Control Implementation for the SPUR Floating Point Coprocessor

Debby Jensen

Master's Report
Computer Science Division
University of California, Berkeley

ABSTRACT

SPUR is a RISC-based multiprocessor workstation being designed to facilitate parallel-processing research. Typically, RISC architectures achieve low performance levels for floating-point intensive applications, as the multiple-cycle floating-point instructions are not implemented in the hardware. In an attempt to raise these performance levels, the SPUR system provides floating-point support through an extended instruction set and a tightly-coupled floating-point coprocessor. This report documents the implementation of the control unit for this floating-point coprocessor; describing the coprocessor interface, control PLA definitions, the finite state machine, the dynamic cycle counter, the 4-stage load-store pipeline, and the random logic generated to drive the datapath modules. Implementation techniques and trade-offs are discussed; including design strategy, area and speed optimization, noise margin considerations, and delay balancing of the datapath control signals for clock skew minimization. Finally, simulation results obtained using SPICE, CRYSTAL, and MOSSIM are presented. The chip is implemented in 2-layer-metal 2μm CMOS technology, and uses a four-phase non-overlapping clock with a target cycle time of approximately 100ns - 140ns.

August 26, 1987

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **26 AUG 1987**	2. REPORT TYPE	3. DATES COVERED **00-00-1987 to 00-00-1987**

4. TITLE AND SUBTITLE
Control Implementation for the SPUR Floating Point Coprocessor

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
SPUR is a RISC-based multiprocessor workstation being designed to facilitate parallel-processing research. Typically, RISC architectures achieve low performance levels for floating-point intensive applications, as the multiple-cycle floating-point instructions are not implemented in the hardware. In an attempt to raise these performance levels, the SPUR system provides floating-point support through an extended instruction set and a tightly-coupled floating-point coprocessor. This report documents the implementation of the control unit for this floating-point coprocessor; describing the coprocessor interface, control PLA definitions, the finite state machine, the dynamic cycle counter, the 4-stage load-store pipeline, and the random logic generated to drive the datapath modules. Implementation techniques and trade-offs are discussed; including design strategy, area and speed optimization, noise margin considerations, and delay balancing of the datapath control signals for clock skew minimization. Finally, simulation results obtained using SPICE, CRYSTAL, and MOSSIM are presented. The chip is implemented in 2-layer-metal 2um CMOS technology, and uses a four-phase non-overlapping clock with a target cycle time of approximately 100ns-140ns.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**82**	

Control Implementation for the SPUR Floating-Point Coprocessor

List of Tables

List of Figures

1. INTRODUCTION

1.1. SPUR System Overview

SPUR (Symbolic Processing Using RISCs) is a shared-memory multiprocessor being designed here at Berkeley [7] to apply the RISC (Reduced Instruction Set Computer) concept to a parallel-processing workstation. The basic SPUR system consists of 6 to 12 identical processors; each with a custom 32-bit central-processing unit (CPU), a 128K-byte instruction and data cache and controller, and a floating-point coprocessor (see Figure 1-1). Each of the processors communicates through a global shared memory which, along with a single shared bus, simplifies parallel programming by eliminating the problem of specifying complex processor interconnections. Like it's predecessors, the SPUR CPU follows the typical RISC philosophy of (approximately) one-cycle pipelined execution, register-register operations with load-store accesses to memory, hard-wired control, and a large register file with overlapping windows.

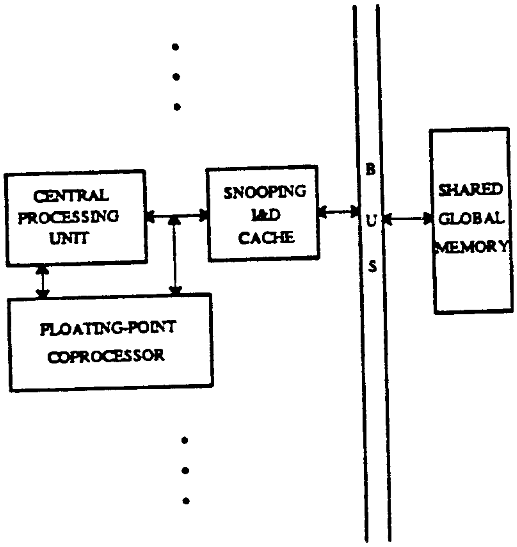


Figure 1-1 SPUR System With Floating-Point Coprocessor

RISCs evolved as a way to circumvent the problems inherent in microcoded control without sacrificing speed, efficiency, and simplicity of design. In order to develop an efficient processor with one-cycle execution, instructions are limited to register-register operations with a few simple addressing modes. The most commonly executed operations are optimized and placed in the hardware while less frequently executed operations, or multiple-cycle instructions, are implemented using software routines. Executing multiple-cycle floating-point instructions in software results in low performance for floating-point intensive applications [15]. One goal of the SPUR project was to increase this performance level by providing a floating-point coprocessor.

1.2. Floating-Point Coprocessor

The SPUR floating-point coprocessor implements the IEEE 754 standard without microcode by executing the most common functions in hardware and trapping to software to handle less frequent operations (such as transcendentals) and exceptions. The possible exceptions which can be detected [1] include: invalid operation, overflow and underflow, divide by zero, and inexact result due to rounding. The IEEE standard defines six different data types [8]: normalized, denormalized, zero, affine infinity, quiet Not a Number, and signalling Not a Number. The normalized and zero types can be implemented entirely in hardware, while the other four types are handled at least partially in software.

The floating-point coprocessor is *tightly coupled* to the SPUR CPU [6], which means that the presence of the coprocessor is transparent to the programmer, who sees only the extended instruction set. That is, the coprocessor is a feature of the *implementation* of the SPUR system, not the *architecture*. If a floating-point instruction is encountered and a coprocessor is present in the system, the instruction will be executed by the coprocessor; otherwise it will be implemented in software. The CPU handles all of the communication details at the hardware level through the coprocessor interface.

1.2.1. Coprocessor Interface

In order to improve floating-point performance through the use of an external coprocessor, it is essential that the overhead of the data transfer between the central processor and the coprocessor be insignificant in relation to the speed-up provided by the coprocessor. Several features of the SPUR coprocessor interface were designed to minimize this overhead [1]. In particular, the floating-point unit (FPU) can operate concurrently with the CPU, FPU loads and stores can be executed concurrently with arithmetic FPU operations, and a 64-bit data bus is provided to transfer data directly between the cache and the FPU (under control of the CPU, which calculates the effective memory address).

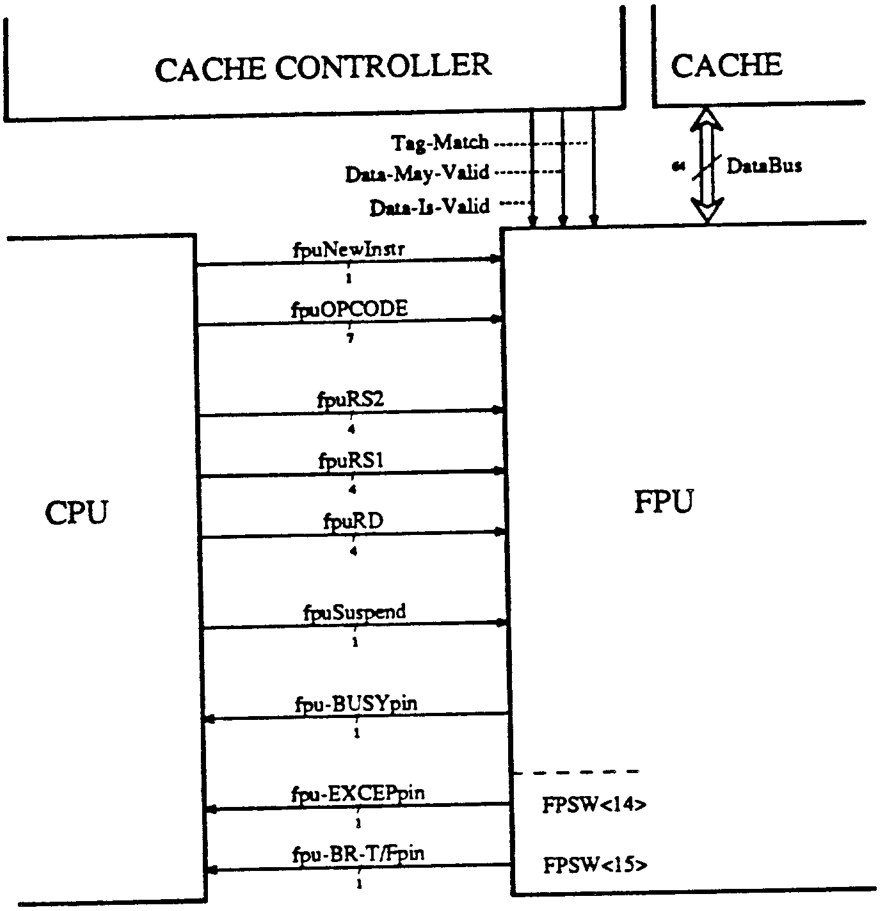


Figure 1-2 Coprocessor Interface Communication

Figure 1-2 shows the interface signals used for communication between the CPU and the FPU, including the three cache signals and data bus used in data transfer. Assuming no misses occur, the CPU receives and decodes the next instruction from its on-chip instruction buffer. The CPU asserts the *fpuNewInstr* signal if the decoded instruction is a floating-point load or store instruction, or if it is an arithmetic floating-point instruction and the FPU is not currently busy executing an arithmetic operation. The CPU stalls if an arithmetic instruction is decoded while the FPU is busy. The assertion of the *fpuNewInstr* signal causes the FPU to latch the instruction opcode and register specifiers into the instruction register, where they are again decoded and the instruction is executed. The addition of a 4-stage load-store pipeline (see Section 2.4) and 15 dual-ported floating-point registers allows floating-point loads and stores to be executed concurrently with arithmetic floating-point operations. The *fpuSuspend* signal allows the CPU to suspend all FPU memory operations, and is an input to the FPU internal finite state machine. Floating-point exceptions are detected and handled by the CPU when the *fpu-EXCEPpin* line is asserted. Also, as the CPU fetches the instructions and maintains the program counter, the *fpu-BR-T/Fpin* is needed to indicate the result (and branch direction) of a floating-point compare operation.

1.2.2. Extended Instruction Set and Data Formats

The extended floating-point instruction set [2,6] is shown in Table 1-1. Maintaining the RISC philosophy, the instruction set contains only register-register arithmetic instructions and load-store memory operations. All of the arithmetic operations except multiply and divide require four cycles. The multiply and divide instructions are implemented using iterative algorithms (See Section 3.1.3) and thus take 9 and 22 cycles, respectively. Loading data from the cache into the FPU requires only one cycle, while storing data into the cache from the FPU takes two cycles.

Table 1-1 SPUR Extended Floating-Point Instruction Set

Instruction Type	Instruction Syntax		Instruction Semantics	Cycles
ARITHMETIC	FADD	Rd,Rs1,Rs2	Rd <- Rs1 + Rs2	4
ARITHMETIC	FSUB	Rd,Rs1,Rs2	Rd <- Rs1 - Rs2	4
ARITHMETIC	FMUL	Rd,Rs1,Rs2	Rd <- Rs1 * Rs2	9
ARITHMETIC	FDIV	Rd,Rs1,Rs2	Rd <- Rs1 / Rs2	22
ARITHMETIC	FABS	Rd,Rs1,0	Rd <- Rs1; sign <- 0	4
ARITHMETIC	FNEG	Rd,Rs1,0	Rd <- Rs1; sign <- -sign	4
COMPARE	FCMP	cond,Rs1,Rs2	FPSW(cond) <- result	4
MOVE	FMOV	Rd,Rs1,0	Rd <- Rs1	4
CONVERT	CVTS	Rd,Rs1,0	Rd(sgl) <- Rs1(ext)	4
CONVERT	CVTD	Rd,Rs1,0	Rd(dbl) <- Rs1(ext)	4
LOAD	LD_SGL	Rd,Rs1,RC	Rd <- M(Rs1+RC)	1
LOAD	LD_DBL	Rd,Rs1,RC	Rd <- M(Rs1+RC)	1
LOAD	LD_EXT1	Rd,Rs1,RC	Rd <- M(Rs1+RC)	1
LOAD	LD_EXT2	Rd,Rs1,RC	Rd <- M(Rs1+RC)	1
STORE	ST_SGL	Rs2,Rs1,SC	Rs2 -> M(Rs1+SC)	2
STORE	ST_DBL	Rs2,Rs1,SC	Rs2 -> M(Rs1+SC)	2
STORE	ST_EXT1	Rs2,Rs1,SC	Rs2 -> M(Rs1+SC)	2
STORE	ST_EXT2	Rs2,Rs1,SC	Rs2 -> M(Rs1+SC)	2

As shown in the table, there are four separate types of load and store instructions. This is because single, double, and extended data formats may be specified (see Figure 1-3), and in the case of the extended format a separate load/store instruction is required for each 64-bit word. Although data may exist in memory in any of the above formats, only the extended format is actually implemented in the hardware. Therefore, when data in the single or double format is loaded into the FPU, it is automatically converted to the extended format by unpacking the exponent and fraction and assigning them to the extended fields, and setting tag bits to specify one of the six IEEE data types. To store data from the FPU into memory in the single or double format requires that the proper convert instruction be used in order to convert the data from the extended format into the desired format before the store is implemented.

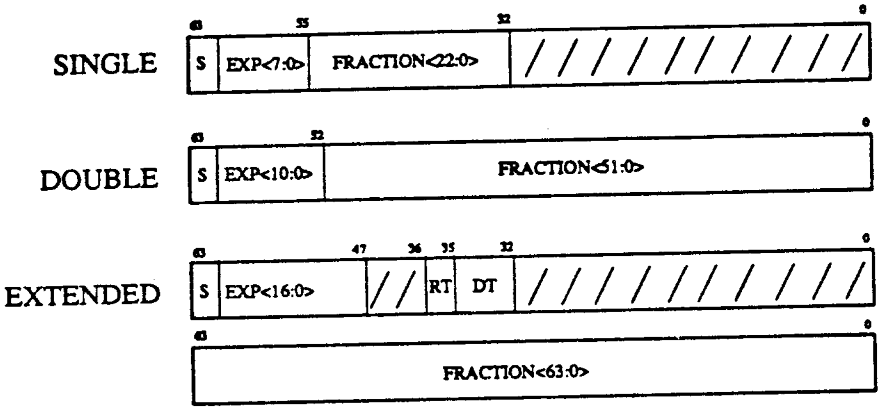


Figure 1-3 FPU Data Formats

1.2.3. FPU Floorplan

The basic structure of the floating-point chip is depicted in Figure 1-4. The FPU consists primarily of four main modules: the exponent (EXP), fraction, and multiply-divide (MULDIV) blocks, which constitute the datapath; and the control module, which is the focus of this report. The control unit latches and decodes incoming floating-point instructions, using a combination of PLAs and random logic to generate the necessary signals to control the datapath.

The main section of the control unit consists of the control programmable logic arrays (PLAs), a cycle counter, and interface logic, whereas the logic which generates the individual datapath control signals are located in the random logic strips in the proximity of the datapath block which they are controlling. The inputs to this logic are generally routed directly from the main control block, though in some instances these inputs come from other portions of the datapath. The outputs of this logic are then individually buffered to drive the datapath.

The floating-point chip is implemented in 2μm CMOS technology with two metal layers, where the

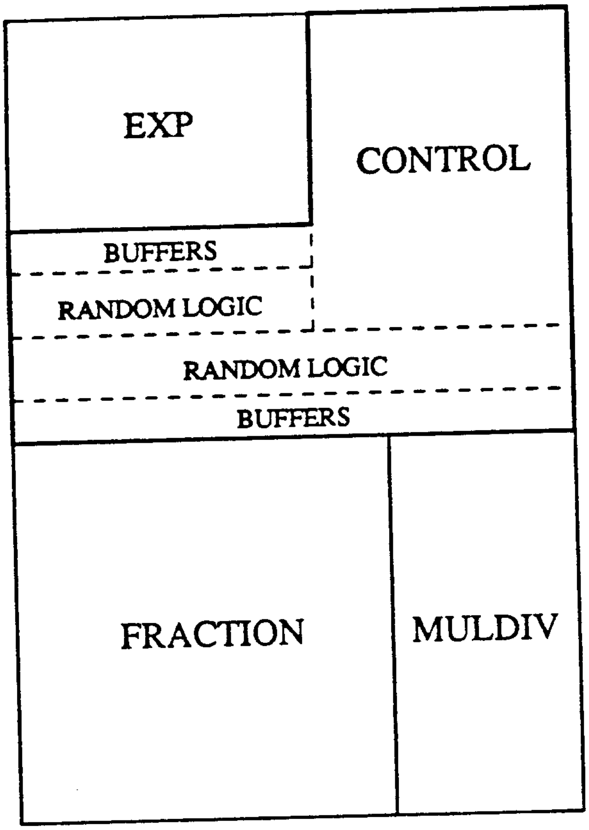


Figure 1-4 FPU Floorplan

power supply and data lines run horizontally in metal-2 and the control lines run vertically in metal-1. The entire FPU design is implemented assuming a four-phase non-overlapping clock, where the target cycle time is approximately 100ns - 140ns with a non-overlap time of 5ns - 10ns between each phase.

1.3. Report Outline

As stated above, this report focuses on the implementation of the control unit for the SPUR floating-point coprocessor. When I joined the project this year, much of the control was already specified at the functional level and had been simulated using SLANG [18,19], but no control had been layed out. This report documents the control layout completed for this project, beginning with a description at the functional level and progressing down to the low-level circuit details, including alternative implementations, functional verification, and simulation results.

Chapter 2 describes the main control block and the interface control, starting with an overview of the entire control unit. The second section describes the three PLAs central to the control unit design. This section is followed by a discussion of the implementation of the cycle counter. The last section of this chapter is centered on the design of the load-store pipeline.

Chapter 3 revolves around the datapath, including a description of the main datapath modules and the random logic used to control the modules. The first sections of this chapter describe the operation of the three datapath modules; the exponent, fraction, and multiply-divide units. The next sections discuss the layout strategies considered for the random control logic, including layout structure and regularity, speed optimization, and area minimization. This discussion is concluded with a comparison of the two alternative methods actually implemented. The following section deals with clock skew considerations for the control lines, including capacitance extractions of the datapath and individualized buffer sizing techniques.

The first section of Chapter 4 presents an overview of the CAD environment, outlining the various simulation tools employed and the interface between the tools. The following sections present the overall simulation results obtained using these tools including, when applicable, a comparison between the various tools.

Chapter 5 summarizes the work completed, providing conclusions and suggestions for future research. This includes interesting points discovered, lessons learned, and suggested paths to follow/avoid in future work. The following two chapters contain acknowledgements and references.

A lengthy appendix is attached which is broken into six main parts. Appendix A defines the inputs and outputs of the three control PLAs. Appendix B includes cell schematics and documentation which is referenced in this report. Appendix C includes the layout plots of most of the unique cells incorporated in the control unit, ranging from single cell plots up to a plot of the entire floating-point control unit. Appendix D documents the control signal definitions and their implementation. Tables of the extracted capacitances for each of the datapath modules, including buffer sizes and total delay times, are given in Appendix E. Finally, Appendix F contains source files for each of the simulators.

2. INTERFACE CONTROL UNIT

2.1. Control Unit Overview

A simplified block diagram of the control unit is shown in Figure 2-1. New floating-point instructions are latched on the chip in clock phase 3. The following phase, the instruction PLA decodes the opcode from the instruction latch, and the register destination is latched into the first stage of the load-store pipeline (see Figure 2-5), along with load-store specifiers in the case of a load or store instruction. The 4-stage pipeline allows the FPU to execute load-store operations concurrently with the multi-cycle arithmetic operations, as the required information can be latched into the pipeline every cycle. If an arithmetic operation is received, and the FPU is not currently busy executing a previous arithmetic operation, the operation type is latched into the arithmetic ops register, where it serves as an input to the arithmetic PLA. Using a 3-bit state register and a PLA, the internal finite state machine records the current state of the FPU (see Figure 2-2) and maintains the *fpuBusy* signal along with various other signals. The cycle counter keeps track of the current instruction cycle dependent on the state of the FPU, and is decoded by the arithmetic PLA. Outputs of the arithmetic PLA, cycle counter, finite state machine, and load-store pipeline serve to generate the random control signals required by the datapath modules.

