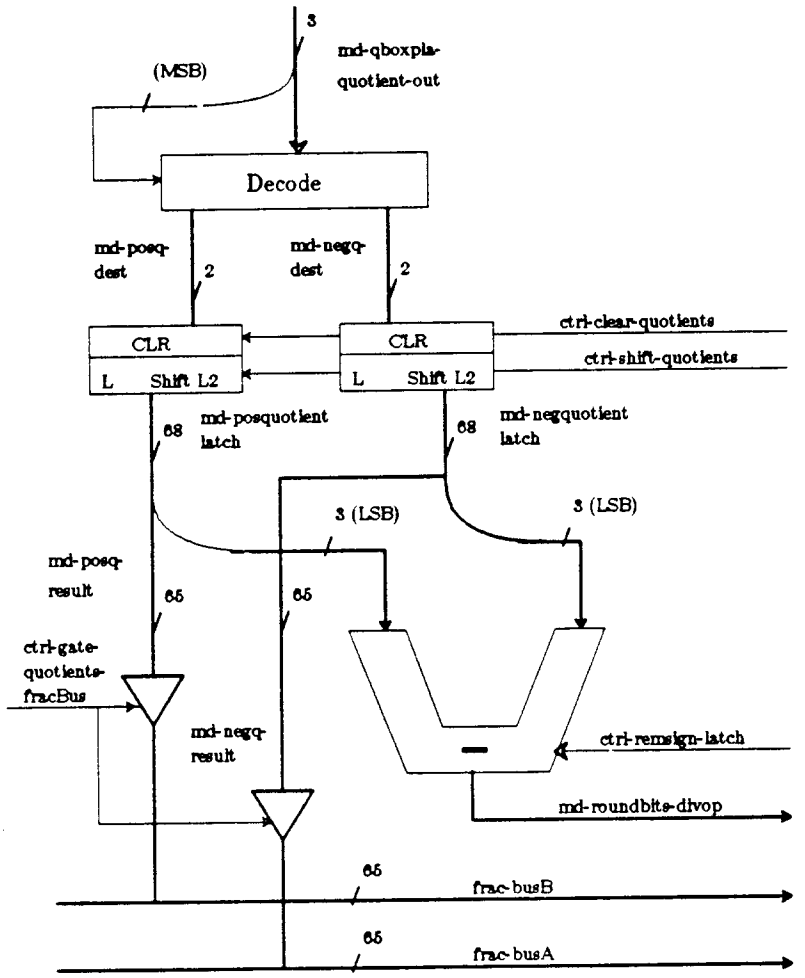


Figure 14. Quotient bit accumulation in divider.

fraction unit adder.

3.3.3. Exponent Datapath

The exponent datapath lies on the critical path of all FPU arithmetic instructions except for the data transfer operations. Each instruction requires the formulation of a preliminary exponent; each instruction except for compare requires that the preliminary exponent be adjusted for normalization to form the final result exponent. This final result must be checked for underflow or overflow. Also, the add and subtract instructions require an indication of the operand with the greater exponent as well as the exact amount of the exponent difference for operand alignment. As shown in Figure 15, each of these functions is realized with separate arithmetic components known as the difference

and result components.

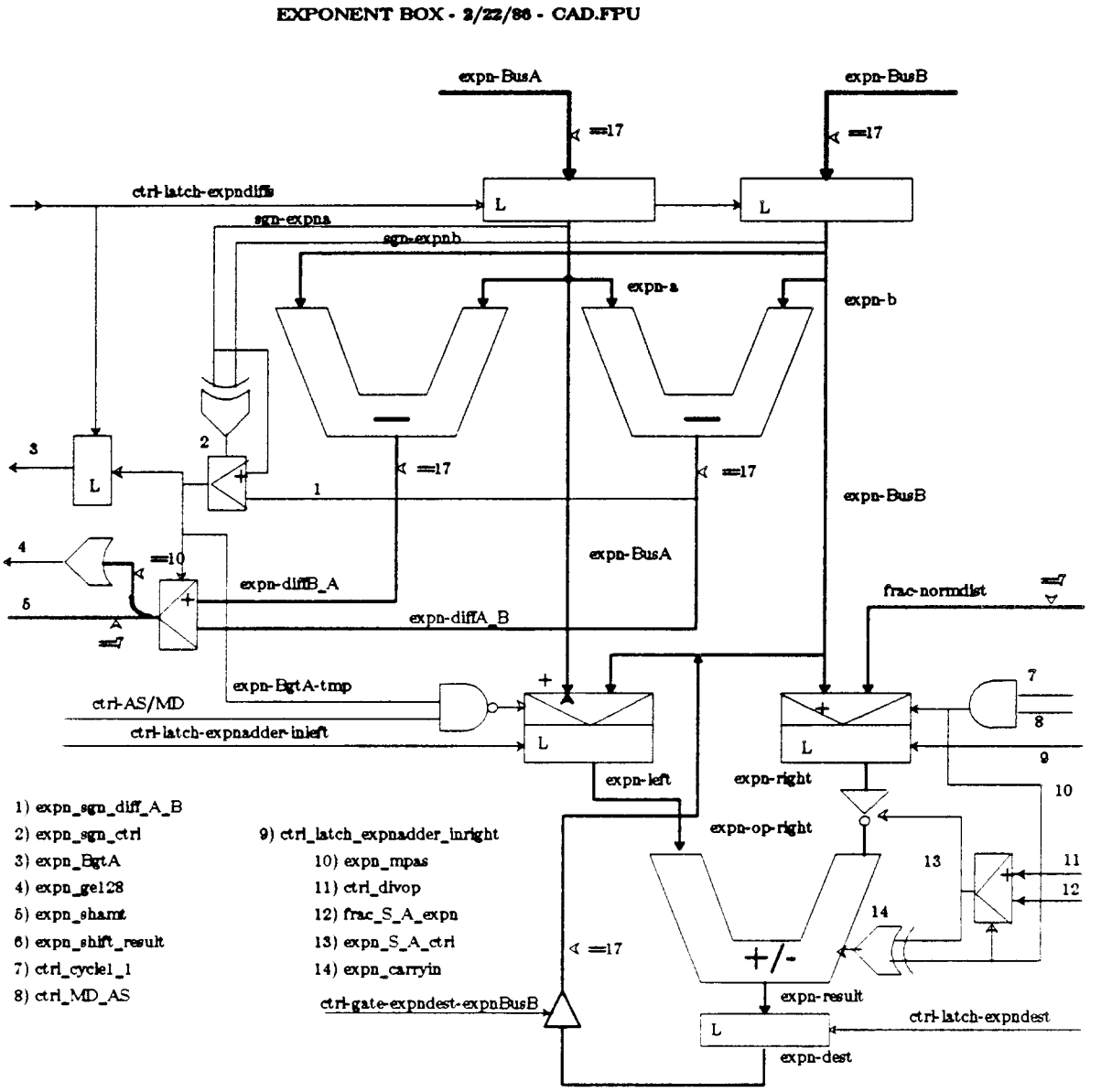


Figure 15. Details of exponent box.

Note that although exponent operands are 17 bits long, each of the elements in the datapath are 18 bits in length.

As mentioned above, the exponent difference section must find the greater of the two operand exponents and compute the positive difference between them. Since these operations must be done in little more than one phase, this section is implemented using two parallel subtractors and selection logic. The exponent difference is sent out in two forms;

a 7 bit positive difference vector for differences from 0 to 127 and a signal to indicate whether the difference is 128 or more. Also, a signal indicating the greater exponent is latched and sent to the fraction box.

The result section consists of a single adder/subtractor with control logic. This datapath is itself straightforward, however there are subtleties in its use. For instance, recall that all exponents are converted to 17 bit two's complement numbers with a bias of -1; this bias must be taken into account when computing intermediate results. Since the bias is now only -1, one can keep the correct bias for intermediate results by manipulating the carry input of the two's complement add or subtract. To see this, recall that the result bias (without adjustment) of an add operation would be -1 + -1 = -2, while for a subtract it would be -1 - (-1) = 0. Note that in the exponent difference section, the result bias of the subtraction (zero) is correct; the true difference between the exponents is needed.

In the add and subtract instructions the result exponent can be computed in one ALU step by adjusting the greater exponent by the normalizing distance. However the multiply and divide instructions as well the two converts require an additional add or subtract to generate a preliminary result. That result is fed back to the adder/subtractor using the datapath busses and input latches for the final adjustment. For all instructions, a special value must be used for the final result exponent if the result fraction is zero. Each pass through the exponent ALU requires one clock phase.

Finally, logic must be added to the result latch to detect overflowed and underflowed exponents. These signals help form the result exception bits for the instruction. The detection logic adds almost an extra phase to the result exponent computation. Because the exponent is the last data portion computed, result exception detect adds to the execution time of any instruction that requires it.

3.3.4. Sign and Type Determination

The sign determination unit consists of combinational logic that must correctly find the sign of the result of all arithmetic operations. Fortunately, the equations for determining the result sign of the various instructions are relatively straightforward. In fact, the result sign determination for the convert and data transfer operations is done immediately after reading the input sign from the register file by transferring the input sign to the result sign latch.

The result sign for multiplies and divides is not much more difficult; it is the exclusive-or of the operand signs. For the instructions add and subtract, the result sign is a function of both the operand signs and their relative magnitudes. The relative magnitudes of the operands are determined from the magnitudes of their exponents, the sign of the fraction's intermediate result and the zero fraction detect signal from the fraction box. The arrival of these signals from the fraction box determines the timing of this unit. For simplicity, all result signs not computed immediately (including multiply/divide) are latched in phi2 of the third execute cycle.