2.2. PLA Partitioning

As shown in Figure 2-1, three PLAs form the core of the control unit: the instruction PLA, the arithmetic PLA, and the PLA in the internal finite state machine (IFSM). Each of the PLAs has a set of pass gates associated with it to ensure that the inputs to the PLA cannot change unless the pass gate control is asserted. That is, the pass gates allow the PLAs to operate only in the phase associated with the pass gate control; in all other phases the outputs of the PLAs will be considered stable and valid. The phase of operation for each of the control PLAs is indicated in Table 2-1. As an example, since the arithmetic PLA is evaluated in phase 2, its outputs can be used only in phases 3 and 4 of the current cycle, and phase 1 of the next cycle; signals needed in phase 2 must by created independent of the PLA. Furthermore, all of the outputs of the PLAs are generated independent of clock phase. That is, if a control signal is defined as a function of various inputs *and* a clock phase, the logical *and* of the PLA output and the clock phase must be

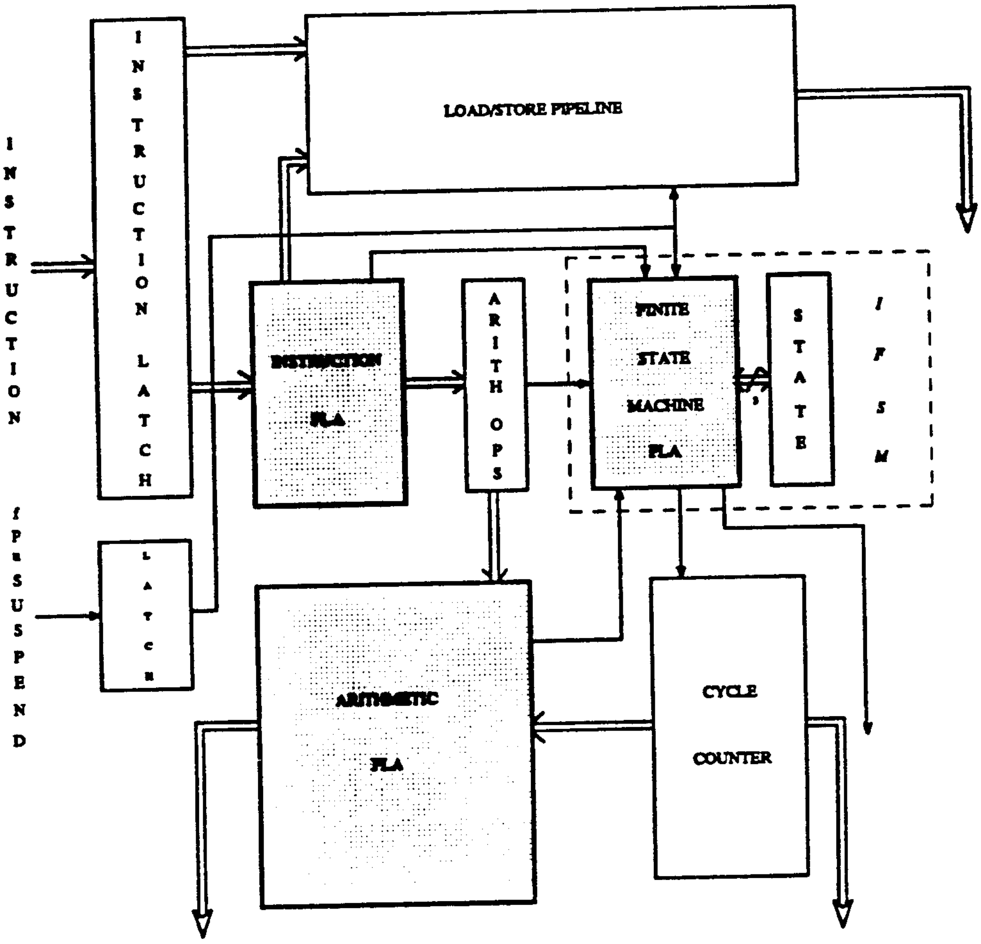


Figure 2-1 Control Unit Block Diagram (The shaded blocks are PLAs).

implemented outside the PLA.

As mentioned earlier, the function of the instruction PLA is to decode the instruction opcode, which is its only input (see Appendix A for a definition of all PLA inputs and outputs). The arithmetic PLA generates control signals for arithmetic operations, basically dependent on the type of instruction being executed and the current cycle count of that instruction. The IFSM is used to keep track of the current state of the FPU. The state transition diagram defining the IFSM is shown in Figure 2-2. As seen in the diagram, the state of the FPU is dependent on its current state and the control signals *ctrl-TrapRecvd*, *ctrl-start-arithop*, *ctrl-fpuSuscond*, and *ctrl-STOP*. As there are eight unique states possible, a state vector of three bits is maintained external to the IFSM to hold the current FPU state.

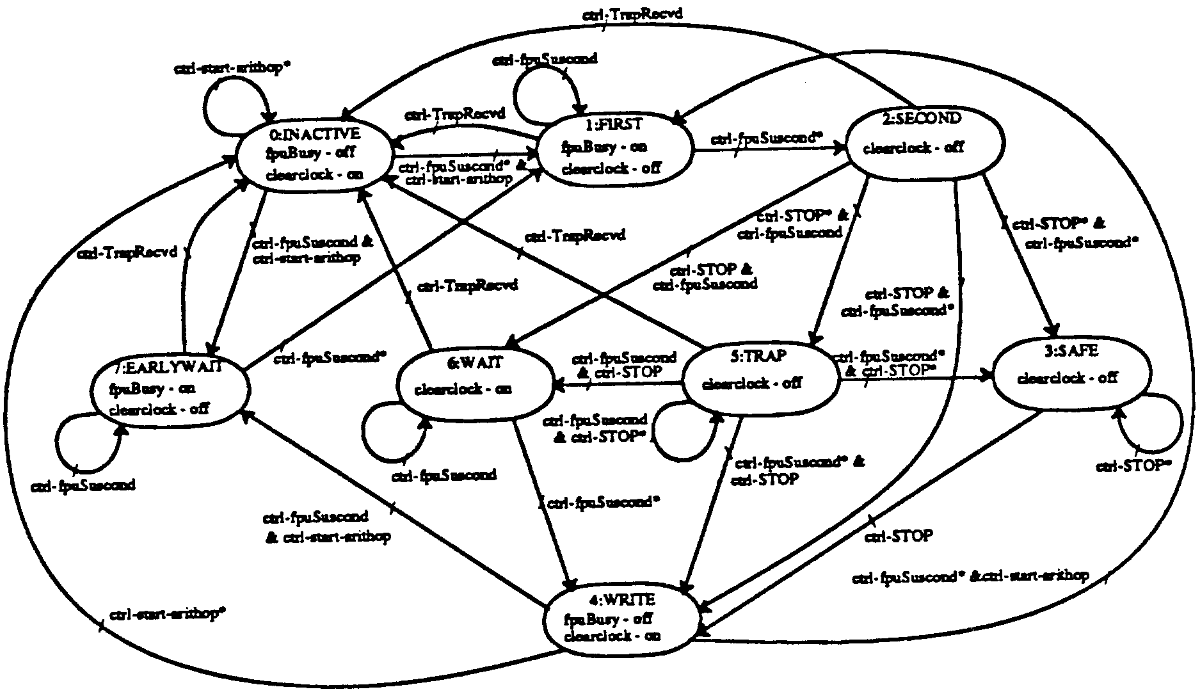


Figure 2-2 State Transition Diagram

The PLAs were each simulated using CRYSTAL [13,16] to determine the worst-case delay times. The results obtained are shown in Table 2-1, along with the corresponding PLA area. The last two columns list the worst-case propagation delays for the three PLAs, where the first column of delays corresponds to the worst-case times for the PLA outputs to fall from high to low measured relative to the change in the input. Likewise, the last column corresponds to the worst-case times for the PLA outputs to rise from low to high. The high-to-low delays are consistently the slowest among the PLAs, where the worst-case is 11.35ns. This is suitable for use in a 20ns (minimum) phase time while still allowing the PLA outputs to be latched within the same phase of evaluation (see Table 2-4) with an appropriate safety margin.

Table 2-1 Comparison of Control PLA Delay Versus Size

PLA	Phase	Inputs	Outputs	Area (μm^2)	$t_{p_{HL}}$ (ns)	$t_{p_{LH}}$ (ns)
instruction	PHI4	7	25	142746	11.13	9.93
arithmetic	PHI2	18	19	180675	10.34	9.74
finite state machine	PHI1	7	7	70282	11.35	9.43

As seen here, there is little difference (only about 0.5ns) between the delay times although, for example, the instruction PLA is about twice the size of the finite state machine. However, if all the PLAs had been condensed into one big PLA the total area would have been approximately five times greater, and we can extrapolate that the extra delay incurred would be at least 2.5ns, not counting the delays involved in the external routing required to and from the PLA. As it is, the partitioning of the PLAs effectively minimizes the average delay time while maintaining a logical grouping by function.

2.3. Cycle Counter

In order to keep track of the progression of the current instruction, a cycle counter has been incorporated as part of the control unit. For this purpose, a 5-bit counter was required, since the maximum number of cycles occurring in any FPU instruction is 22 (see Table 1-1). As shown in Figure 2-3, the cycle value is clocked into the master latch in phase 1, and into the slave latch in phase 2. In phase 3, the current cycle value is passed through to the increment logic, thus ensuring that the counter can be incremented only once per cycle. If the FPU is currently busy with an instruction or starting a new instruction, and is not suspended, the cycle value is incremented by one. Otherwise, the old cycle value is again clocked into the master latch. The counter may be cleared asynchronously in phase 3.

Usually, the current cycle value is fed into the arithmetic PLA and is decoded by the end of phase 2. As discussed in the previous section, this means that the decoded cycle value can only be used in phases 3 and 4, or phase 1 of the following cycle. The five *cycleclock-init* lines are used as inputs to random logic which locally decodes the cycle value for control signals which are needed in phase 2.

The control signal *ctrl-fpuBusy* is latched and stable by the end of phase 2, *ctrl-start-arithop* by the end of phase 4, and *ctrl-fpuSuscond* becomes available in phase 1 (see Figure A-B1 in Appendix B). Thus, the critical path here is to ensure that *ctrl-fpuSuscond* stabilizes early enough in phase 1 to allow the incrementor to work and the count value to be latched by the end of that phase. As the input to the ctrl-fpuSuscond latch is stable by the end of phase 4 and the worst-case delay for the static latch is 3.0ns (see Table 2-4), the worst-case delay for the increment logic (2nand2nand driving 0.404pF) is 2.0ns, and the set-up time for the master latch is 3.0ns, we still have about 12ns left to actually perform the 5-bit increment. (Note that this assumes a minimum specified phase time of 20ns.)

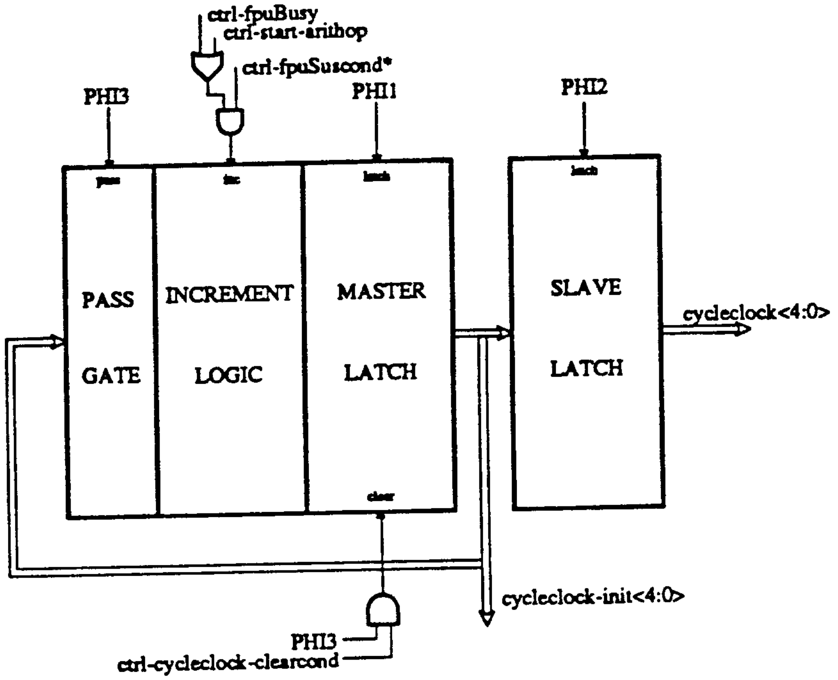


Figure 2-3 Cycle Counter Block Diagram

Two alternate methods of performing the increment were studied: the dynamic one shown below, which I used, and the static carry-look-ahead type used in the fraction datapath. The latter method was easiest to implement since the desired format already existed in the datapath. However, this method was too slow to meet the above specifications, due to the overhead involved in the carry-look-ahead preconditioning -- this overhead is simply too expensive when only a 5-bit increment is involved. The carry-look-ahead logic also incurred a large area overhead. In fact, the incrementor alone using carry-look-ahead took up almost as much space as the whole counter when implemented using the dynamic method!

The dynamic 5-bit incrementor shown in Figure 2-4 is based on the sum and carry definitions derived from the truth table in Table 2-2. Using simple Boolean algebra, we see that the sum for a bit is the *exclusive-or* of the current input and the carry into that bit, whereas the carry-out of that bit is the logical *and* of the current input and the carry-in. The carry-in to the first cell is simply the output of the increment logic. That is, if it is desired to increment the carry-in is high, and thus one is added to the current value. Otherwise, the value remains unchanged.

Table 2-2 Sum and Carry Definition

A_i	C_{in}	S_i	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

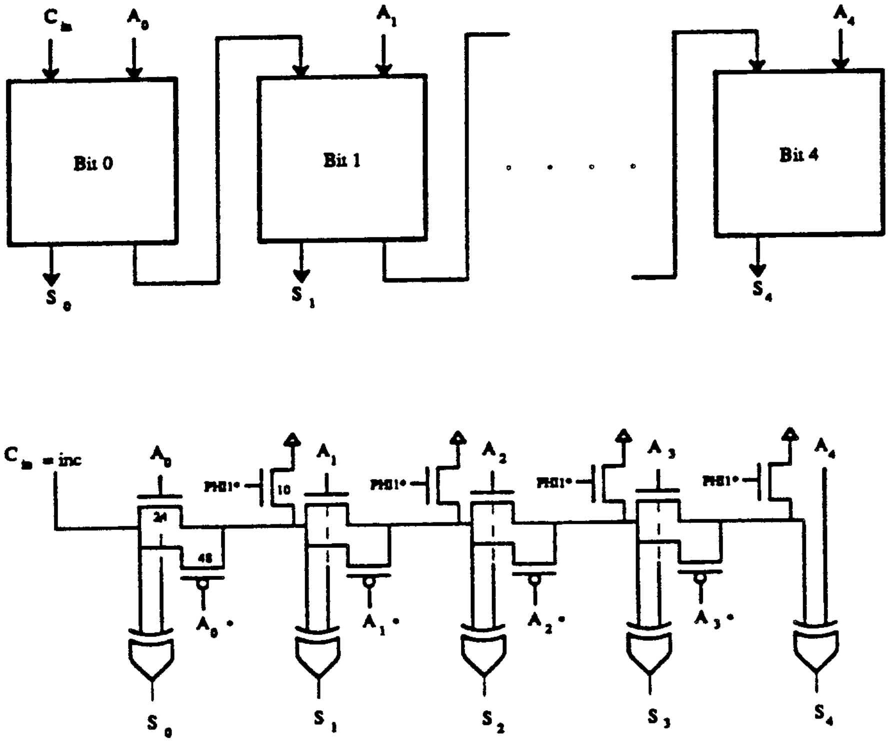


Figure 2-4 5-Bit Increment Implementation

A prime consideration here was the delay incurred in the dynamic carry propagation from the least significant bit to the most significant bit through the chain of pass gates. Various p-channel sizes were simulated using CRYSTAL to find an appropriate trade-off between layout area and propagation delay. As seen in Table 2-3, an equivalent pass width of 2λ (8/4) results in a total delay (including the *exclusive-or* delay) of 75.74ns, whereas an equivalent pass width of 12λ (48/4) brings this delay below 5ns. Further increases in p-channel width result in comparatively small gains in speed. Each of the p-channel sizes were thus chosen to be 48λ, and the corresponding n-channel sizes were made 24λ with no additional area requirements. Since the mobility of p-type carriers is about 50% less than that of n-type carriers, and the mobility of the transistor carrier is directly proportional to the current in the transistor channel, keeping the

2:1 ratio in width between the p-channel and the n-channel serves to equalize the available currents (and thus the delays) through both types of transistors. This ensure that in the worst case, where the *inc* signal goes low late in phase 1, any carry which may have been propagated up to that point can be discharged through the n-channel pass gates at approximately the same rate as the p-channel gates propagated the carry. The discharge transistors ensure that no carry is propagated except during phase 1, when the actual increment and latching occurs.

Table 2-3 P-Channel Width Versus Carry Propagation Delay

Channel (λ)	Cell Size (λ)	t_d (ns)
8	54	75.74
48	68	4.99
75	75	4.06

The implementation of the ***exclusive-or*** circuit, derived by a truth table similar to that above, is shown in Figure A-B2 in Appendix B. The 1.7ns delay associated with the ***exclusive-or*** function could possibly be reduced, if necessary, simply by increasing the transistor widths, which are currently minimum size.

2.4. Load-Store Pipeline

The loading of data from memory into the on-chip register file and the storing of data into memory from the register file is handled by the load-store pipeline and its associated memory control logic. As shown in Figure 2-5, the decode stage of the pipeline receives the load-store and register destination specifier information in phase 4 if a new FPU instruction is signalled. The load-store information is obtained from the instruction PLA, where it is decoded from the instruction opcode, while the register destination specifier is received directly from the instruction. If the FPU is not suspended and no traps have been received, this information is passed from one pipeline stage to the next in phase 1, allowing the pipeline to accept new memory information every cycle. The pipeline is suspended along with the FPU by recirculating the current contents of the two intermediate stages through a mux rather than passing along the contents of the previous stage. In this case, the contents of the write stage are cleared. If a trap occurs, the two intermediate stages are cleared, effectively flushing the pipeline.

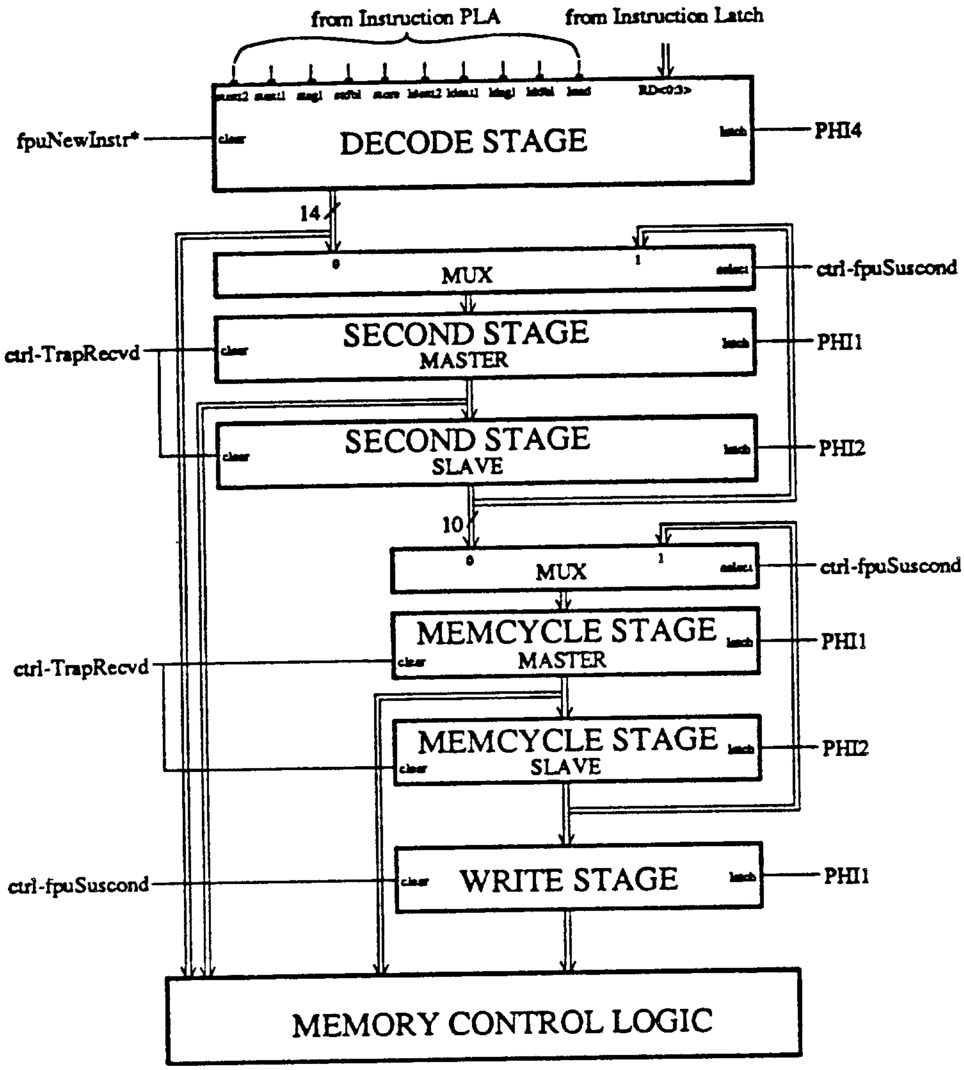


Figure 2-5 Load-Store Pipeline Block Diagram

The implementation of the pipeline was very systematic as it contained only two basic cells: a 2:1 mux and a clearable latch. Since a dynamic clearable latch offered little time or area savings yet had questionable noise margins (see Table 2-4), full static cells were used (see Figures A-B3 and A-B4 in Appendix B). For both the load-store pipeline and the cycle counter, I chose to maintain the 77λ cell pitch for consistency with the datapath modules, thus allowing me to access an existing library for many of the basic cell forms needed. These cells required modification basically to reduce the amount of routing associated with interconnecting the cells, some custom sizing of transistors, and simulation to verify functionality and timing constraints. The static mux was simulated using SPICE [20] and found to have a delay time of 0.5ns (assuming an output capacitance of 0.4pF). SPICE parameters obtained for both the static and dynamic latches (assuming an output capacitance of 0.8pF) are listed in Table 2-4.

Table2-4 Comparison of Latch Circuit Parameters

Parameter	Static	Dynamic
$t_{p_{hl}}$	3.0 ns	2.0 ns
$t_{p_{lh}}$	4.0 ns	3.0 ns
t_{setup}	3.0 ns	3.0 ns
t_{hold}	1.0 ns	2.0 ns
t_{clear}	0.5 ns	-
$V_{L_{max}}$	0.0 V	0.52 V
$V_{H_{min}}$	5.0 V	4.23 V

The memory control logic associated with the load-store pipeline uses the same random logic design discussed in the next chapter. However, for the sake of modularity, this logic is included as part of the load-store pipeline, as it is used by both the exponent and fraction front-ends. Detailed definitions of the control logic and implementation can be found in Appendix D.

3. DATAPATH CONTROL UNIT

3.1. Datapath Modules

The following three sections present a brief overview of the operation of the main datapath modules: the exponent unit, the fraction unit, and the multiply-divide unit. As seen in the simple block diagrams given with each section, the majority of the datapath consists of only a few basic types of simple cells: multiplexors, latches, shifters, tri-state bus drivers, and adders. A few miscellaneous cells are also needed, including the register cells and the front-end unpacking and convert logic. For the most part, this regularity greatly simplifies layout generation and simulation.

Control logic is associated with most of the basic cells, such as select lines for the multiplexors, clock lines for the latches, enables for the tri-state buffers, and so on. The second part of this chapter is concerned with the generation of this control logic, including implementation methods and problems such as speed optimization, area minimization, and clock skew. Appendix E contains detailed diagrams for each of the datapath modules, indicating the relative placement of the control signals used for each of the basic cells. The labels used in the diagrams are the actual names given the layout cells and their associated control.

3.1.1. Exponent Unit

A simplified block diagram of the 17-bit exponent datapath [1] is shown in Figure 3-1. The operation of the datapath is very straightforward. Data is loaded into the register file (from the bottom of the figure), being converted to the extended format and setting data type tags if necessary. The instruction register specifiers are used to access the current source exponents. For an add or subtract operation it is necessary to determine which exponent is largest, so that the binary points of the numbers can be aligned. It is important to determine this quickly, as the fraction unit cannot begin the addition or subtraction until the alignment is complete. Therefore, the difference between the two exponents is determined simultaneously using the two subtractors shown, Ea-Eb and Eb-Ea. The reason for executing both subtractions simultaneously is that although only one subtraction will determine which exponent is largest, the positive difference between the largest and the smallest is needed for alignment, and it may be that the one

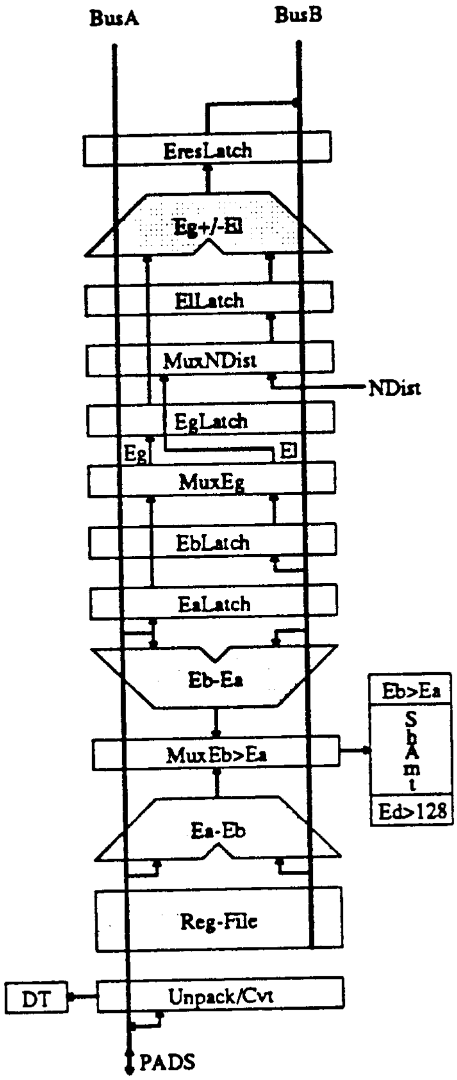


Figure 3-1 Exponent Datapath

subtraction instead gave a negative number. Thus, performing both subtractions at once ensures that the proper difference will be available immediately. The multiplexor, MuxEb>Ea, selects which of the differences obtained from the two subtractors is to be used as the shift amount for the fraction unit. Two other lines, Eb>Ea and Ed>128, are also used by the fraction unit. The first signal indicates that the exponent on the B-bus is larger than that on the A-bus. This is necessary as the fraction datapath assumes that the operand with the greater exponent comes from the A-bus, and if this is not the case a swap must be performed by the fraction unit. The Ed>128 signal indicates that the difference between the exponents was greater than 128 (7 bits). Once the addition or subtraction is complete, the Eg+/-El adder/subtractor is used to adjust the result exponent by the correct normalization distance. As the adder/subtractor assumes that the left operand is the greater exponent and the right operand is the normalization distance of the result, the first multiplexor, MuxEg, is used to select the greater exponent and the second multiplexor, MuxNDist, is used to select the normalization distance. The normalized result is then latched into EresLatch, and is ultimately put onto the B-bus.

The same datapath is used to perform the necessary tasks for a multiply or divide operation. The Eg+/-El adder/subtractor is used to calculate the sum of the exponents for multiplication, and the difference of the exponents for division. This adder is then used again to normalize this result. Unlike the operation above for an add/subtract instruction, the left operand to the adder is now the smaller exponent. Thus the second multiplexor is used to select the smaller exponent rather than the normalization distance.

3.1.2. Fraction Unit

The main function of the 64-bit fraction unit [1] as shown in Figure 3-2 is to perform the addition or subtraction of the floating-point fraction (magnitude). Symmetrical with the operation of the exponent unit, the fraction portion of the data is loaded into the register file, again going through any necessary unpacking and conversion to the extended format. The register specifiers access the current source magnitudes from the register file, and prepare to perform the specified addition or subtraction. To do this, however, requires that the binary point of the two magnitudes be aligned. As mentioned earlier, the fraction unit receives the required shift amount for this alignment from the exponent unit, along with a signal indicating which exponent is the largest. If this signal indicates that the operand with the greater exponent is not on the A-bus, a multiplexor is used to swap the two operands. The barrel shifter is then used to shift the greater magnitude right until alignment of the two operands is achieved, as determined by the difference of their exponents. The information lost during the right shift can be condensed into the three GRS bits (guard, round, and sticky). The guard and round bits are the two most-significant bits shifted out, and the sticky bit is the logical *or* of all of the rest of the lost bits. This bit indicates whether any precision was lost in the shift operation, or if the bits shifted out were all zeros.

Once the operands are aligned, the addition or subtraction is performed and clocked into an inter-

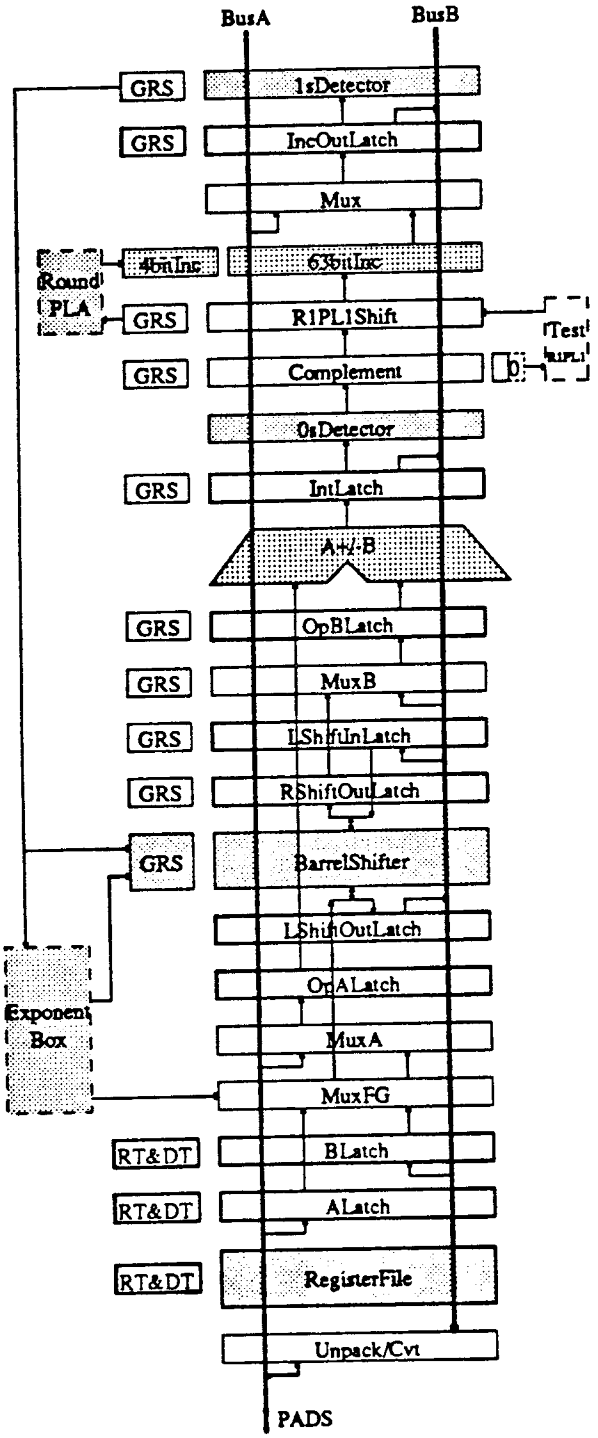


Figure 3-2 Fraction Datapath

mediate latch. To complete the operation, rounding and normalization are done using the three GRS bits, a rounding PLA, and an incrementor. If necessary, a normalization distance is sent to the exponent unit for normalizing the final result exponent. The final result is ultimately put onto the B-bus.

3.1.3. Multiply-Divide Unit

A simplified diagram of the multiply-divide unit [3] is depicted in Figure 3-3. Due to area constraints on the chip, the 64x64-bit multiply is implemented as an iterative 64x8-bit multiply, with the partial sum and carry vectors being accumulated in the PPS-SLatch and PPC-SLatch, respectively. A large carry-look-ahead adder is required to add these two vectors, forming the final product. Again due to area constraints, the adder already existing in the fraction unit was borrowed for this purpose. This adder is also used for calculating the complement of the multiplicand/divisor.

As shown in the diagram, the multiplier (MPR) is latched from the A-bus, and the complement of the multiplicand (COMPMCD), along with the multiplicand (MCD) itself, is latched from the B-bus. A version of the Booth recode algorithm is used here, which takes in turn each byte of the multiplier (including the most-significant bit from the previous byte) and groups the byte into four overlapped triplets. Each of the triplets is decoded by the Booth logic to select one of five inputs to the carry-save-adder (CSA) tree: zero, MCD, 2MCD, COMPMCD, and 2COMPMCD. The input selected by the least significant triplet is latched directly into its associated latch, the input selected by the next significant triplet is shifted left two bits, and so on. The four (shifted) inputs selected from the four triplets are added to the accumulated partial sum and carry vectors from the previous iterations using the 4-stage CSA tree to obtain the new partial sum and carry vectors. To avoid using a 128-bit datapath, the partial sum and carry vectors are shifted right by eight with each iteration. A 'rounding' adder accumulates the bits which are shifted out. As stated above, the final sum and carry vectors are added in the fraction unit to obtain the product.

The divide operation is performed using an iterative non-restoring algorithm [3], where two bits of the quotient are determined per iteration. The same datapath is used for both multiply and divide, and much of the hardware is shared. The multiplexor preceding the PPS-SLatch allows the PPS slave latch to be loaded either with the contents of the master latch or the contents of the A-bus. For a divide operation, the A-bus is used to initially load the dividend into the latch. As opposed to the multiply operation which shifts right by eight, the PPS and PPC latches, which hold the partial remainders, are shifted left two bits with each iteration. Under the algorithm, six bits of partial remainder and four bits of divisor are used by the quotient logic to select the next two bits of the quotient.

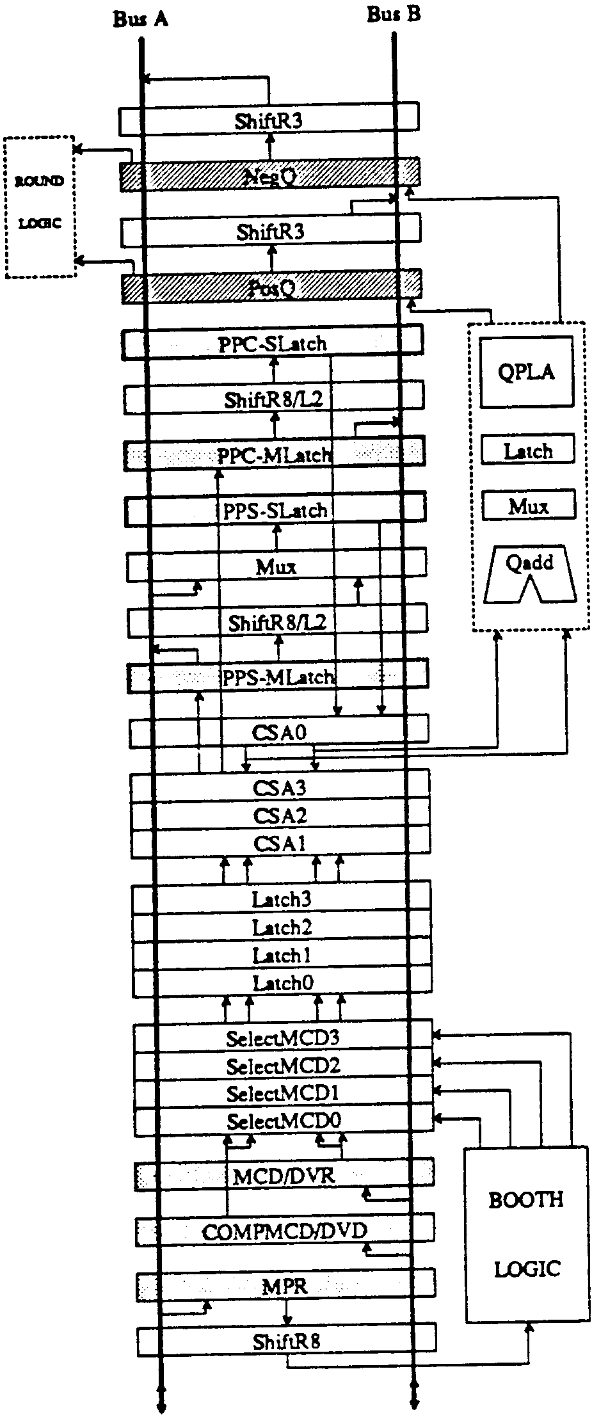


Figure 3-3 Multiply-Divide Datapath

3.2. Control Implementation Considerations

The random logic design associated with the datapath and memory control is definitely the most challenging part of the control unit implementation. This is mainly because there is little inherent structure in (and between) the individual cells, and each cell needs to be fine-tuned for its particular application. That is, each signal must be buffered depending on the capacitance of the line it is driving to ensure that there will be minimal propagation delays, and the delays must be balanced such that clock skew is avoided. The following sections contain a detailed discussion of these considerations, including a study of alternative implementations and support of design decisions.

3.2.1. Layout Strategy

The features I strove for in the design of the random logic included layout structure and regularity, speed optimization, and area minimization. Structure and regularity are especially instrumental in achieving layout that is easy to generate, simulate, debug and modify. In particular, it is much easier to simulate and change one cell that is used several times, rather than individually testing and modifying several unique cells. These features are also important in terms of cell interconnection and routing. Several error-prone steps can be avoided simply by arranging the basic cells such that most of the interconnections between cells, particularly the power supply lines, are made automatically. This reduces the chances of missing interconnections, and the regularity generally allows such flagrant errors to be spotted on layout plots.

As always, there are trade-offs involved in any design problem. The questions of area minimization and speed optimization are often opposed to one another, and both may be opposed to the concept of structure and regularity. For example, minimizing the area and associated delays for each signal requires the individual design and optimization of each random logic cell, contrary to the features desired above. Likewise, all cells could be designed minimum-sized, but would not operate at the optimum speed. It has been shown [10] that in order to minimize the delay for a single stage, the ratio of the stage's output capacitance to input capacitance should be approximately e. As an example of the use of this rule, a chain of inverters (buffers) is often used to drive a large load capacitance. If an unlimited number of buffer stages may be used, the first inverter is designed with minimum-size transistor channels, the second inverter's channels are about three times larger, and so on. However, if the number of stages is limited, then the optimum solution is to scale each of the succeeding stages by the fanout factor f, where $Y = f^N$. Here, Y is defined as the ratio of the total load capacitance to the input gate capacitance of the first stage, and N is the total number of stages. Another consideration here is the concept of delay balancing and clock skew, which is discussed in detail in Section 3.2.4.

3.2.2. Alternative Implementations

Looking for some degree of structure in the random cell layout, I studied the various forms of logic needed for the datapath and memory control signals (see Appendix D). After some manipulation, most of the logic was transformed into a two-level *and-or* form (or *nand-nand* as shown in Section 3.2.3), giving me a template upon which to base my cell design. This led to the consideration of several methods of implementation. The *and-or* format naturally suggested a dynamic or domino cell [5], whereas the *nand-nand* format suggested a static layout using a library of *nand* cells. Pure domino logic was rejected on the basis that many of the logic blocks are used in successive phases for some operations, leaving no time for pre-charge. However, a pseudo-nMOS implementation seemed a likely extension. The following sections present a pseudo-nMOS and a full static implementation studied in detail, concluding with a comparison of the two methods and the selection of the full static design for the random logic implementation.

3.2.2.1. Pseudo-nMOS

As shown in Figure 3-4, the pseudo-nmos form adapts readily to the *and-or* format, and can easily be extended to any number of inputs corresponding to this format. From the layout plots included in Appendix C, we can see that this design style provides both economy of space and a well-structured design. The area savings is mainly provided by the fact that only two p-channel pull-ups are required, whereas in typical static circuits a full complementary design is employed -- providing one p-channel pull-up for every n-channel pull-down. However, there are many important drawbacks to this type of ratioed logic, including increased power consumption and charge-sharing considerations.

To begin, several hand-calculations were performed to establish a ratio factor for the cells. Given that it was desirable to keep the static pull-up transistor as small as possible to decrease the amount of power consumed, I wanted to find the equivalent width of the pull-down transistors required to guarantee that they were strong enough to pull the pre-charged node (V_x) down below a specified safety voltage to avoid any chance of accidentally triggering the succeeding stage. At the time, we set the safety voltage at

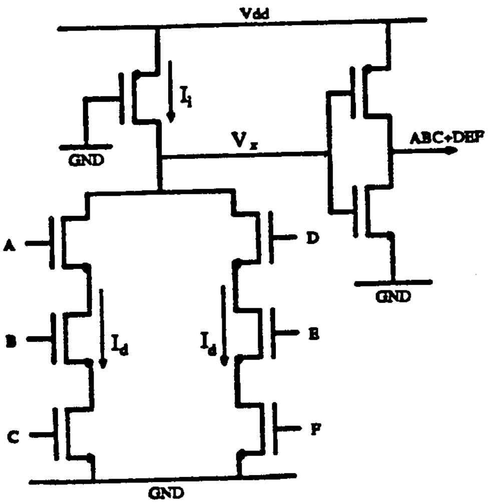


Figure 3-4 Pseudo-nMOS Design Style

0.5V, given that the threshold voltage of the following stage is about 0.75V and including a slight safety margin, and derived the ratio factor using this value. Obviously, the bigger the p-channel device, the more current it will source, the more power it will consume, and the harder it will be for the n-channel devices to overcome it to pull the node down towards ground. No matter how many *and* strings I have hanging off of the V_x node, the worst case is where only one of them is turned on (assuming all *and* branches have equivalent n-channel widths), thus giving less current to ground than the case where two or more branches are turned on. This assumption will give a conservative estimate for the actual size needed, as some leakage will occur among the off branches thus helping to lower the node voltage. For this case, I can easily solve for the steady-state voltage at the pre-charged node using the fact that the source (pull-up) current is equal to the sink (pull-down) current. The calculation proceeds as follows:

$$\text{PMOS}: V_{gs} = -5\,V;\ V_T = -0.75\,V;\ V_{ds} = V_x - 5$$

$$V_{gs} - V_T = -4.25\,V.\ \ (\text{Assume } V_x < 0.75\,V.)$$

$$\text{Therefore, } V_{ds} < -4.25\,V.\ \ (\text{The transistor is saturated.})$$

$$I_{l_{sat}} = \frac{KP_p}{2}\frac{W}{L}(V_{gs} - V_T)^2 \text{ where } KP_p = 76u.$$

Assume that for a string of n-channel gates, the equivalent W/L ratio for the string is the individual ratio divided by the number of input gates in the string.

$$\text{NMOS}: V_{gs} = 5V;\ V_T = 0.75V;\ V_{ds} = V_x$$

$$V_{gs} - V_T = 4.25V.\ \ (\text{Assume} V_x < 0.75V.)$$

Therefore, $V_{ds} < V_{gs} - V_T$. (The transistor is linear.)

$$I_{d_n} = \frac{KP_n}{2} \frac{W}{L} [2(V_{gs} - V_T)V_{ds} - V_{ds}{}^2] \text{ where } KP_n = 27u.$$

Setting these currents equal, and setting Vx=0.5 V:

$$\frac{27}{2} \frac{W_p}{2} (-4.25)^2 = \frac{76}{2} \frac{W_n}{2} [2(4.25)(0.5)-(0.5)^2]$$

$$\frac{W_p}{W_n} = \frac{152}{243.8} = 0.623$$

Therefore, if W_p is minimum size (4λ), then W_n must be approximately 6.42λ. Solving the quadratic equation for V_x given $W_p = 4$ and $W_n = 6$ gives us $V_x = 0.54$V. This was still considered acceptable, giving a safety margin of 0.21V.

Many cells were layed out and simulated using this ratio factor. Table 3-1 lists the results for some of the more common configurations used for comparison with the full static method, which is discussed in the next section.

3.2.2.2. Full Static CMOS

A typical full-complementary static design is presented in Figure 3-5. It is noticeably more complex than the pseudo-nMOS design, due to the multiple pull-ups, the two-level design, and the extra routing. However, this static design has none of the ratio, noise margin, charge sharing, and power consumption problems associated with the previous design.

A library of cells was constructed containing the basic building blocks needed to generate any of the specified control signals. Following the previous discussion I tried to minimize the delay between the logic levels, and therefore designed cells suited for a specific level. That is, all first-level cells were built with minimum-size channel widths and all second-level cells have channel widths about three times this size. There were only a few instances where a third level of logic was required -- these were sized on an individual basis. Since there are so many different control signals driving a wide range of capacitances, it would be a difficult problem to individually optimize each level of each cell. Thus the factor of three was chosen as a universal approximation, the discrepancies in delay are taken up in the double buffer stage as discussed in Section 3.2.4. To maintain the goals of structure and regularity, the levels were designed such that inputs flow in from the top in metal-1, and the logic outputs flow out the bottom in metal-1,

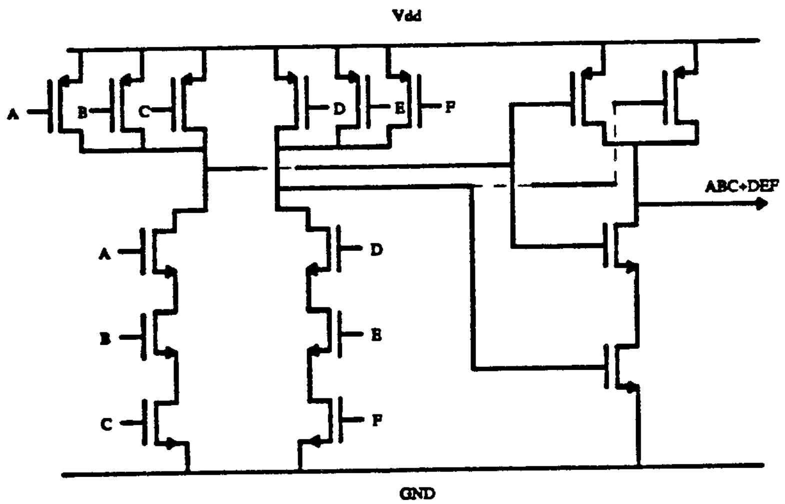


Figure 3-5 Full Static CMOS Design Style

interconnections between levels being performed automatically (see Appendix C). As most of the cells used metal-2 only for Vdd and GND routing, many channels are left free and can be used for global routing.