Like the sign determiner, the type datapath consists primarily of combinational logic. Besides determining the result data type of all arithmetic operations, this datapath also contains the logic to generate the new FPSW for each operation. Note that the result data type is not a function of the operand types; instead it is formed from the fraction box zero detect signal. The operand types go directly into the new FPSW and also pass through logic to determine whether the operand will be correctly handled by the hardware. An exception is posted in the FPSW and the result write is cancelled if either of the operands are of illegal type. The signals for the other exception and compare bits in the FPSW come from the fraction and exponent boxes. Still more logic, however, is required to gather these signals and ensure that the new FPSW is written correctly. Because the result exception timing is critical, part of this evaluation time will be overlapped with the final result write.

3.3.5. Load/Store Datapath and Register File

Whereas all of the above mentioned sections are devoted to arithmetic operations, only one datapath each is needed for all load and store instructions. Data flows directly from the chip data pins to the register file in the load datapath; flow is in the opposite direction for stores. Access to both the register file and data pins is controlled by placing latches at each end of both datapaths. The latches also allow consecutive memory instructions to execute. Although much of the logic for the load and store datapaths is similar, the two datapaths must be kept entirely separate. This is because the timing of a load operation immediately followed by a store causes the two datapaths to be active simultaneously. Further details on the timing of these instructions is given in the section on the memory control unit.

The function of the load datapath has already been described in the section on instruction execution. The logic required to perform this function, illustrated in Figure 16, is implemented in the simplest manner possible.

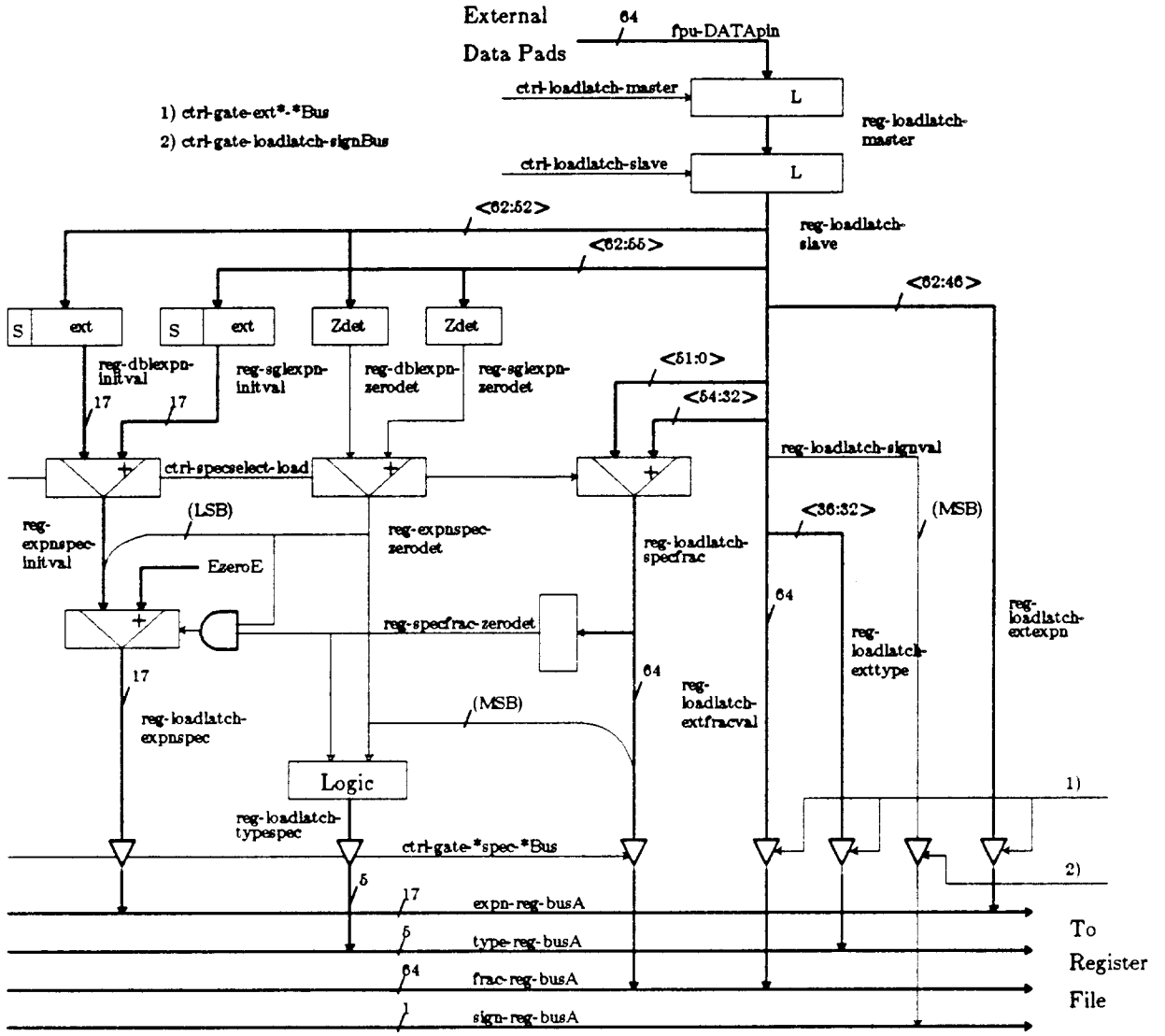


Figure 16. Datapath for load instructions.

Each portion (fraction, exponent, sign and type) of the input data flows through a separate logic path. Furthermore, separate nodes implement the different paths that a given data portion will take for different types of loads. These paths are brought together by multiplexors controlled by signals indicating the exact type of load being executed. Also, the datapath logic nodes currently extend from the input slave latch and gate directly onto the register file busses. While this particular description style simplifies

debugging, it does not necessary reflect the structure of the layout implementation.

The store datapath is organized differently, using separate input latches for each data portion. As illustrated in Figure 17, the datapath logic is contained between the input latches and the external data bus.

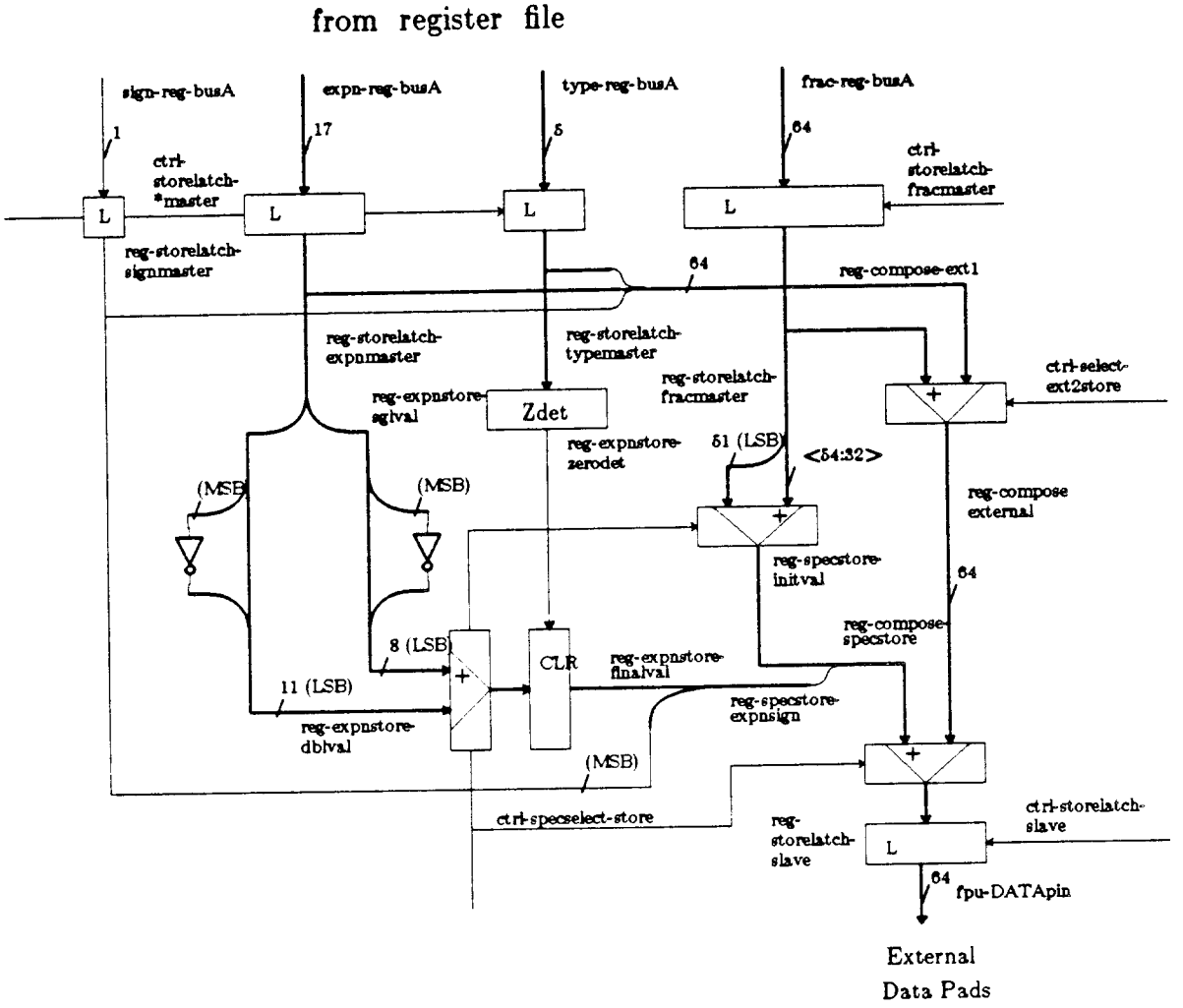


Figure 17. Datapath for store instructions.

On store instructions, conversion is needed only for the exponent; this conversion is the inverse of the conversion in the load datapath. Implementing the rest of the packing logic requires only a little more than one node per type of store instruction. Finally, for simplicity the external data pins are implemented in Slang not as a bus but as a latch with a multiplexor.

The register file is a high level bi-directional component, making it the most difficult datapath component to implement in Slang. Separate Slang nodes must be used for each portion (fraction, exponent, sign and type) of the register file. Each node itself is implemented as a generalized Lisp array containing the particular type operand in internal format. The lack of bi-directional Slang nodes is overcome by having the register file nodes themselves active only on writes. An intermediate latch is used to hold the data value that is read or written on any operation. These latches are multiplexed so that data for both reads and writes may go through them. Separate intermediate latches are needed to handle transactions on each of the two register file busses.

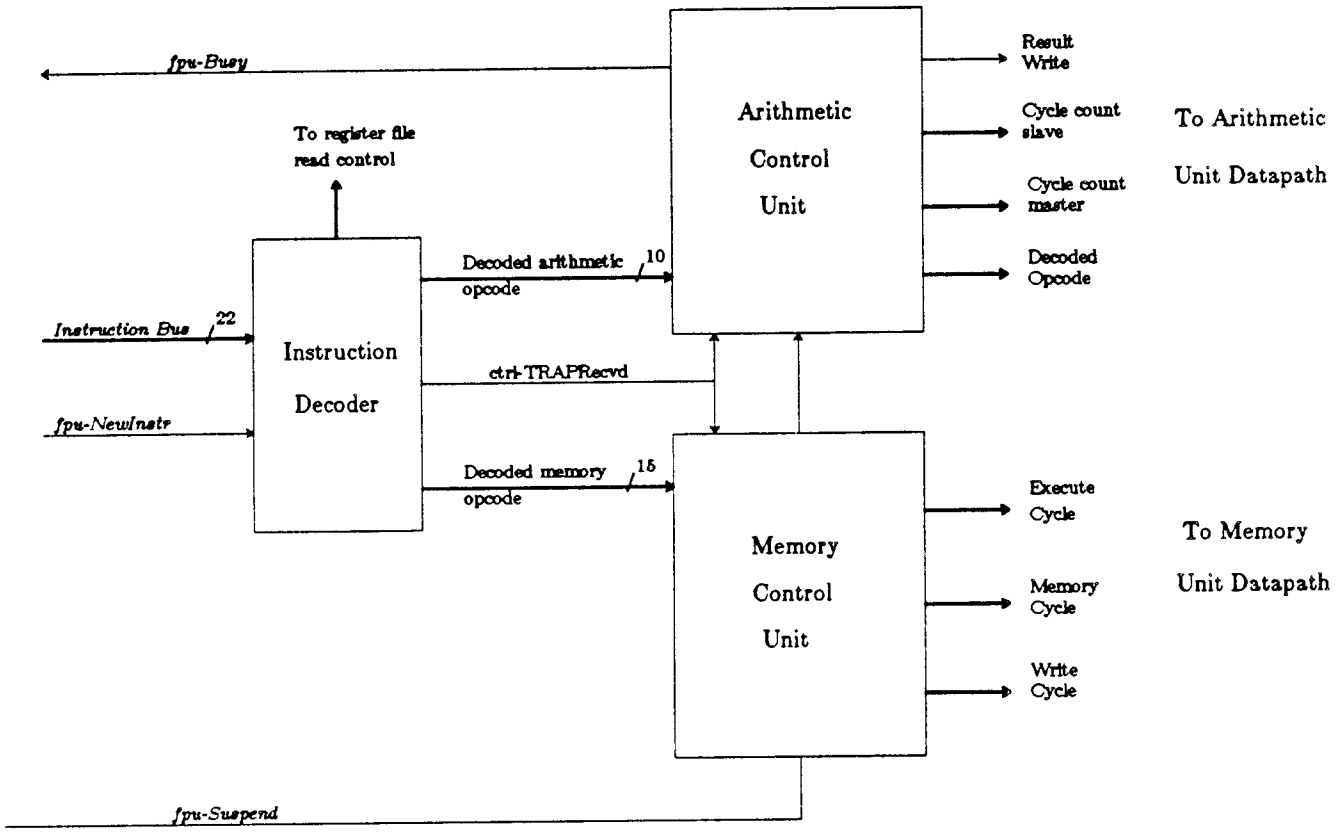
4. FPU CONTROL ORGANIZATION

4.1. Introduction and Overview

The objectives of the FPU chip control are to sequence through the memory and arithmetic instructions handled by the chip and to interface with the SPUR central processor. The instruction sequences are very simple, and the mechanics of the CPU interface are simplified by using the same four phase clock for both CPU and FPU. Complexity is introduced, however, by the requirements of the CPU-FPU interface and by chip performance criteria. For example, whereas FPU arithmetic operations are sequenced serially, the memory operations may be issued at the rate of one per machine cycle. Also, FPU memory and arithmetic instructions may proceed in parallel. Finally, all FPU instructions may be suspended or trapped (killed) during their execution. These constraints are the primary determinants of the FPU control structure.

An overview of the control section for the FPU chip is shown in Figure 18. This section consists of separate control units for the memory and arithmetic instructions; both these units are driven by the instruction decoder unit. Note that although the memory and arithmetic control units both rely on the instruction decoder, they also receive control information directly from the FPU-CPU interface. The two kinds of FPU instructions use separate control units so that they may operate in parallel and so that different implementation structures may be used to meet the differing requirements of the two units.

By far the simplest portion of the control is the decoder unit. The decoder unit is always active, monitoring the co-processor instruction bus and the fpuNewInstr signal.



NOTE: *Italice* denote signals for FPU-CPU interface.

Figure 18. Overview of FPU Control.

On every machine cycle this unit must determine whether a valid FPU instruction has been issued and if so send a decoded version of that instruction to the appropriate control section. It also decodes the special trap opcode directly into a signal that is sent to the other two control units. In all other cases (non-FPU opcode, suspension etc.) the unit takes no action.

Since FPU memory operations are issued once per machine cycle, the memory unit control receives input from the instruction decoder on every cycle. These memory operations are considered to be part of the CPU pipeline, furthermore the CPU participates in these operations by supplying the memory address. For these reasons, the timing of FPU memory operations matches that of the CPU; the memory control unit also uses an opcode pipeline for its control structure. Each station in the pipeline is active for exactly

one machine cycle; the stations themselves contain a decoded version of the input opcode and the value of the destination register for load instructions. The contents of the pipeline stations control the activities of the load/store datapath directly. Under ordinary conditions this pipeline operates continuously; the suspend and trap signals cause the pipeline to stall or clear respectively.

Whereas the memory control unit handles a pipeline of fixed execution time instructions, the arithmetic control unit only needs to sequence a single instruction. Although the instruction sequences themselves are very simple, the instructions have varying execution time and may be suspended or trapped during a portion of their execution. The structure of this unit therefore consists of a simple machine cycle counter with a separate state machine for handling suspend and trap conditions. This division of control simplifies the unit and also allows arithmetic instructions that have been suspended to actually continue execution in the FPU datapath. This is done by letting the cycle counter continue during the suspension while delaying the result write using the supervisory state machine. Also, the state machine can take input directly from the instruction decoder stage without interfering with the operation of the chip datapaths. This allows the write cycle of the active FPU instruction to be overlapped with the decoding of the next instruction.

4.2. Details of Operations

4.2.1. Operation Timing

Although the memory and arithmetic control machines mentioned above are quite different in structure, they use the same operation timing scheme. The timing of control machine transitions is dictated by the fact that the fpuSuspend signal arrives in the fourth phase (phi4) of a given cycle. For this reason the opcode pipeline and arithmetic state machine cannot change state until phi1 of the following cycle. Control signals that depend on this state will not be stable until the next phase; this leaves the problem of how to control events that must occur on phi1 of a cycle. The solution is to use master/slave latches for all state holders and to generate appropriately timed control signals. On a given cycle, all events occurring in phi2 are controlled by the master latch while the slave is loaded. The slave latch controls signals for all other phases, including phi1 of the following cycle. This provides an effective solution for all execution cycles except the very first. As shall be seen, events in phi1 of the first execution cycle is controlled directly by

the instruction decoder logic.

4.2.2. Instruction Decoder

The instruction decode unit must monitor the external instruction bus on every cycle and begin the execution of the instructions intended for the FPU. Since the instruction bus changes on phi3, the decoder has little more than one phase (until phi1 of the next cycle) to perform these functions. As seen in Figure 19, this unit consists of a simple opcode decoding filter with logic to decide whether to accept the decoded instruction.