3.2.2.3. Comparison

A comparison of various parameters of interest for the two design styles discussed above is presented in Table 3-1 below. In order to make the comparisons as accurate as possible, all simulations were performed assuming a typical datapath load capacitance of 3.0pF (from tables in Appendix E) and double buffering of 60λ (n-channel width). The first column is the function name of the cell being simulated, where the cells are ordered in groups of the two alternatives. In all cases, the top cell (*and-or* structure) is the pseudo-nMOS implementation, and the bottom cell (*nand-nand* structure) is the full-complementary static implementation. As shown in the following section, these two implementations can be shown to be functionally equivalent through repeated applications of DeMorgan's theorems. See Appendix C, Figures A-C1-C10, for a comparison of the actual layout plots implementing the following functions.

The second column indicates the number of transistors required to implement the given function. For the simplest function shown, a 2-input *and* gate (2and1or or 2nand1nand), less transistors are required for the static implementation than pseudo-nMOS. This can be explained by the inverter overhead required

Table 3-1 Comparison of Pseudo-nMOS Versus Full Static CMOS

Function	Devices	H (λ)	W (λ)	$V_{L_{max}}$ (V)	$V_{H_{min}}$ (V)	$t_{P_{f}}$ (ns)	$t_{P_{r}}$ (ns)
2and1or	5	63	28	0.420	4.908	4.5	6.0
2nand1nand	4	147	24	0.000	5.000	5.0	5.0
2and2or	7	63	43	0.420	4.926	5.0	6.0
2nand2nand	12	147	48	0.000	5.000	5.5	5.5
2and3or	9	63	46	0.418	4.834	5.0	6.5
2nand3nand	18	147	72	0.000	5.000	6.0	6.0
2and4or	11	63	60	0.417	4.697	5.0	7.5
2nand4nand	24	147	96	0.000	5.000	6.5	6.0
2and5or	13	63	74	0.417	4.660	5.0	7.5
2nand5nand	30	147	120	0.000	5.000	7.0	6.0
3and2or	9	63	48	0.473	4.704	5.0	7.0
3nand2nand	16	147	64	0.000	5.000	6.5	5.5

for the pseudo-nMOS implementation. However, for most of the configurations, the number of transistors needed for the static implementation is around twice that of the pseudo-nMOS requirement. This is because the static method requires two transistors for every additional input, whereas only one is required using pseudo-nMOS. Furthermore, the static implementation has the additional overhead of the second level of logic, which also has two transistors for every input, while the pseudo-nMOS has only the overhead of the inverter in all cases. The most commonly used cell type is the 2-input *and* gate, which is used about twice as many times as the next three configurations in the table and about five times as often as the last two configurations.

The height and width of the actual layout cells required by the functions are given in columns three and four. As shown, the pseudo-nMOS uses much less chip area, attributable to three facts. In the first case, less devices take less area, especially since basically only one type of channel is used (n-channel) and thus the overhead of the tub separation design rule is reduced. Secondly, the implementation allowed the layout to be resolved using a single level, reducing interconnect overhead. Finally, the *and* strings using the pseudo-nMOS method could be placed immediately adjacent to their neighbor string, thus sharing many ground and node connections. In contrast, the use of *nand* cells from a general library in the static implementation required that the individual cells be separated by a minimum distance of 4λ, and no connections could be shared. The main consideration here was to try to keep the logic cells tall and narrow, so that they could all be placed directly above the datapath cells where required. As shown, the pseudo-nMOS method best met this goal.

The next two columns show the noise margin characteristics of the cells. As expected from a full CMOS implementation, the static style obtained full restoration of the voltage levels. The pseudo-nMOS method also performed as expected, with the low voltage being slightly less than the 0.5V threshold limit. The high voltage indicates some charge sharing, though the amount also remains within the 0.5V specified safety threshold.

Propagation delay times are given in the final two columns. All times are comparable, with static low-to-high (rise) times usually slower and high-to-low (fall) times usually faster than the pseudo-nMOS times. The pseudo-nMOS fall times will always increase with the addition of *and* strings, as there is essentially a limited supply of current available through the minimum-size p-channel source and an increasing amount of capacitance to charge. The rise times are dependent on the width of the n-channel sinks, which could be increased with a small area penalty. However, they cannot be decreased without upsetting the ratio balance, so it is difficult to reduce skews at this level by balancing the rise and fall times. The static cells are designed with a 2:1 p-channel to n-channel ratio in order to provide automatic balancing of the rise and fall times. As seen, a better natural balance could possibly be obtained by increasing this ratio towards 2.5:1 (this is dependent on the mobilities of the p- and n-type materials, which is a processing parameter).

The decision to proceed with the "safe" static design came after much discussion [2], although the area required for this implementation was shown to be much larger. The basic argument in support of this decision was the fact that the extra area needed was available on the chip, and so the "safer" design seemed appropriate. Besides significantly decreasing design time, the use of library cells also aided the goal of structured and regular layout, providing an easy way to add new cells or quickly modify existing ones simply by replacing one library cell with another. No major penalties were taken with the rise and fall times, either, as both were comparable, and the buffer sizes can still be optimized to reduce the times shown here.

Looking back, I believe that the pseudo-nMOS implementation would have been quite suitable for this application, though at the expense of an increased design cycle. In particular, the safety voltage threshold should be lowered to allow an even greater noise margin. Specifying a safety threshold of 0.25V would have involved increasing the n-channel sizes such that a ratio of 12.44:4 is maintained between the n-channel width and the minimum-size p-channel width. However, this would also have served to increase the charge-sharing problem, as increasing the n-channel width increases the amount of capacitance in the *and* strings available for charge sharing. This could then be alleviated by increasing the size of the pre-charge node, possibly by increasing the size of the output inverter associated with this implementation and thus increasing its input gate capacitance. However, this also has its penalty, as increasing the size of the pre-charge node increases the circuit delay. Many more SPICE simulations (and costly design time!) would have been necessary to ensure safe operation using the pseudo-nMOS method, and safety is especially important in control implementation.

3.2.3. Logic Definition and Conversion

Most of the random control signals defined in the SLANG description of the floating-point unit were converted to a *nand-nand* format for implementation using DeMorgan's theorems. These theorems can be stated simply as (AB)* = A*+B* and (A+B)* = A*B* [9], where '+' denotes the *or* function and '*' indicates complementation. Two typical examples of the sequence of steps taken to define and implement the random logic are shown below. The first example, *cm-latch-master*, illustrates one of the most common logic forms implemented: the 2nand2nand structure.

```
(defnode cm-latch-master
 (depends ctrl-mul/div-latch phi2 phi4)
 (update  (And (Or phi2 phi4) ctrl-mul/div-latch))
 )
```

Example 3-1 SLANG Description of *cm-latch-master*

From Example 3-1, we see that *cm-latch-master* = (phi2+phi4)ctrl-mul/div-latch. This can be directly expanded to (phi2*ctrl-mul/div-latch) + (phi4*ctrl-mul/div-latch), bringing us to the *and-or* representation. To use the theorems given above, we can let A = (phi2*ctrl-mul/div-latch)* and B = (phi4*ctrl-mul/div-latch)*. This gives us an *and-or* form of the type A*+B*. Using the first theorem, we have A*+B* = (AB)*, or *cm-latch-master* = ((phi2*ctrl-mul/div-latch)*(phi4*ctrl-mul/div-latch)*)*, which can be recognized as the *nand-nand* representation. Both of these forms are depicted in Figure 3-6.

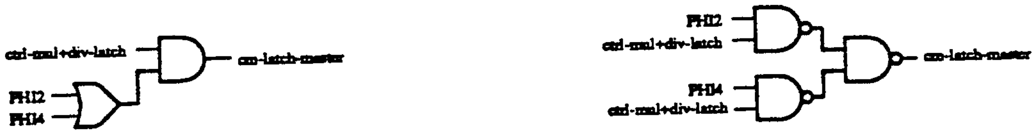


Figure 3-6 Equivalent Representations of *cm-latch-master*

As another example, I considered the SLANG definition of *cm-latch-slave*. Unlike most of the control signals, *cm-latch-slave* is unique in that I implemented it using three-level logic, as illustrated in Figure 3-7. It was possible to reduce *cm-latch-slave* to a two-level implementation using DeMorgan's theorems, but this required 4 invertors, one 2-input *nand* gate, eight 4-input *nand* gates, and one 9-input *nand* gate! Generally, *nand* gates are limited to at most five inputs as using more inputs leads to either a large series resistance to ground or a huge input gate capacitance, both of which increase the delay times. Therefore, the three-level implementation was considered more appropriate, and also required only two different types of *nand* gates.

```
(defnode cm-latch-slave
 (depends ctrl-mul/div-latch phi1 phi3 clock6_mulop clock18 ctrl-latch-ops)
 (update  (Or (And (Or phi1 phi3)
    (Not (And phi3 clock6_mulop))
    (Not (And phi1 clock18))
    ctrl-mul/div-latch)
    ctrl-latch-ops
    )))
```

Example 3-2 SLANG Description of *cm-latch-slave*

To begin, I converted phi1+phi3 to the *nand* type, again using the first theorem. By letting A* = phi1 and B* = phi3, we get that A*+B* = (AB)*, or (phi1*phi3*)*. The last two levels of logic are treated as in the example above.

Appendix D contains a complete listing of all of the control signal definitions as derived from the SLANG description, along with their equivalent *nand-nand* representation. Except in special instances, all signals were implemented using *nand-nand* logic.

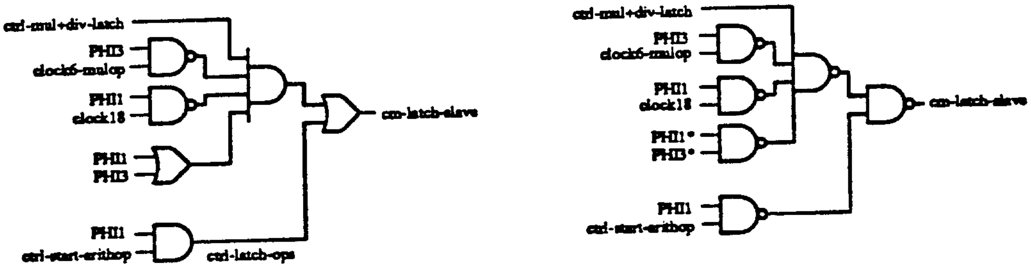


Figure 3-7 Equivalent Representations of *cm-latch-slave*

3.2.4. Clock Skew Considerations

The minimization of *clock skew* is one of the most important considerations in the design of the random control logic. Clock skew [11] is the delay in a clock signal from its point of generation to the point at

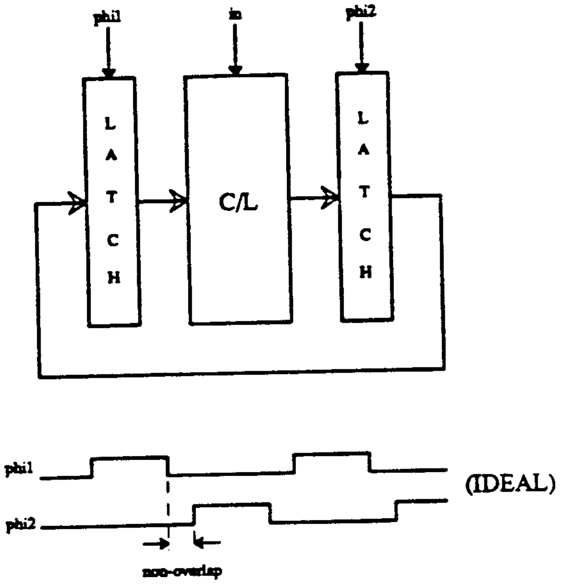


Figure 3-8 Typical Clock Skew Example with Non-Overlapping Clocks

which it is used, and is due to the resistive and capacitive delays incurred in the travel between the two points. The effects of this delay between clock signals can be clearly seen in the case of two latches, as shown in Figure 3-8. (The example shown assumes a two-phase non-overlapping clock, but the concept can easily be extended for the case of a four-phase non-overlapping clock.) Given the ideal timing diagram for phi1 and phi2, we see that both latches will never be operating simultaneously, thus providing the isolation needed for correct operation. That is, the combinational logic determines a value which is clocked into the second latch on the falling edge of phi2, and on the falling edge of phi1 the new value is clocked into the first latch where it serves as an input to the combinational logic. A problem arises only when the feedback loop to the combinational logic is closed, resulting in a form of *race condition.* In this case, it becomes possible for the combinational logic to receive as input the value which it has just determined, causing the value to be updated at least twice within one cycle. This condition could occur only if one of the signals were delayed (or in other words, the signals phi1 and phi2 were *skewed*) to such an extent that the non-overlap time between the phases disappeared.

Since it is obvious that delays are going to occur in the clock lines, the best way to handle clock skew is to balance the delays such that the relative clocking scheme is preserved. That is, given the worst-case delay of all of the control signals, it is desirable to keep the delay for all signals as closely bounded to this delay as possible, and the non-overlap time must be longer than the maximum variation in delay between the signals. For high performance, it is necessary to keep this non-overlap time as short as possible in order to minimize the clock cycle.

Most of the datapath control logic uses a type of *gated clock* [11], which is formed by the logical *and* of a control signal and one of the clock phases. The same idea applies here, though in actual implementation is it now important to consider the total delay from the generation of the clock signal, through the random logic, to each latch in the datapath. Given the huge datapaths in the FPU (> 70 bits), this delay can be considerable. That is, although the random logic may be balanced such that each logic cell has the same delay, if the delays to the last latch on the datapath which the cells are driving are significantly different clock skew can still occur. In order to take this into account, each random logic cell and its associated buffers are scaled to drive their particular load capacitances at approximately the same rate [3].

To implement the delay balancing required individual attention for each control signal along the datapath, as the capacitances varied over a huge range, and often different levels of logic were used in generating the control signals, thus providing various levels of driving ability. Appendix E contains tables of the extracted control line capacitances for the three main datapath modules, along with their chosen buffer sizes and the total delay. As shown, the largest capacitive load occurs in the multiply-divide carry-save-adder tree and is approximately 11.3 pF. The goal here then is to minimize the delay for this control signal

Table 3-2 SPICE Delays for Common Circuit Configurations

Function	Load (pF)	Buffer (λ)	$t_{P_{HL}}$ (ns)	Per Stage	$t_{P_{LH}}$ (ns)	Per Stage
2and	1.0	40	3.5	1.5+1.0+1.0	4.0	2.0+1.0+1.0
	2.0	40	4.5	2.0+1.5+1.0	5.0	2.0+1.5+1.5
	3.0	60	5.0	2.5+1.0+1.5	5.0	2.0+2.0+1.0
	4.0	60	5.5	2.5+1.5+1.5	6.0	2.5+2.0+1.5
	6.0	80	7.0	2.0+2.0+3.0	7.0	3.0+2.0+2.0
	6.0	100	6.0	2.5+2.0+1.5	6.5	3.0+2.0+1.5
2nand2nand	1.0	40	4.5	2.5+1.0+1.0	4.5	2.5+1.0+1.0
	2.0	40	5.5	2.5+1.5+1.5	5.5	2.5+1.5+1.5
	3.0	60	5.5	2.5+2.0+1.0	5.0	2.5+1.0+1.5
	4.0	60	6.0	2.5+1.5+2.0	6.0	2.5+2.0+1.5
	6.0	100	6.0	3.0+2.0+1.0	5.5	2.5+1.5+1.5
	8.0	100	7.0	3.0+2.0+2.0	7.0	3.0+2.0+2.0
	11.3	100	9.0	3.0+2.5+3.5	9.0	3.0+3.0+3.0
	11.3	160	7.5	3.5+2.0+2.0	7.0	3.0+2.5+1.5
	11.3	200	7.5	4.0+2.0+1.5	8.0	4.0+2.0+2.0
2nand3nand	1.0	40	5.0	3.0+1.0+1.0	4.5	2.5+1.0+1.0
	2.0	40	6.0	2.5+2.0+1.5	6.0	2.5+1.5+2.0
	3.0	60	6.5	3.0+2.0+1.5	5.5	2.5+1.0+2.0
	4.0	60	7.0	3.0+2.0+2.0	6.5	2.5+2.0+2.0
	6.0	100	7.0	3.5+2.5+1.0	6.0	2.5+2.0+1.5
2nand4nand	1.0	40	5.0	3.0+1.0+1.0	5.0	2.5+1.5+1.0
	2.0	40	6.5	3.0+2.0+1.5	6.0	2.5+2.0+1.5
	3.0	40	8.0	3.0+2.5+2.5	7.0	2.5+2.0+2.5
	3.0	60	7.0	3.5+2.0+1.5	6.0	2.5+2.0+1.5
	4.0	60	7.5	3.5+2.5+2.0	7.0	2.5+2.5+2.0
	6.0	60	9.0	3.5+2.5+3.0	8.0	2.5+3.0+2.5
	6.0	100	8.0	3.5+2.0+2.5	7.0	3.0+2.5+1.5
2nand5nand	1.0	40	6.0	3.5+1.0+1.5	5.0	2.5+1.5+1.0
	2.0	40	7.0	3.5+2.0+1.5	6.0	2.5+2.5+1.0
	3.0	40	8.0	3.5+2.0+2.5	7.0	2.5+2.5+2.0
	3.0	60	7.0	3.5+2.5+1.0	6.0	3.0+2.0+1.0
	4.0	60	8.0	4.0+2.0+2.0	7.0	3.0+2.5+1.5
	4.0	80	7.5	4.0+1.5+2.0	6.5	3.0+2.0+1.5
	6.0	80	9.0	4.0+2.5+2.5	7.5	3.0+2.5+2.0
	6.0	100	8.0	4.0+2.5+1.5	7.0	3.0+2.0+2.0
3nand2nand	1.0	40	4.5	2.5+1.0+1.0	4.5	2.5+1.0+1.0
	2.0	40	5.5	2.5+1.5+1.5	5.5	2.5+2.0+1.5
	3.0	60	6.0	2.5+2.0+1.5	6.0	2.5+2.0+1.5
	4.0	60	6.5	3.0+2.0+1.5	6.5	2.5+2.0+2.0
	6.0	100	7.0	3.0+2.0+2.0	6.5	3.0+2.0+1.5

(*cm-latch-master*) and to scale all other delays to this match this delay. As the datapath control line capacitances range anywhere from about 0.5 pF to 11.3 pF, this is a difficult task.

A series of SPICE simulations were exercised in order to compile Table 3-2. As shown here, a few of the more commonly used circuit types were simulated iteratively with various load capacitances and buffer sizes in order to find the configuration which most closely matched the *cm-latch-master* (2nand2nand) delay of 7.5 ns. (It is still possible to optimize this delay by increasing the strength of the first levels in order to equalize the delays between each of the stages.) Given these numbers, the cells were placed and buffers were sized accordingly, and Crystal was then used for fine-tuning the delay of each cell. It is important to note that the numbers below are only a crude approximation, as the iterations were performed at the SPICE level, and did not take into account all of the changes in area and junction capacitances for the different buffer sizes.

Since none of the global routing is complete at this point, the method above assumes that all clock inputs to the random logic arrive at the same time and that the other inputs are stable through the clock transitions. This minimizes additional skews that could be incurred by random delays in control inputs.

4. SIMULATION AND VERIFICATION

4.1. CAD Environment

As stated in the introduction to this report, much of the control had already been specified at the functional level and had been simulated using SLANG when I joined this project. My role thus involved turning the SLANG description into working layout. A progression of CAD tools was used in this process, as illustrated in Figure 4-1.

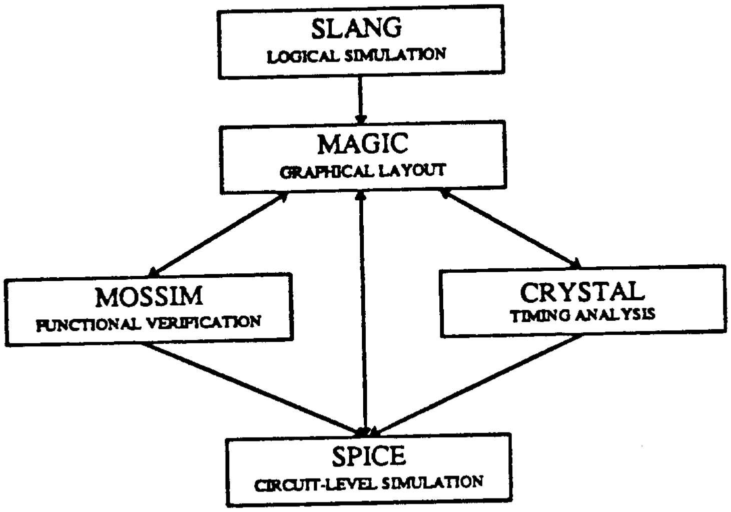


Figure 4-1 CAD Tools Used in the Control Development Cycle

The PLAs, block diagrams, and random logic definitions presented in this report were all derived from the SLANG description of the FPU. The graphical layout editor, MAGIC [14,16], was then used to implement the SLANG definitions at the layout level. Circuit descriptions of the various blocks were extracted from the MAGIC environment, and conversion programs were executed to obtain the necessary file formats for the various simulators. Three types of simulators were used in verifying the control layout: MOSSIM [4], CRYSTAL [13,16], and SPICE [20]. The first two are high-level interactive simulators, while SPICE is a low-level batch simulator. Appendix F contains a section on each of these simulators, detailing important hints on starting up the simulators and including example source files used for initializing the simulation parameters and inputs nodes.

MOSSIM is an interactive functional simulator which employs a very primitive circuit model. The simulator is based on an idea of the relative strengths and sizes of the circuit, where strength is a function of the relative width of a transistor (to all the other transistors in the circuit), and size is a function of the relative capacitance of a node. For the most accurate representation, it is desired to allow as many strength and size definitions as the circuit requires. Unfortunately, MOSSIM is limited to a total of 15 strengths and sizes, so a ratio factor is used to group transistors within a certain range of widths under the same strength, and nodes within a certain range of capacitances under the same size. This fact is especially important in dynamic or ratioed circuits. For instance, MOSSIM may consider some circuits incorrectly simply because the model has treated each of the transistor sizes as the same strength where in fact the transistors are ratioed safely, or has grouped critical nodes under the same size where in fact the charge-shared node may be quite larger than those with whom it is supposedly sharing charge. Therefore, it is important when creating the MOSSIM input file to keep the ratio factor as small as possible for the most accurate circuit representation.

The MOSSIM timing model I used assumes that each phase specified is as long as required for all the nodes to stabilize. This means that if the circuit functions successfully under MOSSIM, it should function successfully in practice, given no timing constraints (an infinite clock cycle). Once the circuit description is read into the simulator, inputs and control signals can be changed interactively, and any specified nodes may be watched. Also, intermediate nodes may be forced to a particular value, in order to isolate subsections of the circuit for further testing. There are basically only three possible logic levels: high (1), low (0), and intermediate (X). The intermediate level can mean a variety of things, such as undefined, charge sharing, short, and so on. By asserting various control signals, and varying the inputs to the circuit, we can immediately observe the levels on the output nodes in order to test the functionality of the circuit.

CRYSTAL is an interactive simulator used for timing analysis. This simulator also follows a very primitive circuit model, using signal-flow analysis to find the worst-case delay paths for the given circuit. As done in MOSSIM, nodes can be forced to a desired logic level in order to direct the signal flow along paths of interest, or to rule out impossible paths which CRYSTAL may consider. Also, specified nodes may be watched to find the worst-case delay to that node, independent of the delay of the overall circuit.

SPICE was used for simulating the very low-level circuit considerations. This includes noise margin voltages and charge sharing, low-level functional testing, latch set-up and hold times, and to provide accurate delay times for comparison with CRYSTAL's somewhat cruder model. This is particularly important as only CRYSTAL is used for timing delays of the bigger layout modules.

Most of the basic cells used in the control layout were simulated first using SPICE, giving functional verification and timing information at the lowest level of layout. Each of the functional blocks were simulated in their entirety using MOSSIM, which in particular tests the interconnections between the basic cells and any related effects. Critical paths were simulated using CRYSTAL for an over-all timing evaluation. If a problem was found using any of the simulators, it was usually uncovered and fixed at the layout level. The majority of these errors were related to labelling, particularly not labelling some power supply lines, or accidentally attaching the label of a circuit node to one of the power supply lines, such that the simulator believes it to be shorted to that line although the layout itself is correct. This happens often while editing in MAGIC, since when layout is stretched or moved the associated labels often are transferred to another layer in an intermediate stage, and are never restored to the original layer. If the layout yielded no information about the problem, SPICE was used in order to study the voltages (as a function of time) at individual nodes in greater detail.

All in all, the biggest problem I found concerning the CAD support was the inconsistency between the various tools. Each of the simulators requires a different type of input file for the circuit description, all of which are derived from the .sim file. (CRYSTAL is the friendliest here, as it actually uses the .sim file as input.) For large modules, this requires lengthy conversion times and the accumulation of several types of huge data files, each describing the same circuit! This is especially inefficient when a simulation result indicates a problem which must be changed in the layout. The circuit must then be re-extracted, reconverted, and re-read into the simulator in order to simulate the modified layout. One example I encountered. was in the CRYSTAL simulation of the datapath control delays in order to tweak the sizing of the double buffers. Each time I adjusted a buffer size I had to repeat the above procedure in order to monitor the effects of that change!

Given my experience with the above problems, I definitely recognize the need for "smarter" CAD tools. For example, one design group here has completely automated the process of random logic generation through the use of their *Design Manager* [17] and a standard cell library. A program has been written which, given a set of equations, generates a file suitable for input to the *Design Manager*, which can then be used to automatically select the appropriate standard cells from the library and access placement and routing programs to complete the layout. Another such tool is EPOXY [12], currently under design here at Berkeley. EPOXY actually takes the concept of silicon compilers one step further -- aiming to not only synthesize layout from a given circuit description but to also improve the performance of the generated layout in order to meet desired specifications. Using this tool, it will be possible to model the circuit given user-specified circuit constraints and parameters, such as desired delays and power consumption considerations, layout area, and so on; specify the layout style and technology to be used; simulate the circuit as specified and observe the effects; interactively or automatically adjust the parameters and constraints until the desired behavior is observed; and then automatically generate the layout.

4.2. Simulation Results

Each of the basic control cells have been simulated at the lowest level using SPICE; detailed results of the individual simulations are included within this report. The three control PLAs have been simulated using CRYSTAL, worst case delay times are presented in Table 2-1. MOSSIM has been used to functionally verify the major blocks of the control unit: the load-store pipeline, the cycle counter, and the random logic blocks associated with each datapath module. All of the simulation results indicate full functionality. SPICE and CRYSTAL results verify that the control unit can meet the minimum specifications of a 20ns clock phase, though to ensure against clock skew the non-overlap time should remain at 10ns.

5. CONCLUSIONS

The control unit implementation for the SPUR floating-point coprocessor has been presented. The actual control unit is divided into two main sections: interface control and datapath control. The basic blocks needed for the interface control unit were designed first, taking about 15% of the total design time; routing between the blocks required another 5% of the effort. The random logic implementation comprised about 80% of the total design time. The disparity in design effort as shown here is due to the inherent lack of regularity in the datapath control logic, as opposed to the interface control, which is based upon automatically-generated PLAs.

A major portion of the design of the random logic involved studying the alternative implementations available and developing a structured approach to the layout. Initially, a lot of time was spent trying to optimize each cell in terms of area and speed, before actually determining the placement of the cells and their load capacitances. A better approach would have been to determine the area available and then design the cells accordingly. Also, the individual speed optimization of each cell was not required, and in fact was not even desirable. As discussed in this report, clock skew between the control lines is minimized when the delays in all of the lines are balanced. Therefore, the best approach would have been to first determine the longest control line delay in the datapath, and design each cell to closely match this delay.

In retrospect, I feel that I should have implemented the random logic first, rather than the interface control. The reason for this is that in placing and simulating the random logic and datapath, I obtained a much better feel for the layout style of the other team members, design issues they considered, how the control logic interfaces with the datapath, and so on. With this in mind, the implementation of the interface control would have been much easier.

The rapid advances in VLSI technology have led to the proliferation of several CAD tools to aid in the various steps of chip development. A lot of tedious and error-prone functions, such as routing and PLA generation, can now be performed automatically. However, no such tools were available to SPUR for aiding in the design, generation, placement, and optimization of the random control logic. Much of the design effort above could have been avoided if more sophisticated CAD tools had been available.

6. ACKNOWLEDGEMENTS

Members of the FPU design team, B.K. Bose and Corinna Lee, provided invaluable assistance. In particular, I would like to thank B.K. for sharing his ideas, philosophies, and especially his friendship during the past year. I would like to thank my advisors, David Patterson and David Hodges, for their willingness to read this report quickly, allowing me to complete this project on time. I would also like to acknowledge Sandia National Laboratories, in particular Art Verardo, for making this year possible. Special thanks go to all of my friends, who provided steady support and encouragement. Principal funding for the SPUR project is provided by DARPA, under contract N00039-85-C-0269.

7. REFERENCES

[1] Adams, G., B.K. Bose, L. Pei, and A. Wang, "The Design of a Floating-Point Processor Unit", *Proceedings of CS292i: Implementation of VLSI Systems*, Ed: R.H. Katz, University of California, Berkeley, September 1985.

[2] Bose, B.K., Private Communication, University of California, Berkeley, 1987.

[3] Bose, B.K., L. Pei, G.S. Taylor, and D.A. Patterson, "Fast Multiply and Divide for a VLSI Floating-Point Unit", *Proceedings of the 8th Symposium on Computer Arithmetic*, Ed: M. Irwin and R. Stefanelli, Como, Italy, May 19-21, 1987, pp. 87-94.

[4] Bryant, R., M. Schuster, and D. Whiting, "MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual", Jan. 1983 (Revised May 1986).

[5] Duncombe, R., "The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLSI Multiprocessor", *University of California, Berkeley, Master's Report*, Report No. UCB/CSD 87/307, Aug. 1986, pp. 19-44.

[6] Hansen, P.M., S.I. Kong, "SPUR Coprocessor Interface Description", *Computer Science Division Technical Report*, Report No. UCB/CSD 87/308, University of California, Berkeley, October 1986.

[7] Hill, M. et. al., "Design Decisions in SPUR", *IEEE Computer*, Vol. 19, No. 10., Nov. 1986, pp. 8-22.

[8] Lee, C., "Description of the SPUR Floating-Point Unit", *University of California, Berkeley, Internal Working Document*, May 1986.

[9] McCluskey, E., *Logic Design Principles with Emphasis on Testable SemiCustom Circuits*, New Jersey: Prentice-Hall, 1986.

[10] Mohsen, A., C. Mead, "Delay-Time Optimization for Driving and Sensing of Signals on High-Capacitance Paths of VLSI Systems", *IEEE J. Solid-State Circuits*, Vol. SC-14, Apr. 1979, pp. 462-470.

[11] Mukherjee, A., *Introduction to nMOS & CMOS VLSI Systems Design*, New Jersey: Prentice-Hall, 1986.

[12] Obermeier, F., "EPOXY: Electrical and Physical Layout Optimizer", *EECS/ERL 1987 Research Summary*, Department of Electrical Engineering and Computer Sciences, Electronics Research Laboratory, Industrial Liaison Program, University of California, Berkeley, 1987, pp. 167-168.

[13] Ousterhout, J.K., "A Switch-Level Timing Verifier for Digital MOS VLSI", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-4, No. 3, July 1985, pp. 336-348.

[14] Ousterhout, J.K., G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor, "The Magic VLSI Layout System", *IEEE Design & Test of Computers*, Feb. 1985.

[15] Patterson, D., "Reduced Instruction Set Computers", *Communications of the ACM*, Vol. 28, No. 1, Jan. 1985, pp. 8-21.

[16] Scott. W., R.N. Mayo, G.T. Hamachi, and J.K. Ousterhout, *1986 VLSI Tools: Still More Works by the Original Artists*, Computer Science Division, Department of Electrical Engineering & Computer Science, University of California, Berkeley, Jan. 1986.

[17] Shung, C., R. Jain, *Design Manager User's Manual*, Department of Electrical Engineering & Computer Science, University of California, Berkeley, July 1987.

[18] Van Dyke, K.S., "SLANG: A Logic Simulation Language", *University of California, Berkeley, Master's Report*, June 1982.

[19] Van Dyke, K.S., J.K. Foderaro, "SLANG Slinger's Cyclopedia", in R.N. Mayo, J.K. Ousterhout, and W.S. Scott, *1983 VLSI Tools: Selected Works by the Original Artists*, Computer Science Division, Department of Electrical Engineering & Computer Science, University of California, Berkeley, Report No. UCB/CSD 83/115, March 1983.

[20] Vladimirescu, A., K. Zhang, A.R. Newton, D.O. Pederson, and A.Sangiovanni-Vincentelli, "SPICE Version 2G User's Guide", Department of Electrical Engineering & Computer Science, University of California, Berkeley, Aug. 1981.

APPENDIX A: PLA DEFINITIONS

Table A-A1 Instruction PLA Input and Output Definitions

Type	Signal Name
INPUT	instr-OPCODE<6>
INPUT	instr-OPCODE<5>
INPUT	instr-OPCODE<4>
INPUT	instr-OPCODE<3>
INPUT	instr-OPCODE<2>
INPUT	instr-OPCODE<1>
INPUT	instr-OPCODE<0>
OUTPUT	ctrl-TrapRecvd
OUTPUT	instr-ldext1
OUTPUT	instr-ldext2
OUTPUT	instr-lddbl
OUTPUT	instr-ldsgl
OUTPUT	instr-stext1
OUTPUT	instr-stext2
OUTPUT	instr-stdbl
OUTPUT	instr-stsgl
OUTPUT	instr-addop
OUTPUT	instr-subop
OUTPUT	instr-mulop
OUTPUT	instr-divop
OUTPUT	instr-cvtsop
OUTPUT	instr-cvtdop
OUTPUT	instr-cmpop
OUTPUT	instr-fabsop
OUTPUT	instr-fnegop
OUTPUT	instr-fmovop
OUTPUT	instr-loadop
OUTPUT	instr-storeop
OUTPUT	instr-MD/AS
OUTPUT	instr-AS/MD
OUTPUT	instr-cvrtop
OUTPUT	instr-fpuArithop

Table A-A2 IFSM PLA Input and Output Definitions

Type	Signal Name
INPUT	stp2
INPUT	stp1
INPUT	stp0
INPUT	ifsm-STOP
INPUT	ifsm-TrapRecvd
INPUT	ifsm-fpuSuscond
INPUT	ifsm-start-arithop
OUTPUT	stn2
OUTPUT	stn1
OUTPUT	stn0
OUTPUT	st-to-write
OUTPUT	fpuBusy
OUTPUT	ctrl-latch-fpuBusy
OUTPUT	ctrl-cycleclock-clearcond

Table A-A3 Arithmetic PLA Input and Output Definitions

Type	Signal Name
INPUT	arith-cycleclock<4>
INPUT	arith-cycleclock<3>
INPUT	arith-cycleclock<2>
INPUT	arith-cycleclock<1>
INPUT	arith-cycleclock<0>
INPUT	arith-subop
INPUT	arith-mulop
INPUT	arith-divop
INPUT	arith-fabsop
INPUT	arith-fnegop
INPUT	arith-fmovop
INPUT	arith-fcvrtop
INPUT	arith-MD/AS
INPUT	arith-AS/MD
INPUT	arith-arithop
INPUT	arith-opexcept-detect
INPUT	arith-sign-muldiv
INPUT	arith-expn-BgtA
OUTPUT	clock1
OUTPUT	clock2
OUTPUT	clock3
OUTPUT	clock4
OUTPUT	clock5
OUTPUT	clock18
OUTPUT	clock19
OUTPUT	clock20
OUTPUT	clock1_AS
OUTPUT	clock1_MD
OUTPUT	clock1_cvrtop
OUTPUT	clock2_AS
OUTPUT	clock6_mulop
OUTPUT	clock7_mulop
OUTPUT	ctrl-STOP
OUTPUT	ctrl-ASMD
OUTPUT	ce-adder-first
OUTPUT	ce-pass-expna
OUTPUT	cf-adder-AS

APPENDIX B: SCHEMATICS

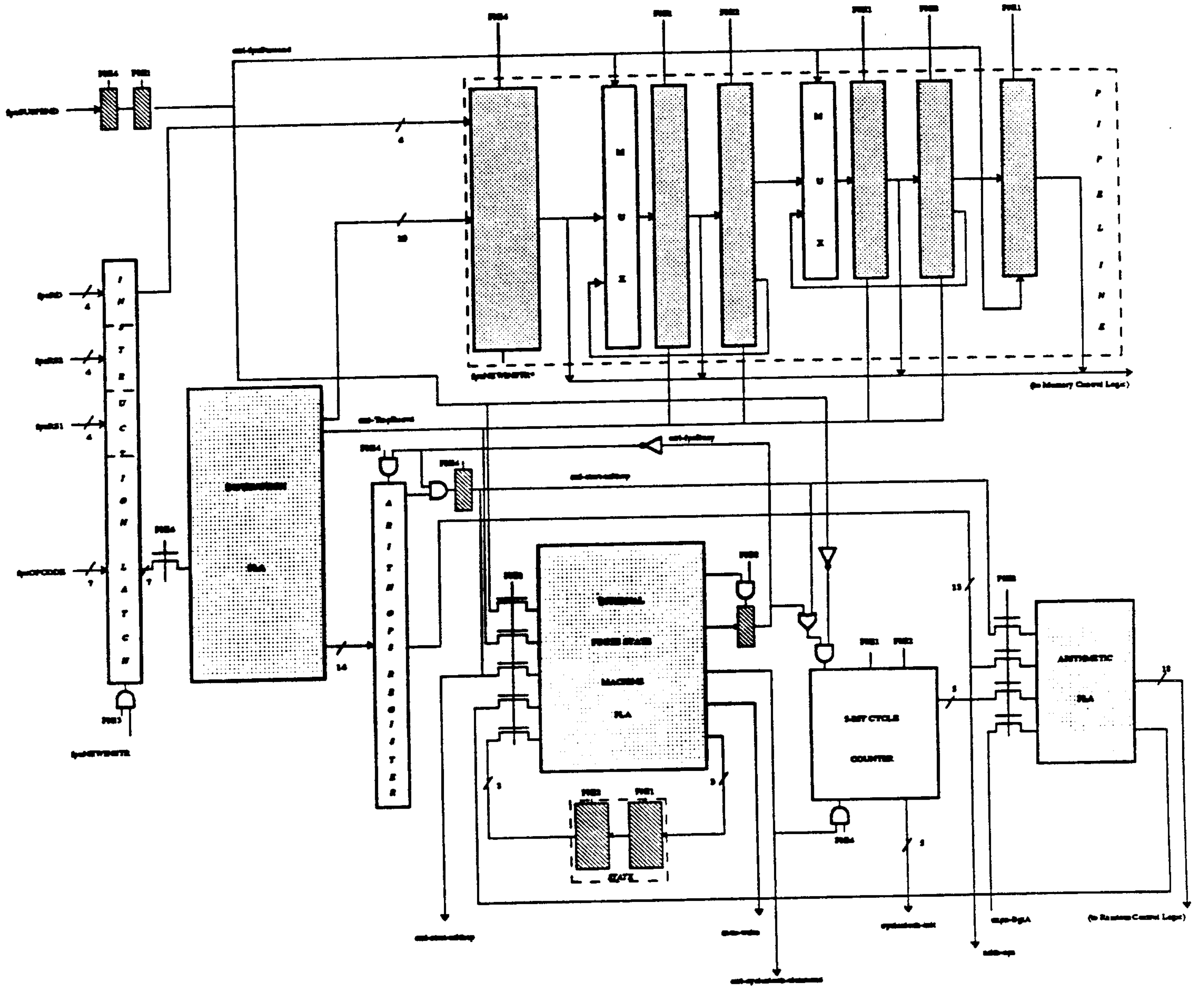

4
20
M U X
M U X
P I P E L I N E
(to Memory Control Logic)
4
4
4
7
7
I N S T R U C T I O N L A T C H
PLA
14
A R I T H O P S R E G I S T E R
FINITE STATE
MACHINE
PLA
3
3
STATE
CYCLE
COUNTER
3
3
13
ARITHMETIC
PLA
12
(to Random Control Logic)