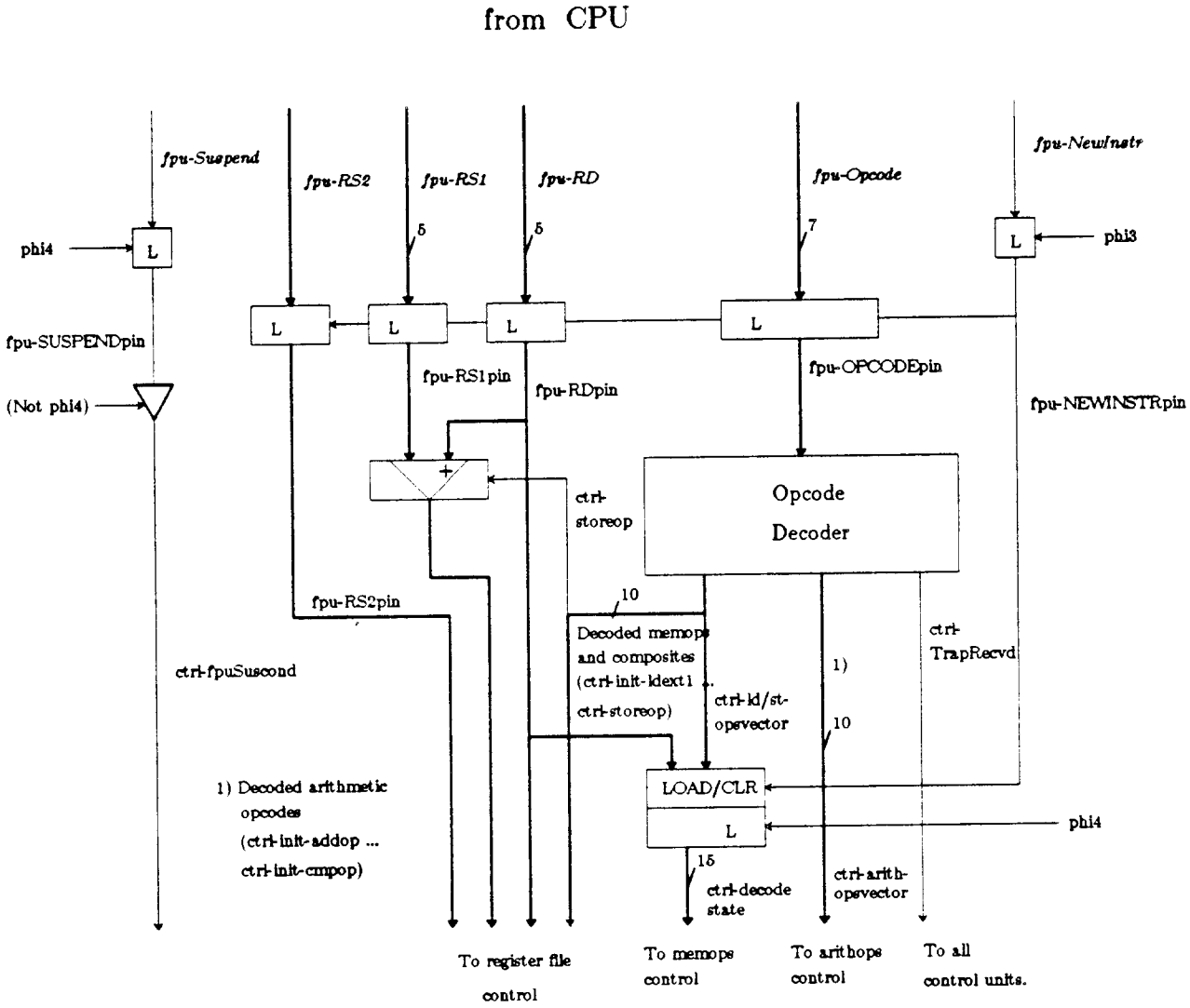


Figure 19. Details of Instruction Decoder Unit.

The logic to begin the execution of an instruction must run in parallel with the opcode

decoder in order for the unit to meet its strict performance criteria.

To further increase performance, the decoder logic itself attaches directly to the opcode pins latch and takes no other signals as input. The decoding logic generates a trap received signal directly from the appropriate opcode; this allows that signal to take effect immediately. In addition, it generates two vectors of decoded opcode signals; one each to represent the type of memory or arithmetic operation encountered. These vectors will contain no asserted signals if a trap or non-FPU opcode is encountered. The fpuNewInstr signal directly controls the latching of the instruction bus; no other signal is needed.

The conditions under which a decoded FPU instruction will begin execution depend upon whether it is a memory or arithmetic operation. A memory operation will be accepted only if the FPU is not suspended (i.e. fpuSuspend disasserted). On the other hand, an arithmetic operation begins whenever the arithmetic unit is not busy (fpuBusy disasserted). Both machines will simply refuse to latch the output of the opcode decoder if the specified conditions are not met.

In addition to initializing the control machines, the decoder unit must control the reading of the register file to begin an instruction successfully. Because register file decoding and reading take place in phi1, the decoder unit must send the proper register specifier to the file in phi4. Note that while the store operation reads from the destination register number, all arithmetic operations read from the two source registers. Thus, at least a partial opcode decoding is required to send the correct register specifier. The actual register file read takes place regardless of whether the instruction that was decoded is ready to execute. Rather than inhibiting the read operation itself, the arithmetic control unit will not connect its busses to the register file busses if the arithmetic datapath is busy. The memory unit behaves similarly by refusing to latch the output of the register file if the instruction pipeline is suspended. Register file reads are inhibited this way because it is easier from a circuit implementation point of view to control a few latches and gates than the entire register file.

4.2.3. Memory Control Machine

Figure 20 illustrates the four stage memory control unit pipeline along with the exact format of the contents of each opcode latch. Each of the pipeline latches is clearable, in addition the second and third stages use master/slave latches to allow feedback for each

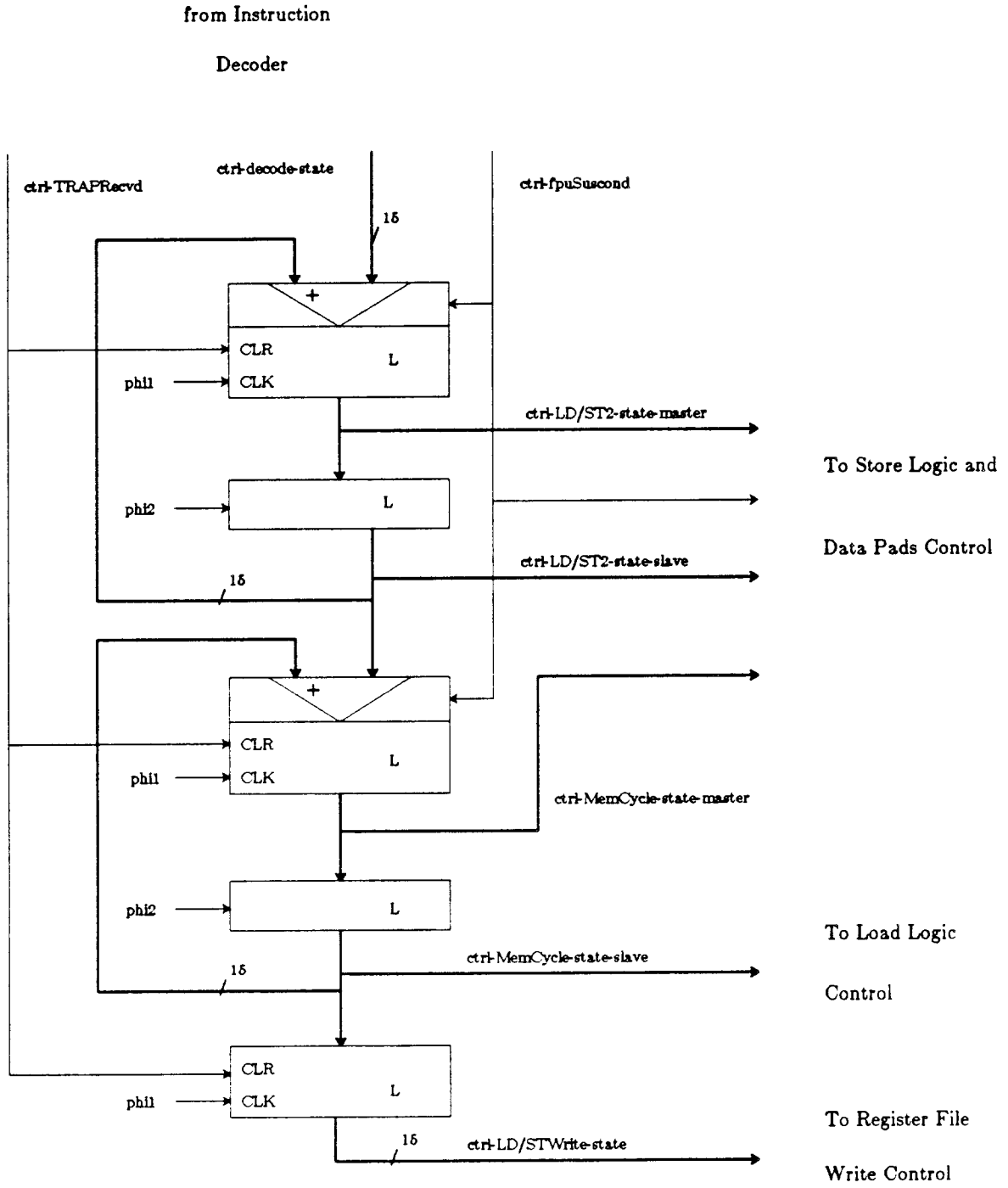


Figure 20. Opcode Pipeline for Memory Operations.

stage. This feedback allows the execute and memory cycles of the pipeline to be repeated on a cache miss on either CPU or FPU memory operations. Also, each of the stages is cleared whenever a trap is signalled. The last stage (for the write cycle) is different in

that it may neither be trapped nor suspended. It takes either the output of the previous stage or a zero if the previous stage is suspended.

Because the CPU handles addressing for all FPU memory operations, load operations do not require any action until the third (memory) cycle. The external data latch is loaded on phi3 whenever the control pipeline indicates a load operation is present. If a cache miss occurs, the fpuSuspend signal will keep the control information for the load in the memory stage latch and zero out the write stage latch. In this event, the contents of the external data latch are not passed on through the load logic and are overwritten on the next cycle. Thus, the load operation may be controlled successfully without monitoring the cache hit/miss signals directly.

The control of the store operation is complicated somewhat by the control of the external data pads for output. These pads must be driven starting in phi1 of the third cycle of a store and held until the cache indicates a hit. Because the fpuSuspend signal arrives at the end of the previous phase, it is not possible for the control machine to decide whether the memory cycle should be repeated and drive the pins accordingly. Therefore, the driving of the pads in phi1 is controlled by the information in the second pipeline stage only if the pipeline is not suspended. In all other cases, the pads are controlled by the memory cycle stage and the cache hit signals. The cache hit signals are used to condition the memory cycle stage because the fpuSuspend signal is not disasserted until one phase after a cache hit.

Although not strictly under the control of the memory unit, it is appropriate to discuss the register file control structure here. The control of the register file is greatly simplified by the fact that the file may be addressed as an array. Independent controls are still required for reading and writing, and separate signals are needed to handle transactions on each of the two register file busses. While arithmetic operations access all portions of the register file simultaneously, the two external format memory instructions access different portions of the file. For this reason, independent controls for each portion of the register file are used only for reads and writes to and from bus A.