Figure A-B1 Detailed Diagram of the Main Control Unit

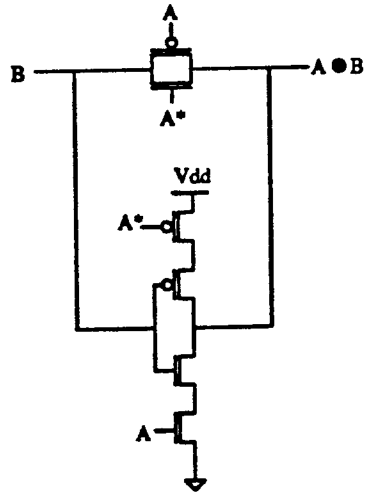


Figure A-B2 Exclusive-Or Implementation

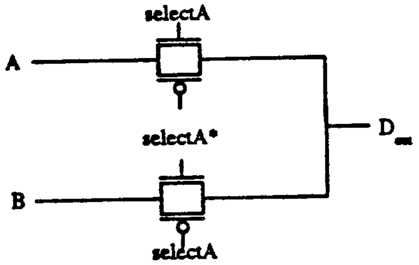


Figure A-B3 2:1 Static Mux

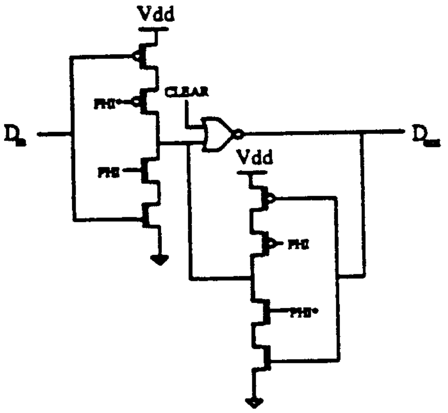


Figure A-B4 Static Latch with Asynchronous Clear

APPENDIX C: LAYOUT PLOTS

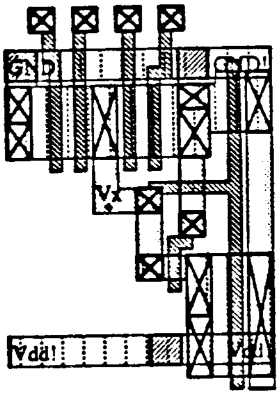


Figure A-C1 2and2or

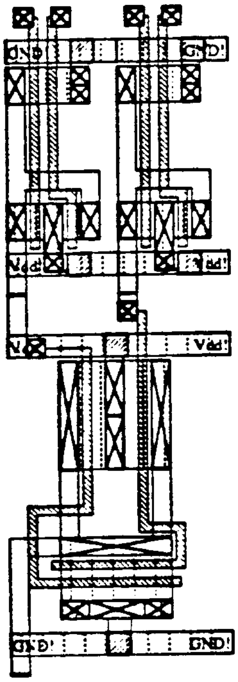


Figure A-C2 2nand2nand

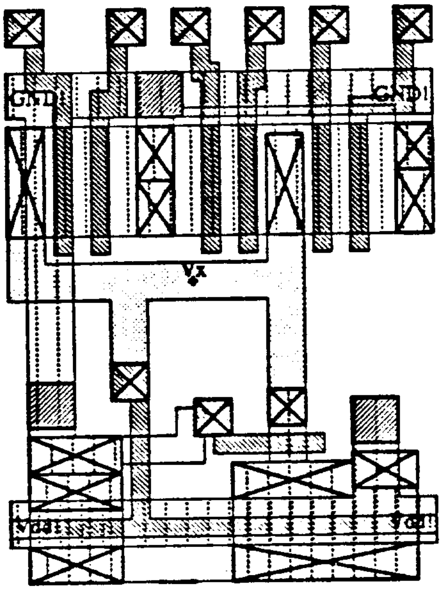


Figure A-C3 2and3or

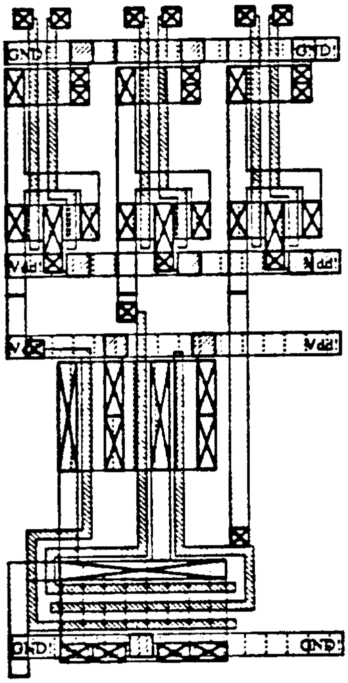


Figure A-C4 2nand3nand

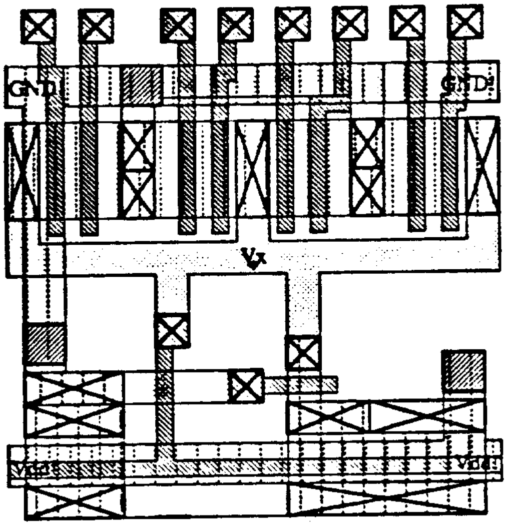


Figure A-C5 2and4or

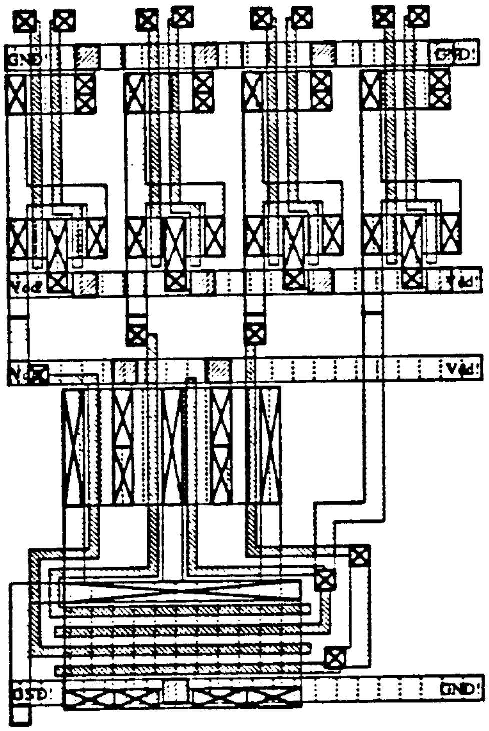


Figure A-C6 2nand4nand

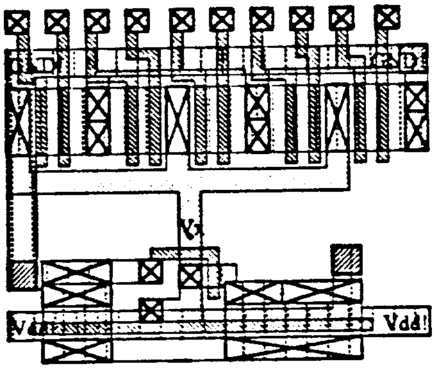


Figure A-C7 2and5or

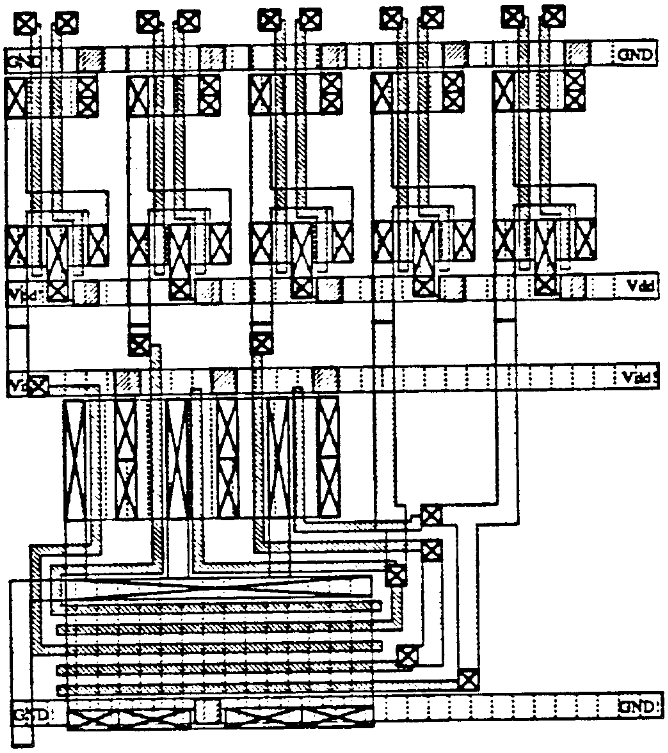