4.2.4. Arithmetic Control Machine

The sequencing of arithmetic instructions is handled by the state machine and machine cycle counter illustrated in Figure 21. Both these sequencers begin operating on

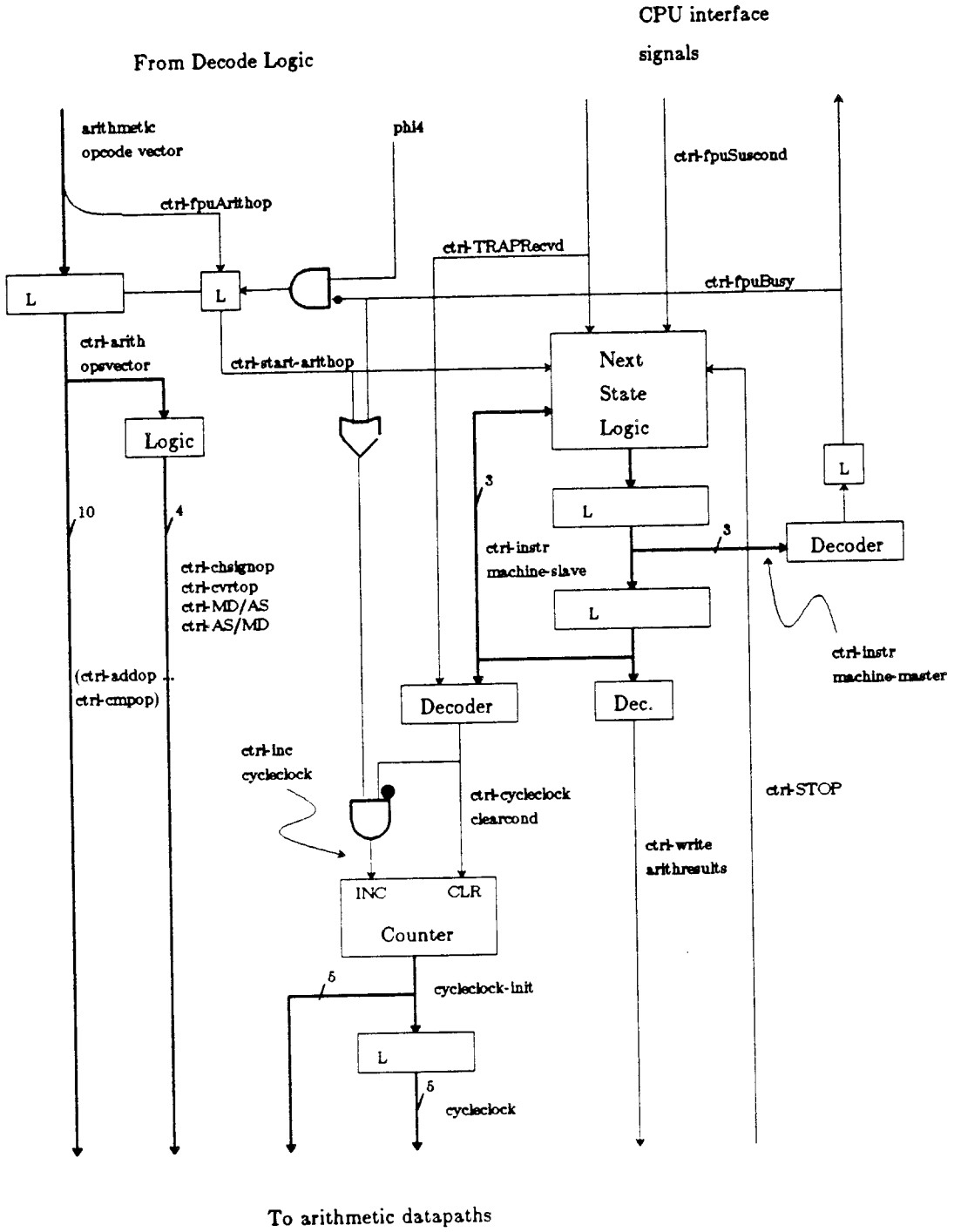
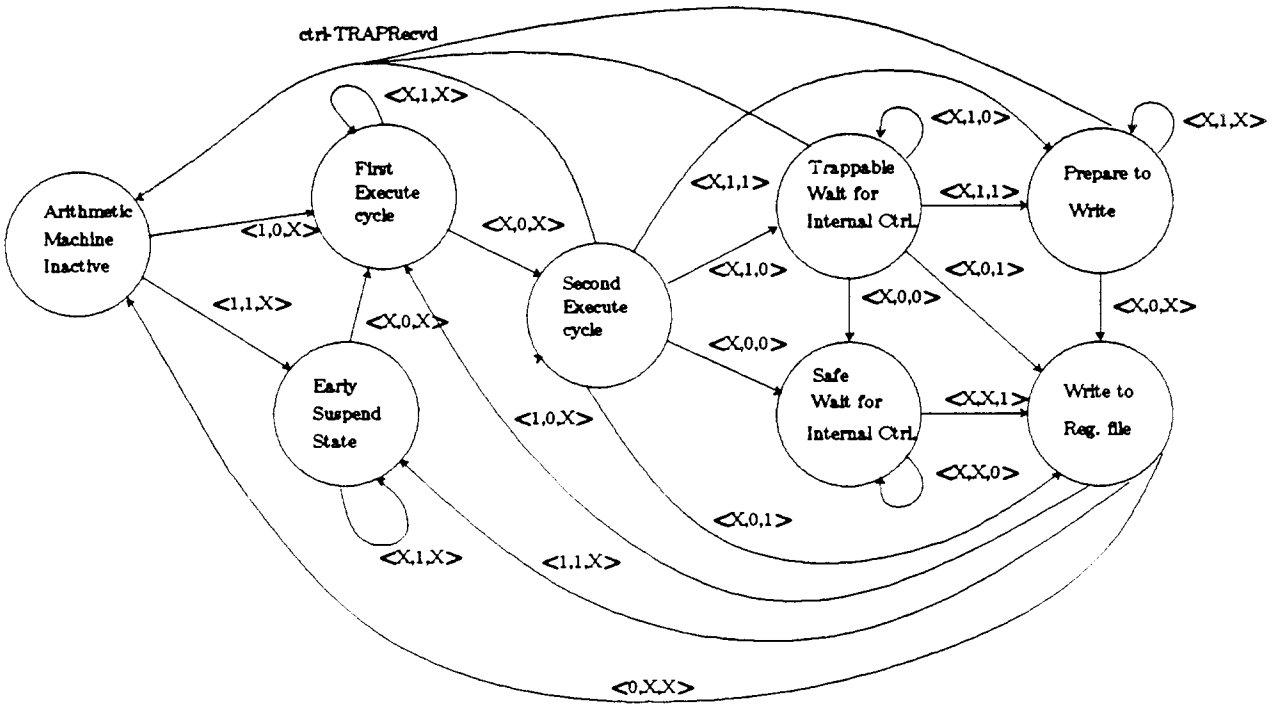


Figure 21. Organization of Arithmetic Control Unit.

the first execution cycle of an arithmetic instruction. The cycle counter controls the datapath directly throughout the execution of the instruction and signals the arithmetic state machine when all results have arrived at the destination latches. The state machine in

turn monitors the trap and suspend signals during instruction execution while controlling the writing of results and the fpuBusy line. The machine will never stop the execution cycle counter on a suspension but instead will delay the result write. In addition, the counter will be reset by the state machine if the instruction finishes or is trapped.

As seen in Figure 22, the arithmetic state machine is implemented using eight states.



Transition Vector := < ctrl-start-arithop, ctrl-fpuSuspend, ctrl-STOP >

NOTE: Signal *ctrl-TRAPRecvd* overrides Transition Vector.

Figure 22. Arithmetic State Machine Diagram.

Since the execution of the FPU instruction continues during suspension, an instruction may finish before the CPU instruction pipeline has proceeded two cycles. Because of this, separate states are required to sequence the first two non-suspended execute cycles because all instructions may be trapped during this period. In addition, an extra "early wait" state is needed for those instructions that are suspended just after the instruction cycle.

After these cycles have been completed, non-suspended instructions will always complete and write results without trapping. These instructions move to the "safe" state or

directly to the write state if they already have completed. Suspended instructions must remain in a trappable state until the pipeline suspension has released. An extra state (the "prepare to write" state) is necessary for instructions completed during suspension to shut off the machine cycle counter immediately upon instruction completion. Finally, the machine may transit from the write cycle to inactivity or to the first execute cycle.

Note that this final transition allows the overlapping of the write cycle of an FPU instruction with the fetch cycle of the subsequent one. This overlapping also affects the implementation of the machine cycle counter, moving the clear signal for the counter to phi4. Ordinarily, both the state machine and cycle counter are master/slave devices that make their transitions as described in the section on control timing. Except for this complication of the clearing condition, both portions of the sequencer are implemented in a very straightforward manner.

The rest of the arithmetic control unit is distributed among the various datapath blocks. Each block receives the decoded opcode vector, both the master and slave of the cycle counter, and all four clock phases. Almost all of the detailed control signals are simple combinations of this state information. The only notable exception to this is the multiply/divide accumulator loop which uses an extra flip-flop as a sequencer. Because the loop is active for many clock cycles, the extra state bit proved easier than a partial decoding of the cycle counter to implement in Slang.

5. FUNCTIONAL TESTING

5.1. Introduction

While the FPU hardware description is useful as documentation, its primary purpose is to verify the chip's logic and circuitry. The first step in this verification process is the testing of the description itself. Fortunately, a ready made suite of floating point vectors is available for testing the arithmetic of any implementation of the IEEE standard. These test vectors (collectively known as the IEEE test suite) were adapted to serve as input vectors for the functional testing of this chip. The adaptation process required both a program to customize the content and formatting of the IEEE test suite and a set of functions to run the resulting diagnostics through the description. The format customization step is run separately from the simulator and generates files containing the test vectors

and the expected results. The simulator in turn has been enhanced with special functions to generate SPUR FPU diagnostics from these files, drive the FPU description, and check the results given by the hardware.

5.2. IEEE Format Converter

As mentioned above, the IEEE test suite customization step is run as a separate processing step outside the Slang simulator. It converts the operand, precision and rounding mode specification of every test vector into an intermediate format which can be easily converted to SPUR diagnostics. Because the chip implements a subset of the standard, the expected result vectors must also be adjusted to reflect the result and FPSW generated by the hardware. The details of this conversion step are beyond the scope of this report, however a description of the details of the input and expected result file formats is in Appendix B of this document.

5.3. Slang Testbed

The Slang hardware description language has very few functions built in for running preformed diagnostics. Thus, all of the functions that form the testing environment for the FPU description were written from scratch. This homemade testbed consists of functions for initializing and driving the simulator, transforming the input test vectors into FPU instruction sequences, and checking the hardware results for the sequence against the expected results. Besides reading the input and expected results files mentioned above, the testbed generates a error file that documents the success or failure of each test vector. In addition to these functions, extra Slang nodes have been added to the hardware description; these nodes drive the CPU-FPU interface, supply instructions and data to the chip and collect the result data. These nodes provide the execution environment for the FPU chip and totally control the simulation once it has started.

Currently, a very simple execution model is used for testing the chip. A translator function converts test vectors into a sequence of instructions to load the operands of the test, execute the target arithmetic instruction and store the result. A single node (called the "executor") drives the interface, supplying the instructions one at a time to the chip. Another node (the "verifier") detects the end of every instruction and collects the results of every store instruction. At the end of every sequence, the verifier node calls a check function to verify the results, and the executor node calls the translator function to form a

new sequence. This process is repeated until no more test vectors are available.

The above model provides the simplest method for testing the validity of the arithmetic performed by the chip. However, it does not test the capabilities of the chip control such as traps, suspends and parallel execution. Furthermore, still other testing capabilities will be needed to use the FPU description as a layout verification tool. All of these capabilities may be achieved, however, by making modifications and possibly additions to the two nodes described above.

6. SLANG IMPLEMENTATION NOTES

The previous sections of this report have illustrated the flexibility of Slang as a hardware description language and as a simulation tool. However, that flexibility has caused several problems with the structure of the chip description. The worst of these problems is the absence of Slang constructs for coding separate chip component descriptions as separate modules. Because a Slang node may connect to any other node on the chip, it is very difficult to determine from Slang the interfaces between major components of the chip. The resulting lack of modularity reduces the readability of the entire description and makes incremental testing of chip components almost impossible. Other problems were encountered during the implementation of particular components of the description; these are described separately in the sections that follow.

6.1. Datapath

Several difficulties arose while writing the Slang description for the datapath components. The most notable of these is the lack of any support in Slang for numeric values containing more than 32 bits. Because of this, all floating point fractions in this description are represented as three element lists. Although many Slang functions (such as multiplexing) are data type independent, special arithmetic functions had to be written in Lisp to handle the new data type. These functions had to handle Slang constructs such as UNK as well.

Another related problem was the inability to explicitly declare the size of Slang node values. While the correct size for a given node could always be maintained by special functions, the correct size for a given numeric value is often impossible to determine from

the description. Thus, portions of the Slang description for the load/store and fraction datapaths are confusing to read because operands change size between nodes in an opaque manner.

Finally, this description contains its own model for all busses, replacing a Slang bus model that was difficult to use and virtually undocumented. The FPU bus model currently takes into account precharging and gives error messages when glitches affect the bus or multiple values are driven on the bus simultaneously. It is also very easy to determine from the model which signals gate onto a particular bus. Unfortunately, Slang does not support the notion of bi-directional signals. Thus, it is difficult to determine which components are taking values from the bus at a given time.

6.2. Control

The hardware description of the FPU control section relies heavily on homemade Slang primitives for latching and timing. In this description, a latch node is implemented as a multiplexor whose select signal feeds its output back to the node. The logic nodes that depend on that latch will not evaluate when the latch control signal is inactive because the actual data value in the latch does not change. Thus, all latches (even those in data loops) may be controlled successfully by simply gating the control signal with the appropriate clock phase. The clock phases themselves are derived from a special node called masterclock which is guaranteed by Slang to be the first event scheduled in each evaluation sequence. The function of the masterclock node is programmer definable; here it is a mod 4 counter which is then decoded into four separate clock phase signals.

Given the primitives mentioned above, the implementation of the control section description is relatively straightforward. The only difficulty encountered in the control section concerns the timing of the register file and bus operations. A successful read or write requires that the exchange of data between these components occur in the proper order within one clock phase. This is done by manipulating the depends clause for these nodes and by adding extra nodes to delay the evaluation of the register file control.

The pipeline for the memory control unit was implemented in a manner similar to portions of the datapath description. The arithmetic state machine consists of only one node. This node contains a single cond clause (LISP equivalent of case statement) where each condition corresponds to a transition in the state machine. Finally, all of the

combinational logic in the control section is implemented using simple AND, OR and Equal (decode) clauses.

7. CONCLUSIONS

The SPUR floating point unit increases the performance of IEEE arithmetic by efficiently implementing the most common arithmetic operations in hardware. To further increase performance, the execution of arithmetic instructions may be overlapped with CPU instructions as well as FPU load and store operations. This requires an internal organization containing separate datapath sections for memory and arithmetic operations; the register file is shared between the datapath sections. Within each of the sections, separate datapath components are devoted to the different FPU operand data portions. Many of the datapath components participate in more than one instruction sequence, however. The control structures for the chip are also divided into arithmetic and memory sections; these sections both rely on the instruction decoder and CPU-FPU interface for input signals. The two structures are organized differently to meet separate instruction control requirements.

Currently, the chip hardware description is being tested using the IEEE test suite for input vectors. Later, the description will be used alongside a switch level circuit simulator to verify the chip layout. In addition, the debugged version of the description will serve as a guide in the final datapath layout and control generation steps for the chip. Thus, the same hardware description will serve as a testing and as a documentation tool.

This adaptation of the same chip description to different purposes is made possible by the flexibility of the Slang hardware description language. This flexibility also has resulted in a disturbing lack of structure in the chip description, however. Furthermore, the coding of the description was hampered by the lack of modularity and arithmetic support for large vector operations. Overall, Slang should be improved significantly in these areas before another large chip description effort is undertaken.

As with all large projects, this chip description was by no means a solitary effort. As the two faculty advisors for this project, Randy Katz and Carlo Sequin have provided the leadership necessary to guide this work through to completion. George Taylor provided the algorithms for the chip's arithmetic. His experience with floating point hardware

resulted in algorithms that were efficient in both time and chip area. Finally, much of the information input to this description has come from the full time members of the SPUR FPU project group, B.K. Bose, Paul Hansen and Corinna Lee. In a sense, this hardware description is a fusion of their efforts on the implementation, interface and architectural aspects of the chip respectively. Also, it is their responsibility to take the hardware description and use it in their effort to complete the implementation of the chip.

8. REFERENCES

[FoD83] J. K. Foderaro and K. S. V. Dyke, *SLANG Slinger's Cyclopedia*, UC Berkeley internal working document, 1983.

[HaK86] P. M. Hansen and S. I. Kong, *SPUR Coprocessor Interface Description*, UC Berkeley internal working document, May 23, 1986.

[HEL85] M. D. Hill, S. J. Eggers, J. R. Larus and G. S. Taylor, SPUR: A VLSI Multiprocessor Workstation, Report No. UCB/Computer Science Dpt. 86/273, Computer Science Division (EECS) University of California, Berkeley, December 1985.

[IEE85] IEEE, IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, Order number CN953, 1985.

[Kat85] R. H. Katz, SPUR Architecture Design Rationale, *Proc. of CS292i: Implementation of VLSI Systems*, R.H. Katz, Ed., University of California, Berkeley, September 1985. Computer Science Division Technical Report UCB/Computer Science Dpt. 86/259.

[Lee86] C. G. Lee, *Description of the SPUR Floating-Point Unit*, UC Berkeley internal working document, March 17, 1986.

[Tay85] G. S. Taylor, Radix 16 SRT Dividers with Overlapped Quotient Selection Stages, in *Proceedings of the Seventh Annual IEEE Symposium on Computer Arithmetic*, June 1985, 64-71.

Appendix A

Sequence of operations for SPUR FPU Instructions.

The following is a table for each instruction currently supported in the SPUR FPU unit.
The table contains the exact sequence of operations required to complete that instruction as well as the control signals and datapath components required to perform that operation.
These tables is not intended to describe the rationale of the FPU datapaths or control.
It does, however, illustrate the timing and control requirements on the datapath components, and it will give an illustration of the use of shared resources (reg. file, busses, datapath components) in the FPU.
The sequences for the actual instructions are given in three files (one for LD/ST, one for add/sub, one for mul/div).
This file contains information on all uses of the register file and busses.

REGISTER FILE ACCESSES

TIME/SIGNALS/COMPONENTS	ACTION/COMMENTS
1-4, phi4,fpu-RS1pin, ctrl-regnumbA	Send RS1 to register file for reading onto A bus. Must be done in parallel with operation decoding.
1-4, phi4,fpu-RS2pin, ctrl-regnumbB	Send RS2 to register file for reading onto B bus. Must be done in parallel with operation decoding and is therefore done without knowing whether an actual read will take place.
2-1, ctrl-read-regsA, {frac,expn,sign}-regfile	Read value onto register A bus. Should read on stores and arithmetic operations and if not suspended.
2-1, ctrl-read-regsB, {frac,expn,sign}-regfile	Read value onto register B bus. Should read only on arithmetic operations if not busy and if not suspended.
4-2, ctrl-regnumbA ctrl-latch-regnumbA	Load register decode latch for write from A bus. The register number comes the load/store opcode state vector!
4-3, ctrl-write-regsA, {frac,expn,sign}-regfile	Write value from register A bus. Should write only on stores.
??-2, ctrl-regnumbB, ctrl-latch-regnumbB	Load register decode latch for write from B bus. The register number comes a temporary latch where the destination register number is held throughout the arithmetic operation (ctrl-regnumbTMP).
??-3, ctrl-write-regsB, {frac,expn,sign}-regfile	Write value from register B bus. Should write only in the write cycle of arithmetic operations.