Figure A-C8 2nand5nand

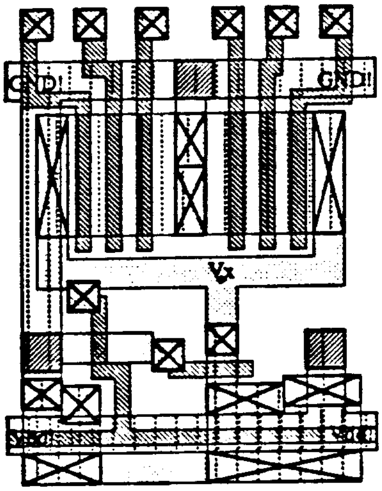


Figure A-C9 3and2or

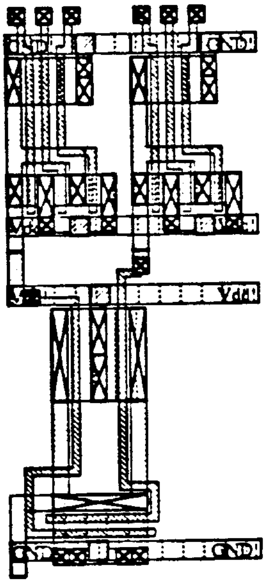


Figure A-C10 3nand2nand

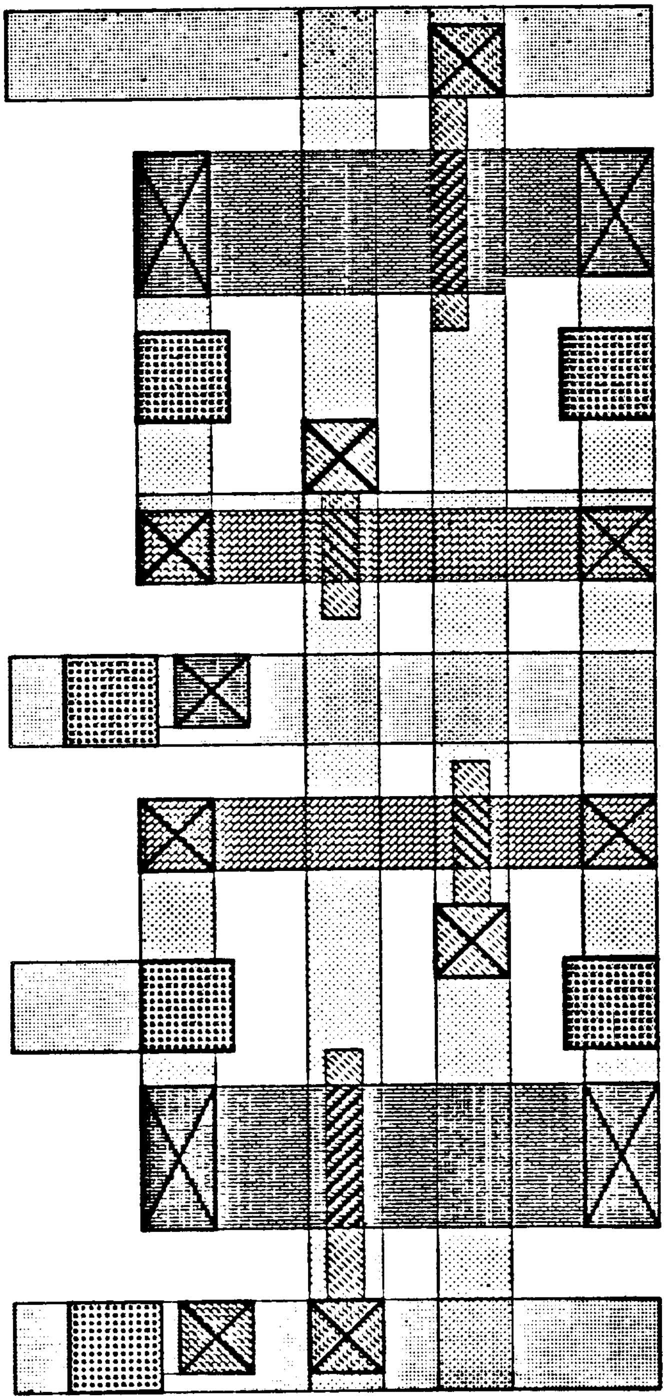


Figure A-C11 2:1 Static Mux

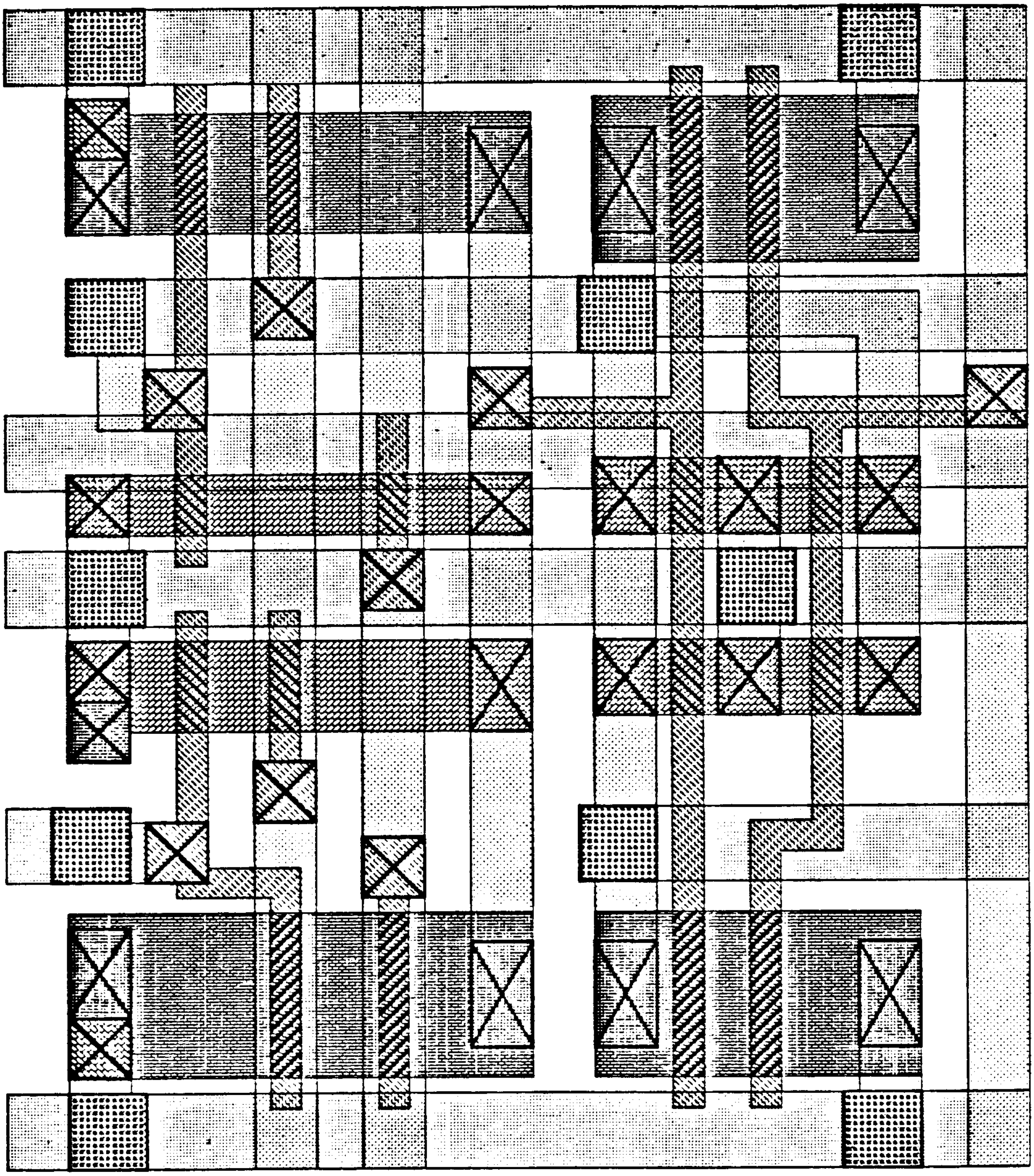


Figure A-C12 Static Latch With Asynchronous Clear

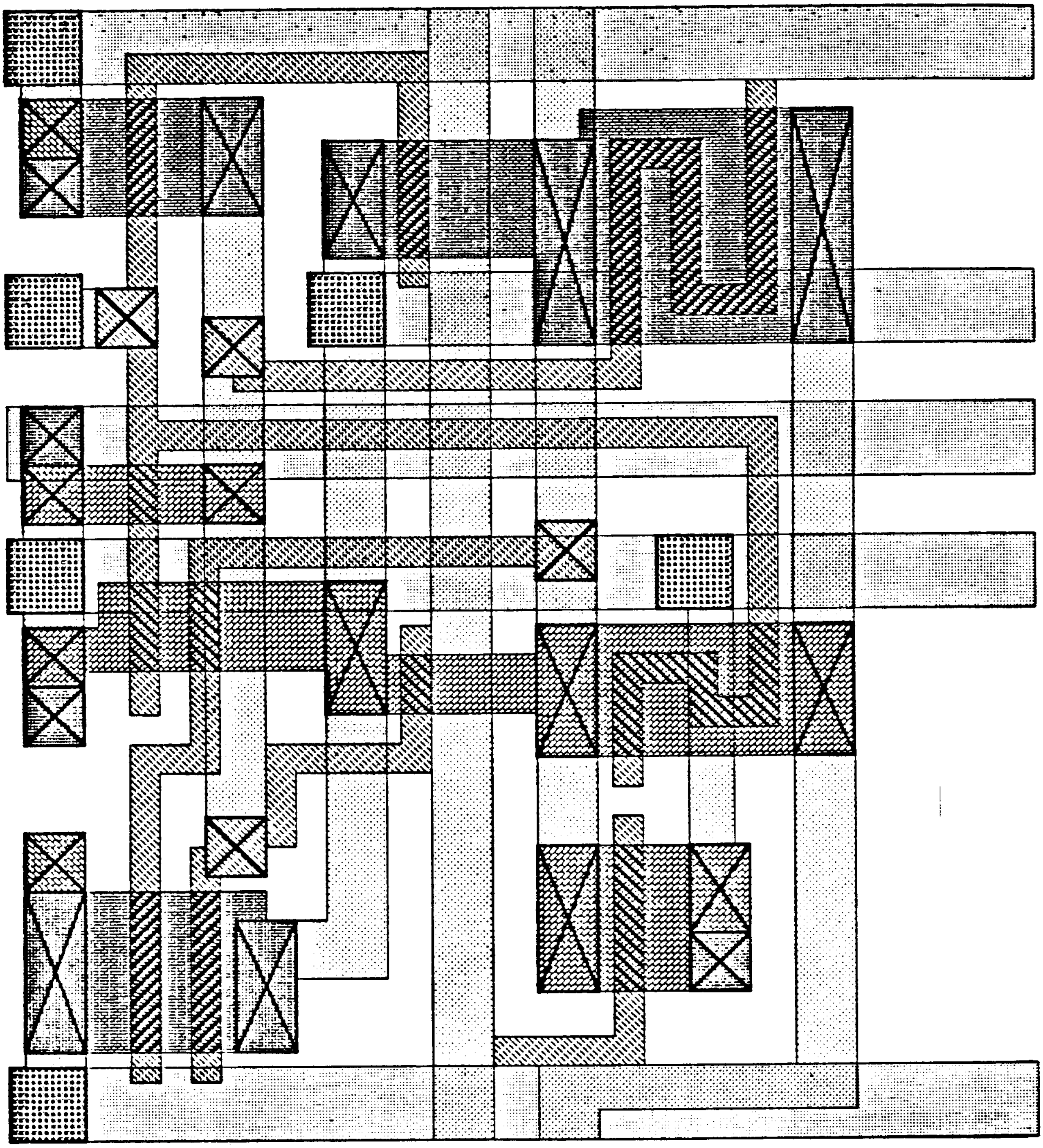


Figure A-C13 Increment Bit

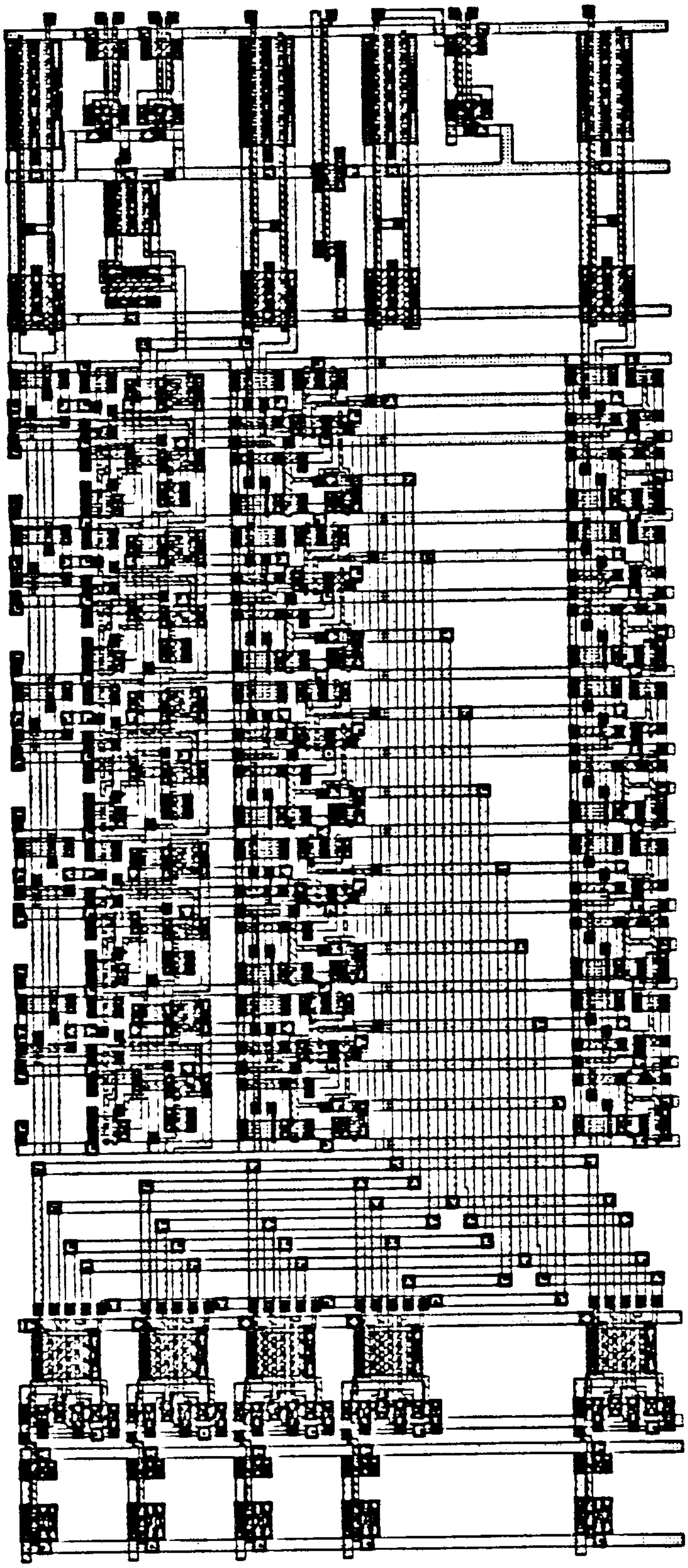


Figure A-C14 5-Bit Cycle Counter With *cycleclock-init* Logic

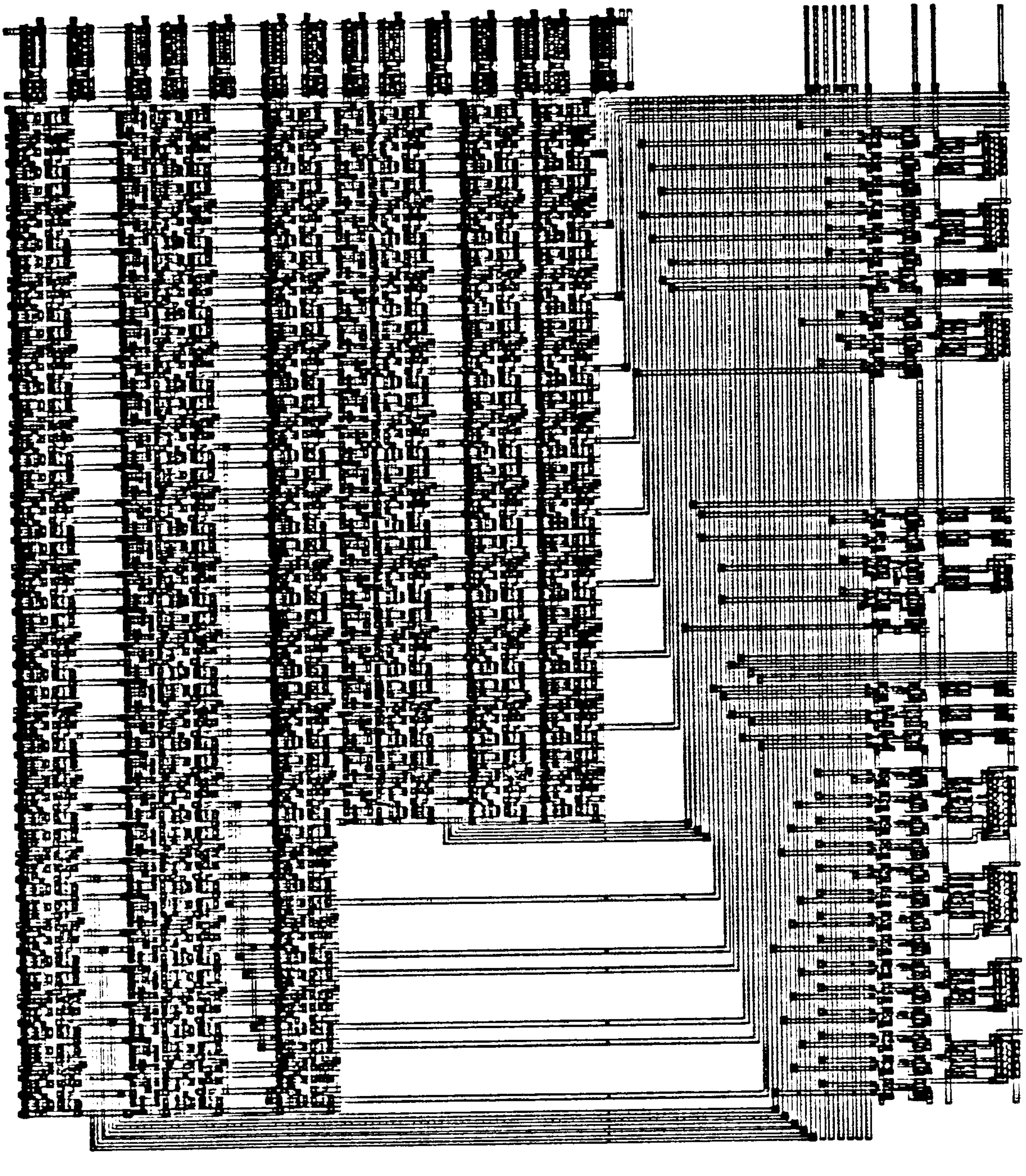


Figure A-C15 Load-Store Pipeline With Memory Control Logic

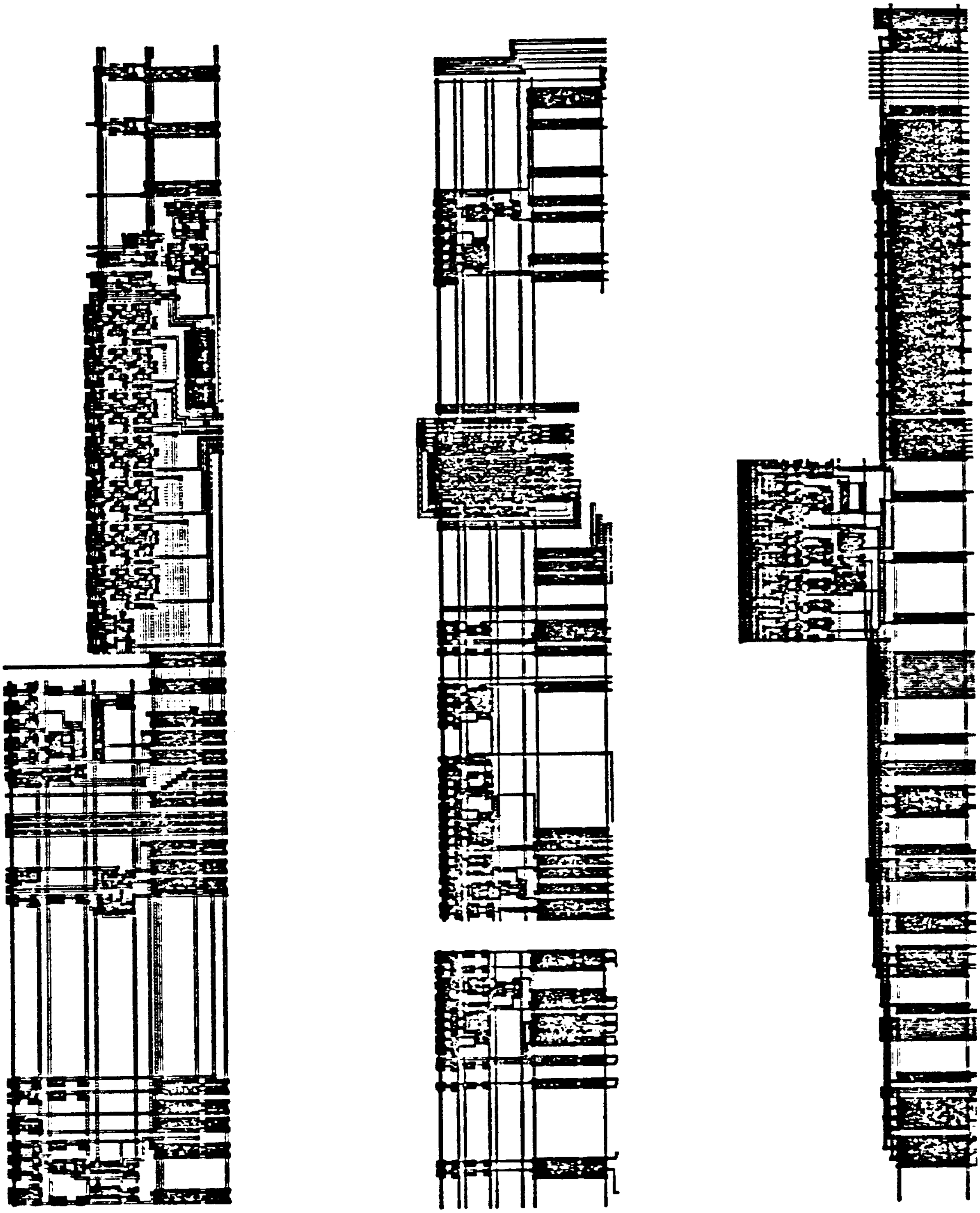


Figure A-C16 Exponent, Fraction, and Multiply-Divide Random Logic

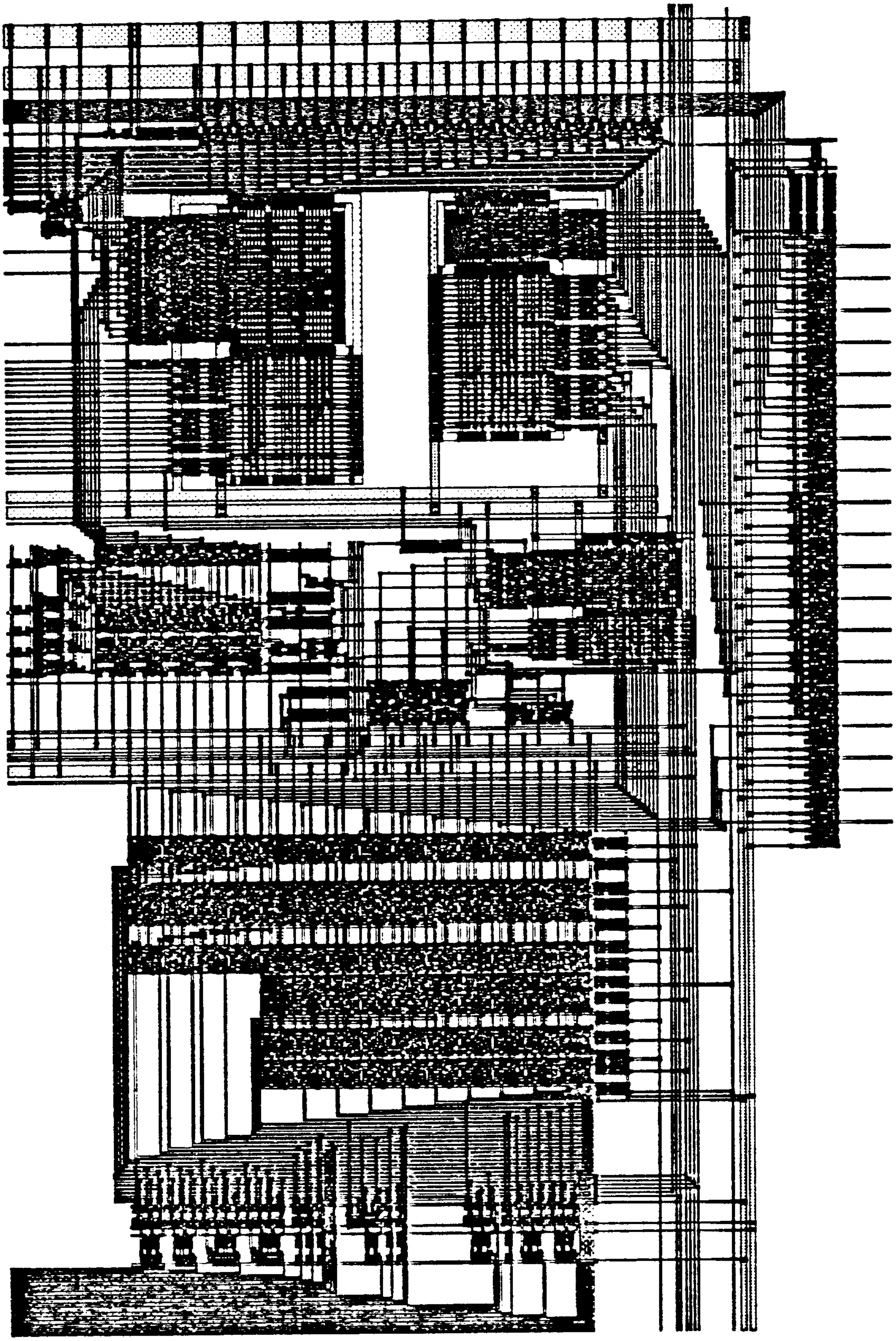
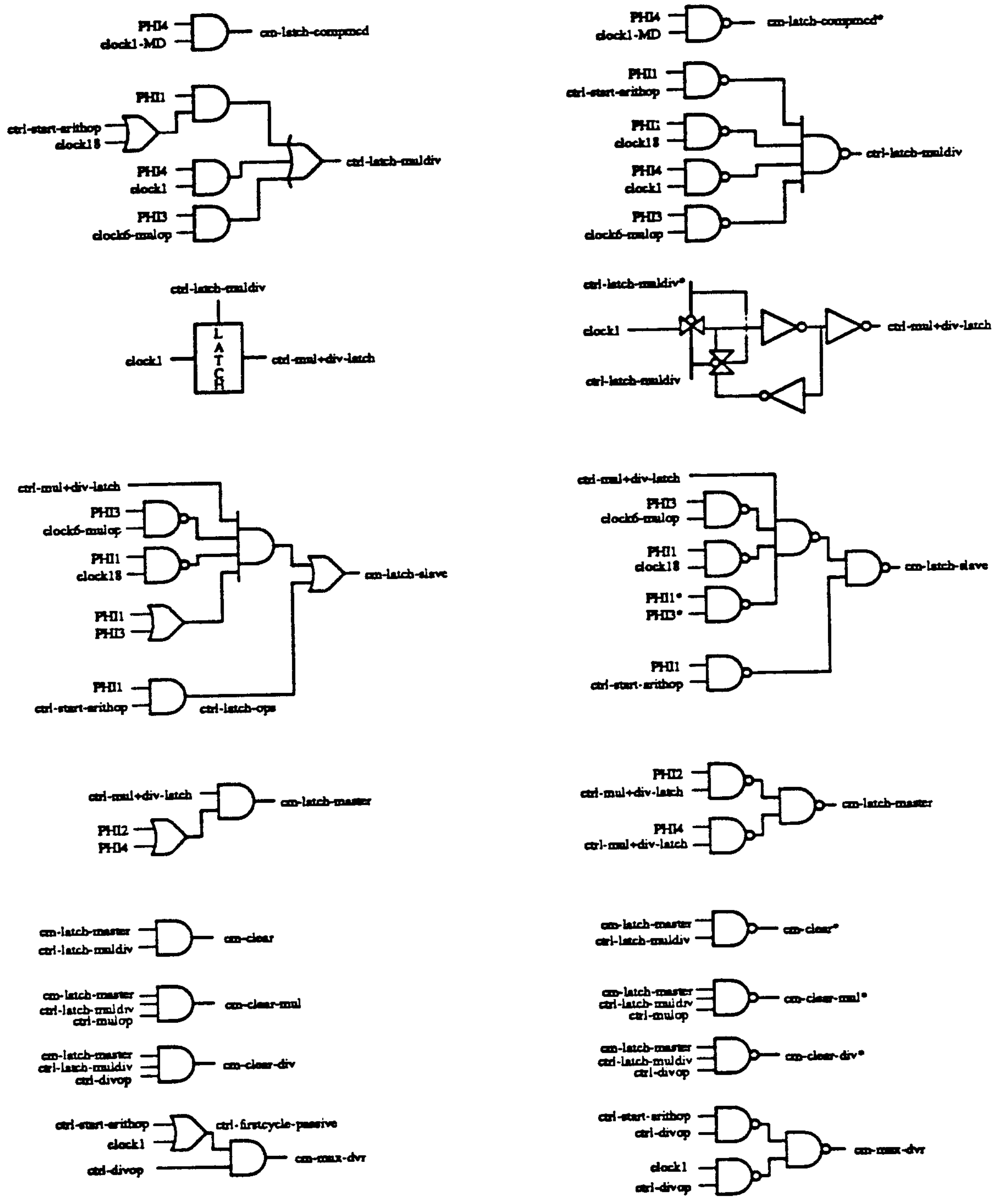