COMMENTS:

1) Register file is arranged such that all reads occur in phi1 and all writes occur in phi3.

2) There is no conflict because both sources of a read (store & arithmetic operation) occur in the same cycle.

3) Although writes from arithmetic operations and loads can happen in the same cycle, they occupy different busses. This implies a dual-port write register file.

BUS ACCESSES

TIME/SIGNALS/COMPONENTS	ACTION/COMMENTS
2-1,ctrl-read-regs{A,B}, {frac,expn,sign}-reg-{a,b},	Read out contents of register file onto register file busses.

{frac,expn,sign}-reg-bus{A,B}.	
2-1,ctrl-gate-regs-busses, {frac,expn,sign}-reg-bus{A,B} {frac,expn,sign}-bus{A,B}.	Connect register file onto arithmetic unit busses on arithmetic operations.
2-3,ctrl-gate-fracb-PPSslave, frac-b,md-PPS-slave,frac-busB	On divides, route dividend into partial remainder latch to start operation.
3-1,frac-intermed,frac-busB ctrl-gate-fracintermed-fracBusB md-compmcd-latch.	Route complemented dividend,multiplicand to selection latch on multiplies/divides.
4-3,reg-loadlatch-slave, ctrl-gate-loadlatch-{frac,expn,sign}Bus, {frac,expn,sign}-reg-busA.	Gate different portions of a load operation onto register file busses for write to register file.
{6-3,20-1},md-PP{S,C}-result, ctrl-gate-PPS/C-fracBus, frac-bus{A,B},frac-addleft, frac-right.	Gate results of multiply and partial remainders of divide for final accumulation in fraction box.
{7-3,22-1},expn-busB, ctrl-gate-expndest-expnBusB, expn-dest,expn-right.	On multiplies and divides, route initial result exponent back to exponent ALU for computation of final result.
21-1,ctrl-gate-quotient-fracBus frac-bus{A,B},md-posq-result, md-negq-result,frac-addleft, frac-right.	Gate quotients to fraction for final accumulation on divides.
??-3,ctrl-write-arithresults, ctrl-gate-expndest-expnBusB, {frac,expn,sign}-dest, {frac,expn,sign}-busB.	Place all arrithmetic results on bus B to be written back to register file.
??-3,ctrl-write-arithresults, {frac,expn,sign}-busB. {frac,expn,sign}-reg-busB.	Connect arithmetic unit busses to register file busses on register file write.

Sequence Table for Add/Subtract Instructions

ADD/SUBTRACT

TIME/SIGNALS/COMPONENTS	ACTION/COMMENTS
NOTE: Register file activity	already described in its own section.
2-1, ctrl-gate-regs-busses, {frac,expn,sign}-reg-bus{A,B} {frac,expn,sign}-bus{A,B}	Connect register file output to main datapath busses.
2-1, ctrl-latch-mainops, ctrl-latch-signops, ctrl-latch-expndiffs, frac-a,frac-b,sign-a, sign-b,expn-a,expn-b	Latch datapath inputs at {fraction, exponent,sign} boxes.
2-2, ctrl-latch-expnshamt expn-shamt,expn-ge128, expn-BgtA.	Latch in difference vectors, result of exponent comparison.
2-{3,4},frac-{left,right}, ctrl-MD/AS,frac-right-prenorm frac-{left,right}-AS.	Choose number with greatest exponent and route to the left. Shift right hand fraction by expn-shamt for decimal point alignment (generating roundbits). Pass through entry multiplexors for multiply/divide input.
2-3,expn-BgtA-tmp, ctrl-latch-expnadder-inleft, expn-left.	Select greater of input exponents and route to left side of exponent ALU for generation of result exponent.
2-4,ctrl-S/A-op, ctrl-fracadder-op, sign-dest.	Generate the control for determining whether an actual subtraction should be done, as well the result sign.
3-1,ctrl-latch-fracadder-in	Latch output of alignment section and pass through to arithemetic unit.
3-{1,2},ctrl-fracadder-op, ctrl-MD/AS,frac-addleft, frac-carryin,frac-right, frac-adder-result.	Complement left hand input and assert carry input if subtract. Do actual addition.
3-3,ctrl-latch-fracintermed	Latch in signed intermediate result.
{3-4,4-1},sign-intermed, frac-roundmode,sign-result frac-abs-intermed, frac-encoder-mainin, frac-S/A-expn, frac-normdist-shifter, frac-Shift-GT1.	Handle pre-normalization,rounding of incrementing of result fraction. Generate exponent adjustment distance and exponent add/subtract signal.
(NOTE: Should latch in again	in 4-1, but don't yet in the SLANG.)
{3-4,4-1}, frac-encoder-zerodet, frac-normdist-encoder, frac-result,frac-dest, ctrl-latch-fracdest.	Do final normalizing shift of fraction and generate zero-detect signal. Latch result into destination latch.
4-{1,2},expn-S/A-ctrl, ctrl-latch-expnadder-inright, expn-right,expn-result, expn-op-right,expn-carryin.	Add or subtract normalizing distance to form the final result exponent.
{4-3} ctrl-latch-expndest, expn-dest,frac-dest, sign-dest, ctrl-write-arithresults ctrl-gate-busses-regs.	Latch in final exponent and write all results back to register file.
(NOTE: Will move exponent result write to phi2.).	

Sequence Table for Multiply Instruction

MULTIPLY

TIME/SIGNALS/COMPONENTS	ACTION/COMMENTS
NOTE: Register file activity already described in its own section.	
2-1, ctrl-gate-regs-busses, {frac,expn,sign}-reg-bus{A,B} {frac,expn,sign}-bus{A,B}	Connect register file output to main datapath busses.
2-1,ctrl-latch-mcds, md-mcd-latch,md-mpr-latch.	Latch multiplicand and multiplier into input section.
2-1,ctrl-latch-fracadder-in, ctrl-MD/AS,ctrl-fracadder-op, frac-left,frac-right.	Latch multiplicand into left hand side of fraction box. Place 0 in the right side, causing multiplicand to be complemented.
2-{1,2},ctrl-fracadder-op, ctrl-MD/AS,frac-addleft, frac-carryin,frac-right, frac-adder-result.	Complement left hand input and do an addition to form complement.
2-3,ctrl-latch-fracintermed	Latch in multiplicand complement.
2-1,ctrl-latch-signops	Form result sign by XORing input signs.
2-1,ctrl-clear-selmrb, md-selmrb.	Clear multiplier byte counter. Start Booth recoding for 0th byte.
2-1,expn-left,expn-right, ctrl-latch-expnadder-inleft ctrl-latch-expnadder-inright.	Latch input exponents into exponent ALU to form initial result exponent.
2-{1,2},expn-S/A-ctrl, expn-op-right,expn-carryin, expn-result.	Add input exponents for initial result.
2-3,ctrl-latch-expndest, expn-dest.	Latch initial result into destination.
2-4,ctrl-latch-signdest, sign-dest.	Latch in result sign.
2-4,ctrl-clear-PPS/C-master, md-PP{S,C}-master.	Clear master of multiply result latch. Also clear master of sticky bit accumulator latch.
3-1,ctrl-latch-compmcds, md-compmcd-latch frac-busB.	Gate complemented multiplicand onto B bus and latch at multiplicand input section.
3-1,ctrl-mul/div-latch	Activate a latch to control multiply/ divide pipeline. Loop starts on an odd phase with the loading of the result slave latches and the latching out of the recoded multiplier byte. The result vectors propagate to the carry save adder stage while the multiplier byte is used to select different versions of the multiplicand.
3-1,ctrl-latch-PPS/Cslave, md-PP{S,C}-slave, md-xtrabits-slave, md-roundadder-PP{S/C}latch.	Latch in initial value of result slave latches for both the main and rounding results. These should both be zero initially.
3-1,ctrl-latch-Boothrcd, md-boothresult-mpy, md-boothresult-latch.	Latch recoding of 0th multiplier byte to multiplicand selector multiplexors.
3-2,ctrl-latchPP{S,C}-master, md-mcdmux{1-4,in}, md-csa-sum{1-4}, md-csa-carry{1-4}.	Propagate multiplicands through selection multiplexors. Add to previous iteration's results and latch in result master. (Addition is carry-save!!).
3-2,ctrl-inc-selmrb,	Increment byte counter and recode next byte

md-selmrb,md-mpr-byte, md-selmrb,md-boothresult-mpy.	of multiplier.
3-2,ctrl-latchPPS/C-master, md-roundadder-dest, md-xtrabits-dest, md-xtrabits-master, md-xtrabits-carryout.	Compute sum of the eight rounding bits for this iteration. Route the carry out of this addition to the rightmost bit of the carry result vector for the next iteration. Perform an OR on the eight result bits to form the sticky bit for this iteration.
3-3,ctrl-latch-PPS/Cslave, md-PP{S,C}-slave, md-xtrabits-slave, md-roundadder-PP{S/C}latch.	Move results to slave latches for next iteration. Note that slave roundbit latch is accumulated into roundbit addition ORing.
{NOTE: Pipeline completes a total of nine iterations (eight after this one.}	
7-2,ctrl-latch-PPS/Cmaster, md-PP{S,C}-master, md-PP{S,C}-result, md-carryin-mulop, md-roundbits-mulop, md-roundadder-dest, md-xtrabits-master.	Load in final result of multiply. Shift result by one and concatenate uppermost bit of rounding adder. Rest of adder result will make up rounding bits.
7-3,frac-carryin-MD, frac-roundbits-MD, ctrl-gate-PPS/C-fracBus, md-PP{S,C}-result, frac-bus{A,B}.	Gate result vectors, rounding bits to fraction box adder. Also send carry input to that addition.
7-3,ctrl-latch-fracadder-in	Latch output of alignment section and pass through to arithemetic unit.
7-{3,4},ctrl-fracadder-op, ctrl-MD/AS,frac-addleft, frac-carryin,frac-right, frac-adder-result.	Do actual partial product addition.
8-1,ctrl-latch-fracintermed	Latch in signed intermediate result.
(NOTE: The path to the final result is the same here as it is for the add/subtract instruction. Please see that table for additional explanation.)	
8-3,expn-dest,expn-b, ctrl-gate-expndest-expnBusB, ctrl-latch-expnb,expn-busB.	Feed initial result exponent back to exponent input B for generation of final result.
8-3,expn-left, ctrl-latch-expnadder-inleft.	Latch initial result through to left side of exponent ALU. Prepare for addition of normalizing distance in order to form final result.
(NOTE: Addition of normalizing distance occurs in same way as it does in the add/subtract instruction. Exponent result is latched in 8-4.)	
9-3,ctrl-gate-busses-regs, expn-dest,frac-dest, sign-dest, ctrl-write-arithresults.	Write all results back to register file.

Sequence Table for Divide Instruction

DIVIDE

TIME/SIGNALS/COMPONENTS	ACTION/COMMENTS
NOTE: Register file activity already described in its own section.	
2-1, ctrl-gate-regs-busses, {frac,expn,sign}-reg-bus{A,B} {frac,expn,sign}-bus{A,B}	Connect register file output to main datapath busses.
2-1,ctrl-latch-mcds, md-mcd-latch,md-mpr-latch, frac-a,frac-b.	Latch divisor and dividend into input section. The divisor goes into the same latch used by the multiplier in the multiply instruction, while the dividend occupies the multiplicand latch. NOTE: We also latch the operands into the fraction box input latches.
2-1,ctrl-latch-fracadder-in, ctrl-MD/AS,ctrl-fracadder-op, frac-left,frac-right.	Latch dividend into left hand side of fraction box. Place 0 in the right side, causing dividend to be complemented.
(NOTE: Dividend complementation proceeds in a manner similar to multiplicand complementation in the multiply instruction.)	
2-1,ctrl-latch-signops	Form result sign by XORing input signs.
2-1,expn-left,expn-right, ctrl-latch-expnadder-inleft ctrl-latch-expnadder-inright.	Latch input exponents into exponent ALU to form initial result exponent.
(NOTE: Initial result exponent generation and sign result generation proceeds as seen in the multiply instruction.)	
2-2,ctrl-clear-PPS/C-master, md-PP{S,C}-master.	Clear master of divide result latch.
2-3,ctrl-gate-fracb-PPSlave, ctrl-mux-PPSlave-in, frac-b,frac-busB, md-PP{S,C}-slave, ctrl-latch-PP{S,C}-slave.	Load dividend (which was contained in the frac-b latch) onto B fraction bus and into remainder result slave latch (formerly known as the partial product latch). Note that the other partial remainder latch is loaded with the zero contained in the master latch.
2-4,ctrl-clear-quotients, md-{pos,neg}quotient-latch.	Clear quotient estimeate latches.
3-1,ctrl-latch-compmcds, md-compmcd-latch frac-busB.	Gate complemented dividend onto B bus and latch at dividend (formerly multiplicand) input section.
3-1,ctrl-mul/div-latch	Activate a latch to control multiply/ divide loop. The divide loop officially starts in an odd phase with the generation of the quotient estimate based upon the divisor and partial remainder vectors. In even phases, the partial quotient is used to select different versions of the multiplicand. This result and the previous partial remainder are accumulated and latched to form the new partial remainder. This selection/accumulation process uses the same hardware employed for partial product generation in the multiply instruction. NOTE: This is a single loop, NOT a pipeline!! The only constraint on this hardware is that the entire loop must complete in two clock phases.
3-1,md-qboxpla-divin, md-qboxadder-in{left,right}, md-qboxpla-dsrin.	Start divide loop by setting up inputs to the quotient estimation PLA. For first input, add together topmost eight bits of partial remainder vectors and take topmost six bits of that result. Other input consists of the topmost four bits of the divisor.

3-1,md-qboxpla-mux-out, md-qboxpla-quotient-out, md-posq-dest, md-negq-dest.	Quotient estimation PLA output appears in two forms. One form is a "one-hot" encoding that drives the selection of the next dividend. The other is an encoding that is broken into two bit estimation vector that is either positive or negative.
3-2,ctrl-latchPP{S,C}-master, md-mcdmux1, md-csa-sum{1-4}, md-csa-carry{1-4}.	Propagate dividend through selection multiplexors. Add to previous iteration's results and latch in result master. Note that only one selection multiplexor is needed here.
3-2,ctrl-shift-quotients, md-{pos,neg}quotient-latch.	Latch in first quotient esimation.
3-3,ctrl-latch-PPS/Cslave, md-PP{S,C}-slave.	Shift partial remainders to the left by two and move to slave latches for next iteration.

{NOTE: Loop completes a total of 34 iterations (33 after this one).

19-4,ctrl-latch-PPS/Cmaster, md-PP{S,C}-master, md-PP{S,C}-result, ctrl-shift-quotients, md-{pos,neg}quotient-latch.	Load in final remainder of divide and the rightmost two bits of quotient.
20-1,frac-bus{A,B}, frac-roundbits-MD, ctrl-gate-PPS/C-fracBus, md-PP{S,C}-result.	Gate partial remainder vectors onto fraction busses to fraction box adder.

{NOTE: Alignment of these vectors is current INCORRECT in the Slang.}

20-1,ctrl-latch-fracadder-in	Latch output of alignment section and pass through to arithemetic unit.
20-{1,2},ctrl-fracadder-op, ctrl-MD/AS,frac-addleft, frac-carryin,frac-right, frac-adder-result.	Complement left hand input and assert carry input. Do actual subtraction.
20-3,ctrl-latch-fracintermed	Latch in signed intermediate result.
20-3,ctrl-latch-remsign, frac-remsign-latch.	Take sign of the partial remainder. NOTE: This is the only part of the partial remainder needed to form the final quotient.
20-4,md-quotient-roundbits, frac-remsign-latch, md-posq-result, md-negq-result, md-roundbits-divop, md-carryin-divop.	Prepare positive and negative quotient estimate vectors for final subtraction. Must actually do subtraction to form the three rounding bits, using the sign of the partial remainder as the carry input of this subtraction. The carry output of this subtraction is cascaded to the final quotient subtraction.
21-1,frac-carryin-MD, frac-roundbits-MD, ctrl-gate-quotientfracBus, md-posq-result,frac-bus{A,B}, md-negq-result.	Gate quotient vectors, rounding bits to fraction box adder. Also send carry input to that subtraction.
21-1,ctrl-latch-fracadder-in	Latch output of alignment section and pass through to arithemetic unit.
21-{1,2},ctrl-fracadder-op, ctrl-MD/AS,frac-addleft, frac-carryin,frac-right, frac-adder-result.	Do quotient estimate subtraction.
21-3,ctrl-latch-fracintermed	Latch in signed intermediate result.

(NOTE: The path to the final result is the same here as it is for the

add/subtract instruction. Please see that table for additional explanation.)

22-1,expn-dest,expn-b, ctrl-gate-expndest-expnBusB, ctrl-latch-expnb,expn-busB.	Feed initial result exponent back to exponent input B for generation of final result.
22-1,expn-left, ctrl-latch-expnadder-inleft.	Latch initial result through to left side of exponent ALU. Prepare for addition of normalizing distance in order to form final result.

(NOTE: Addition of normalizing distance occurs in same way as it does in the add/subtract instruction. Exponent result is latched in 22-2.)

22-3,ctrl-gate-busses-regs, expn-dest,frac-dest, sign-dest, ctrl-write-arithresults.	Write all results back to register file.

Sequence Table for Load/Store Instructions

STORES

TIME/SIGNALS/COMPONENTS	ACTION/COMMENTS
1-4, ctrl-stext1, reg-storelatch-master	Choose between fraction bus and exponent/ sign bus for store operation. Should be done as soon as decoding for opcode is ready.
1-4, phi4,fpu-RS1pin, ctrl-regnumbA	Send RS1 to register file for reading onto A bus. Must be done in parallel with operation decoding.
2-1, ctrl-storelatch-master, reg-storelatch-master	Load master of output latch with result of multiplexor.
3-1, ctrl-storelatch-slave, reg-storelatch-slave	Latch slave of output latch. Should start driving pads. Should NOT relatch when suspended.
3-1, ctrl-activate-datapins, fpu-DATApin, ctrl-store-memcycle.	Should drive pads at this point. On stores, value driven out should be held until cache hit signal is received (or Trap occurs).????

LOADS

TIME/SIGNALS/COMPONENTS	ACTION/COMMENTS
3-3, ctrl-activate-datapins, fpu-DATApin, ctrl-store-memcycle.	On loads, pads should be set to input mode on every phi3 where a load is in the memory cycle.
3-3, ctrl-loadlatch-master, reg-storelatch-master.	Load in master of input latch at same phase as data arrives from cache.
4-2, ctrl-loadlatch-slave, reg-loadlatch-slave.	Load slave latch and prepare to write to register file.
4-2, ctrl-regnumbA ctrl-latch-regnumbA	Load register decode latch for write from A bus. The register number comes the load/store opcode state vector!
4-3, ctrl-gate-loadlatch {fracBus,expnBus,signBus}	Gate slave of input latch to either fraction bus or exponent/sign bus based on type of load being performed.