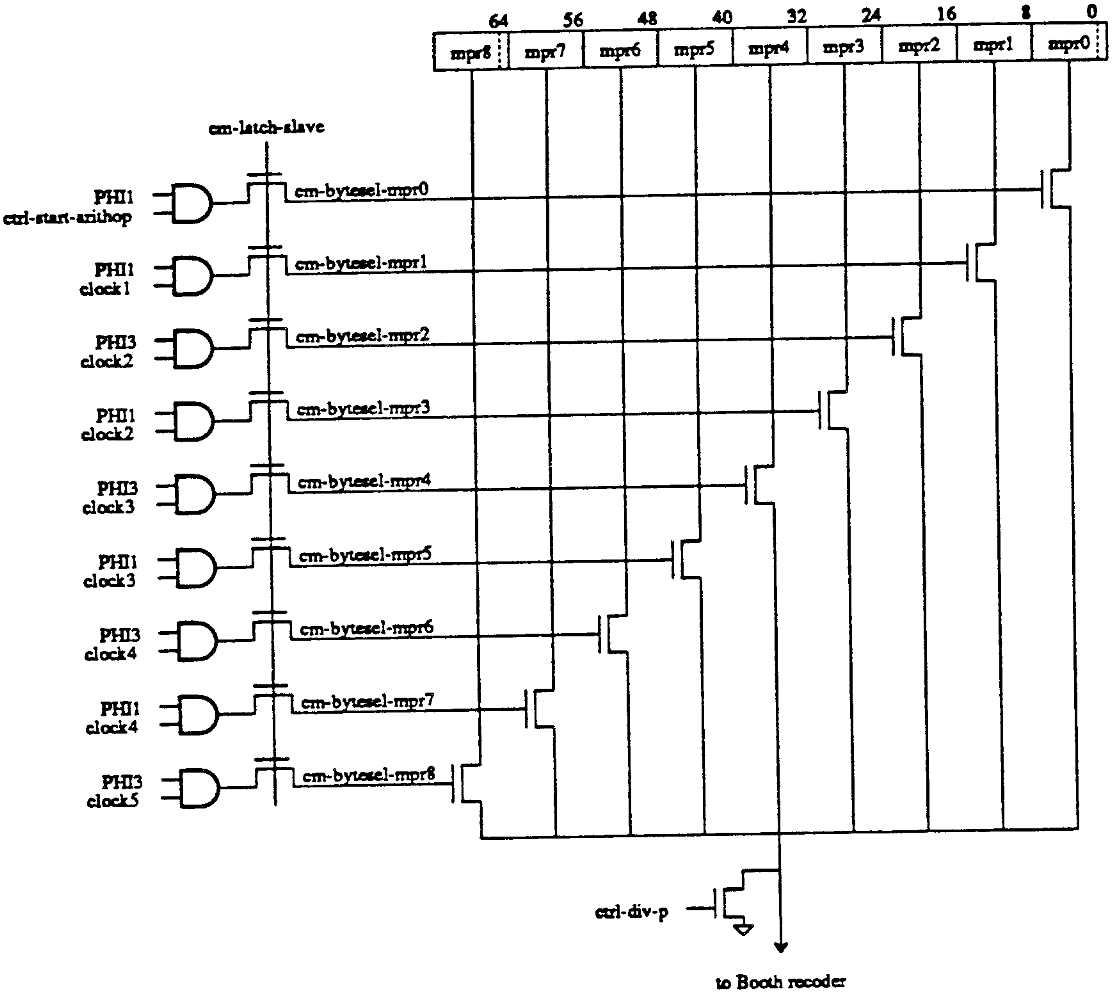
Figure A-C17 Interface Control Unit

APPENDIX D: CONTROL SIGNAL DEFINITIONS AND IMPLEMENTATION

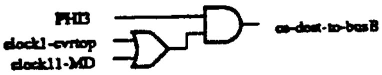
MULTIPLY-DIVIDE CONTROL SIGNALS

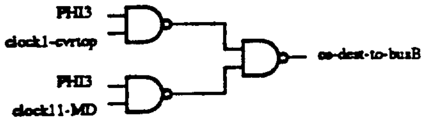


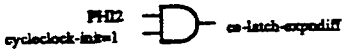
MULTIPLIER BYTE SELECT CONTROL SIGNALS

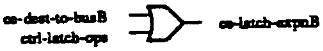


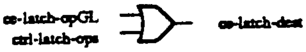
EXPONENT CONTROL SIGNALS

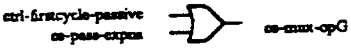


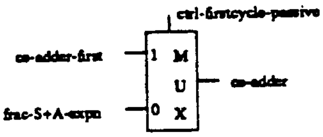


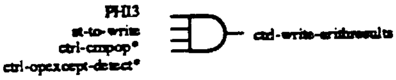


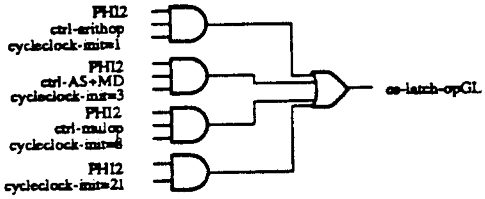


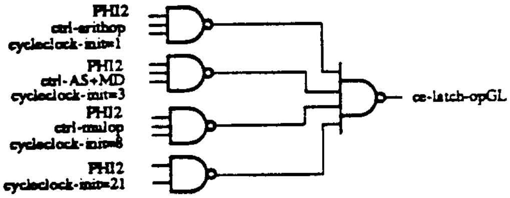




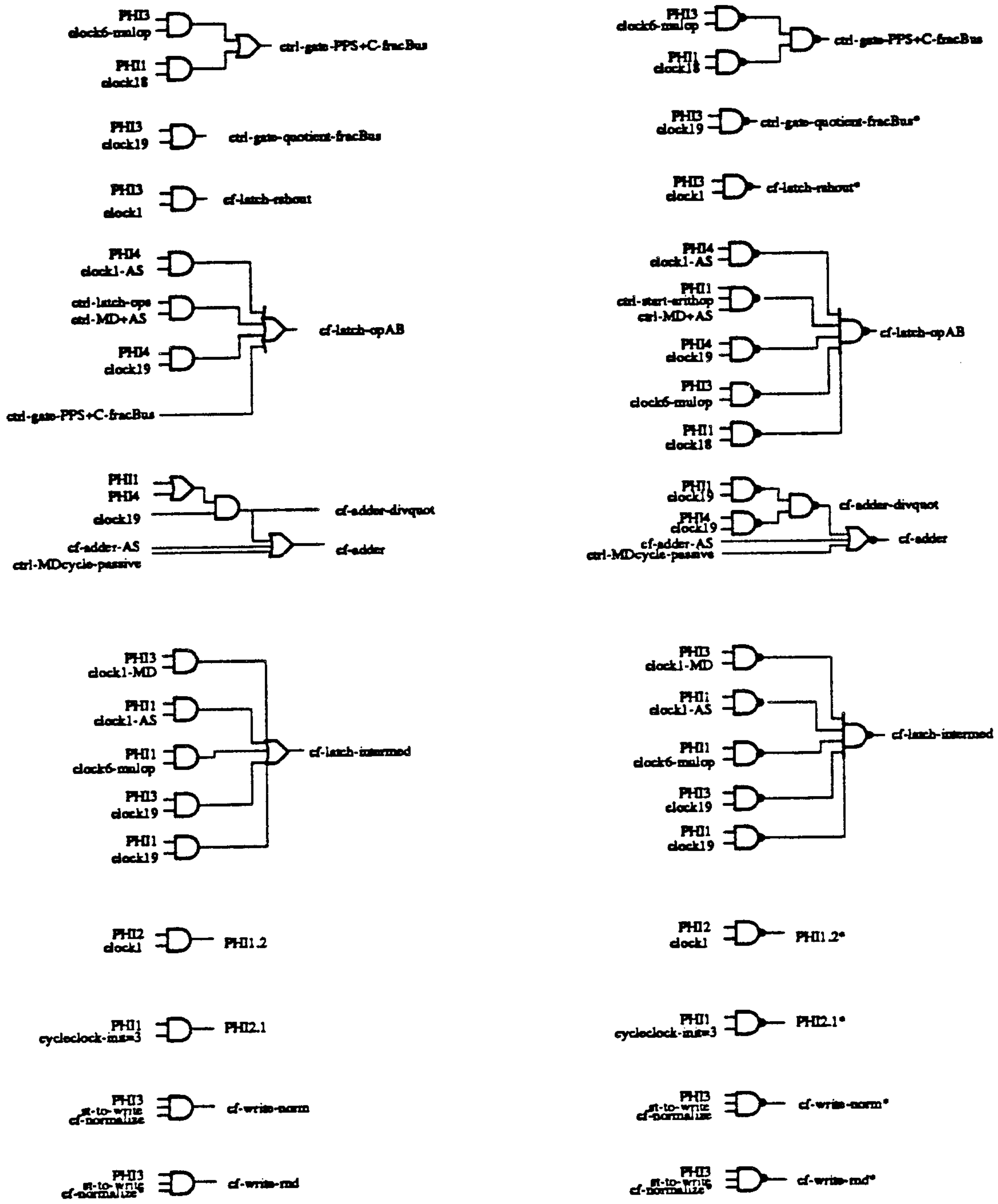


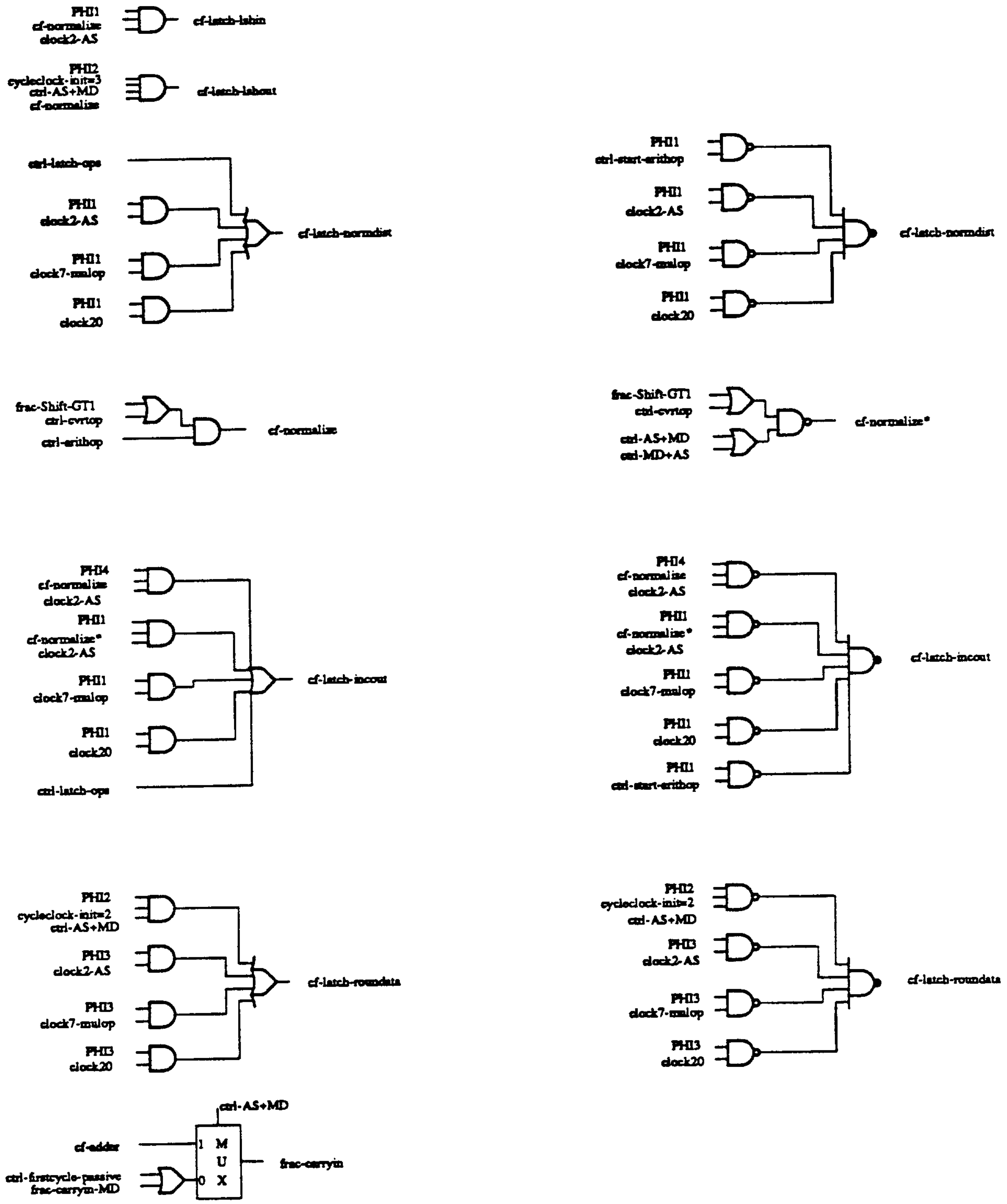




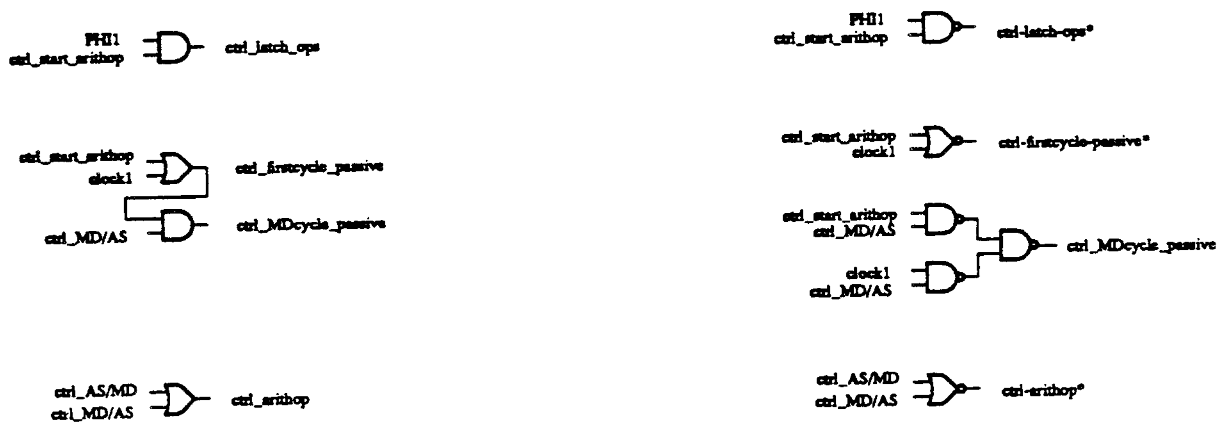


FRACTION CONTROL SIGNALS

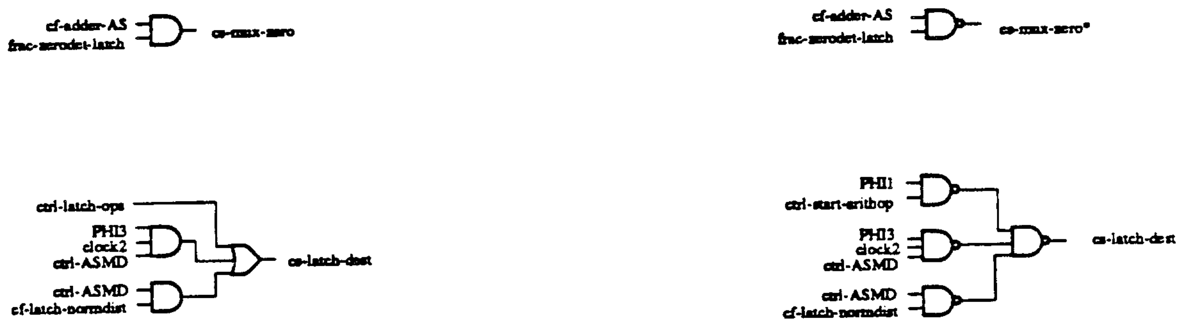



PHI1
cf-normalize
clock2-AS
cf-latch-lshin
PHI2
cycleclock-init=3
ctrl-AS+MD
cf-normalize
cf-latch-lshout
ctrl-latch-ops
PHI1
clock2-AS
PHI1
clock7-mulop
PHI1
clock20
cf-latch-normdist
PHI1
ctrl-start-arithop
PHI1
clock2-AS
PHI1
clock7-mulop
PHI1
clock20
cf-latch-normdist
frac-Shift-GT1
ctrl-cvrtop
ctrl-arithop
cf-normalize
frac-Shift-GT1
ctrl-cvrtop
ctrl-AS+MD
ctrl-MD+AS
cf-normalize*
PHI4
cf-normalize
clock2-AS
PHI1
cf-normalize*
clock2-AS
PHI1
clock7-mulop
PHI1
clock20
ctrl-latch-ops
cf-latch-incout
PHI4
cf-normalize
clock2-AS
PHI1
cf-normalize*
clock2-AS
PHI1
clock7-mulop
PHI1
clock20
PHI1
ctrl-start-arithop
cf-latch-incout
PHI2
cycleclock-init=2
ctrl-AS+MD
PHI3
clock2-AS
PHI3
clock7-mulop
PHI3
clock20
cf-latch-roundata
PHI2
cycleclock-init=2
ctrl-AS+MD
PHI3
clock2-AS
PHI3
clock7-mulop
PHI3
clock20
cf-latch-roundata
ctrl-AS+MD
cf-adder
1
M
U
X
0
ctrl-firstcycle-passive
frac-carryin-MD
frac-carryin

MISCELLANEOUS CONTROL SIGNALS

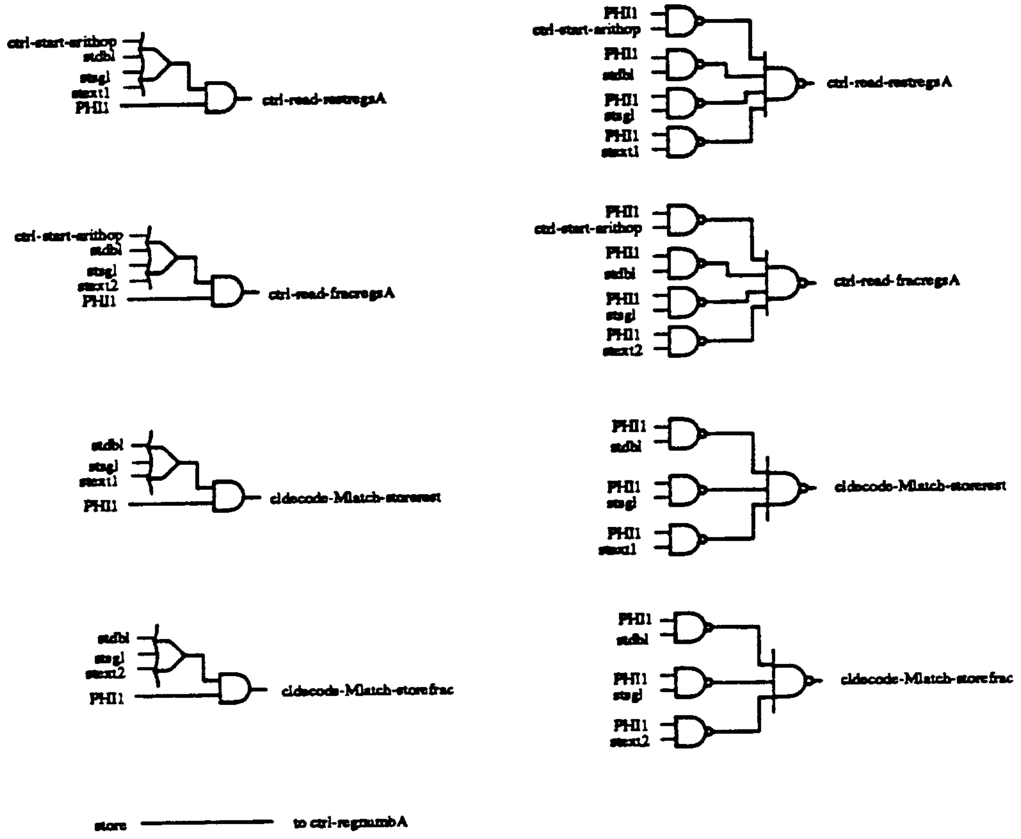


SIGN BOX CONTROL SIGNALS

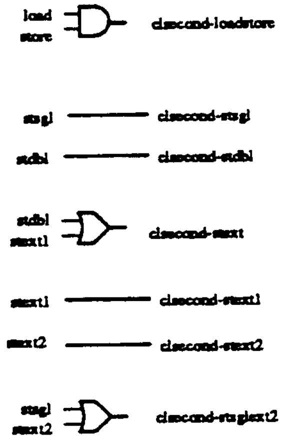


MEMORY CONTROL SIGNALS

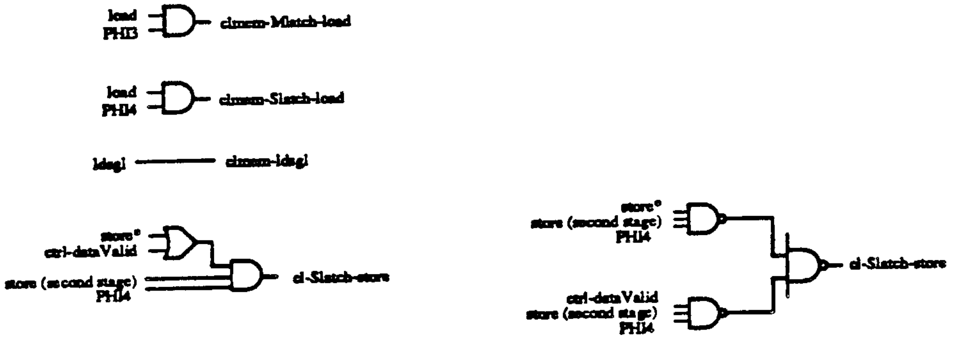
DECODE STAGE



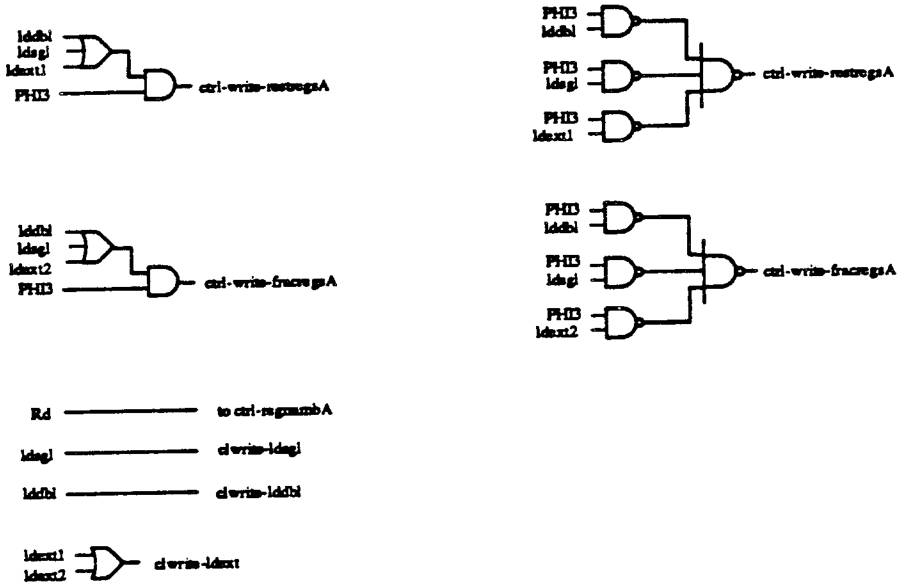
SECOND STAGE



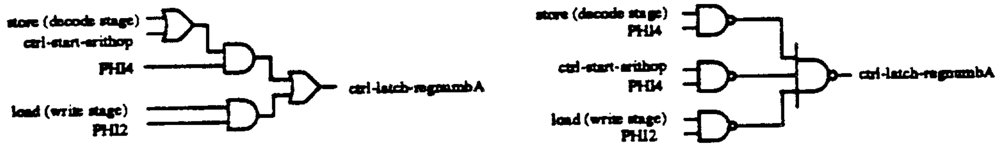
MEMORY STAGE



WRITE STAGE



MISCELLANEOUS SIGNALS



APPENDIX E: DATAPATH CAPACITANCES, BUFFER SIZES, AND DELAY TIMES

WORST CASE CONTROL INFORMATION FOR EXPONENT LAYOUT						
BLOCK	CELL	SIGNAL (layout name)	Cap (fF)	Buffer	Delay (ns)	Label
E1	EAbusdr	ctrl-read-regsB* (en*)	1549	40	3.38	1
E1	EBbusdr	ctrl-read-regsB* (en*)	1549	40	3.32	2
E1	ebusBinv	ctrl-write-arithresults* (en*)	1549	40	5.00	3
AsubB17	AbarB17	PHI1 (phi)	699	40	1.29	4
AsubB17	AbarB17	PHI1* (phi*)	939	40	0.57	5
E2	MuxBgtA17	expn-BgtA* (BgtA*)	869	40	10.47	6
E2	MuxBgtA17	expn-BgtA (BgtA)	852	40	9.58	7
E2	Ege128	ce-latch-expndiff (phi)	515	40	5.20	8
BsubA17	AbarB	PHI1 (phi)	699	40	1.29	10
BsubA17	AbarB	PHI1* (phi*)	939	40	0.59	11
E3	EAlatch	ctrl-latch-ops (phi)	854	40	5.06	12
E3	EAlatch	ctrl-latch-ops* (phi*)	905	40	4.13	13
E3	EBlatch	ce-latch-expnB (phi)	854	40	6.23	14
E3	EBlatch	ce-latch-expnB* (phi*)	905	40	5.84	15
E3	MuxOPG	ce-mux-OPG (EB>EA)	852	40	2.24	16
E3	MuxOPG	ce-mux-OPG* (EB>EA*)	835	40	2.83	17
E3	latchOPG	ce-latch-opGL (phi)	597	40	6.89	18
E3	latchOPG	ce-latch-opGL* (phi*)	734	40	6.16	19
E3	MuxOPL1	ctrl-MD+AS* (MD+AS*)	858	40	1.24	20
E3	MuxOPL1	ctrl-MD+AS (MD+AS)	814	40	0.46	21
E3	MuxOPL2	ctrl-firstcycle-passive* (1cyc*)	902	40	4.05	22
E3	MuxOPL2	ctrl-firstcycle-passive (1cyc)	835	40	3.18	23
E3	latchOPL	ce-latch-opGL (phi)	597	40	6.18	24
E3	latchOPL	ce-latch-opGL* (phi*)	734	40	5.49	25
EGsubEL	SubXor17	ce-adder* (CEsub)	1330	40	5.88	26
E4	MuxEdest1	ctrl-arithop (arith)	852	40	4.13	27
E4	MuxEdest1	ctrl-arithop* (arith*)	835	40	3.26	28
E4	Mux17Edest2	frac-zerodet (0det)	890	40	1.33	29
E4	Mux17Edest2	frac-zerodet* (0det*)	822	40	0.52	30
E4	MuxEdest3	ctrl-firstcycle-passive (1cyc)	852	40	4.05	31
E4	MuxEdest3	ctrl-firstcycle-passive* (1cyc*)	835	40	3.18	32
E4	Ereslatch	ce-latch-dest (phi)	597	40	7.44	33
E4	Ereslatch	ce-latch-dest* (phi*)	734	40	7.96	34
E4	Eresbusdr	(ce-write-to-busB)* (en*)	1642	40	6.50	35

WORST CASE CONTROL INFORMATION FOR FRACTION LAYOUT

BLOCK	CELL	SIGNAL (layout name)	Cap (fF)	Buffer	Delay (ns)	Label
F1	FAbusdr	ctrl-read-regsB* (en*)	5570	60	5.35	1
F1	FBbusdr	ctrl-read-regsB* (en*)	5570	60	5.20	2
F1	fbusBinv	ctrl-write-arithresults* (en*)	5570	60	7.96	3
F1	FlatchA	ctrl-latch-ops (phi)	2947	60	6.34	4
F1	FlatchA	ctrl-latch-ops* (phi*)	3203	60	4.43	5
F1	FlatchB	ctrl-latch-ops (phi)	2947	60	6.34	6
F1	FlatchB	ctrl-latch-ops* (phi*)	3203	60	4.44	7
F1	muxFG	expn-shfA-frac (CeBgtA)	2851	60	1.07	8
F1	muxFG	expn-shfA-frac* (CeBgtA*)	2819	60	2.79	9
F1	muxFG	expn-shfA-frac (CeBgtA)	2851	60	2.79	10
F1	muxFG	expn-shfA-frac* (CeBgtA*)	2819	60	1.07	11
F1	fmuxB	ctrl-AS/MD (AS+MD)	2881	60	1.11	12
F1	fmuxB	ctrl-AS/MD* (AS+MD*)	2945	60	2.85	13
F1	flatchOPB	cf-latch-opAB (phi)	2947	60	8.40	14
F1	flatchOPB	cf-latch-opAB* (phi*)	3203	60	6.50	15
F1	lshfoutbusdr	cf-write-norm (en*)	5197	60	4.60	16
F1	flatlshfout	cf-latch-lshout (phi)	2497	60	2.04	17
F1	flatlshfout	cf-latch-lshout* (phi*)	3075	60	9.42	18
F1	flatlshfout	PHI1.2 (phi1.2)	4161	60	8.27	19
F2	flatchrshfout	cf-latch-rshout (phi)	2146	60	5.85	20
F2	flatchrshfout	cf-latch-rshout* (phi*)	3021	60	4.36	21
F2	flatchrshfout	PHI2.1 (phi2.1)	4223	60	2.07	22
F2	flatchlshfin	cf-latch-lshin (phi)	2242	60	3.16	23
F2	flatchlshfin	cf-latch-lshin* (phi*)	2627	60	4.26	24
F2	fmuxA1	ctrl-MDcycle-passive (MD)	3577	60	3.24	25
F2	fmuxA1	ctrl-MDcycle-passive* (MD*)	3122	60	1.19	26
F2	fmuxA2	ctrl-AS/MD (AS+MD)	3252	60	1.74	27
F2	fmuxA2	ctrl-AS/MD* (AS+MD*)	3187	60	0.38	28
F2	flatchOPA	cf-latch-opAB (phi)	2277	60	7.92	29
F2	flatchOPA	cf-latch-opAB* (phi*)	2799	60	6.35	30
F2	FBxor	cf-adder* (sub)	4481	60	10.10	31
F3	fintlatch	cf-latch-intermed (phi)	1973	60	7.59	32
F3	fintlatch	cf-latch-intermed* (phi*)	2588	60	6.17	33
F3	fintlatdr	cm-latch-compmcd (en*)	5291	60	5.22	34
F3	fcompl	frac-sign-intermed (Compl)	4559	60	1.59	36
F4	fmuxincout	ctrl-firstcycle-passive (SELA)	3083	60	5.97	37
F4	fmuxincout	ctrl-firstcycle-passive* (SELA*)	3083	60	4.28	38
F4	flatincout64	cf-latch-incout (phi)	3087	60	7.04	39
F4	flatincout64	cf-latch-incout* (phi*)	3355	60	5.55	40
F4	fincoutdr	cf-latch-lshin (en*)	5953	60	8.65	41

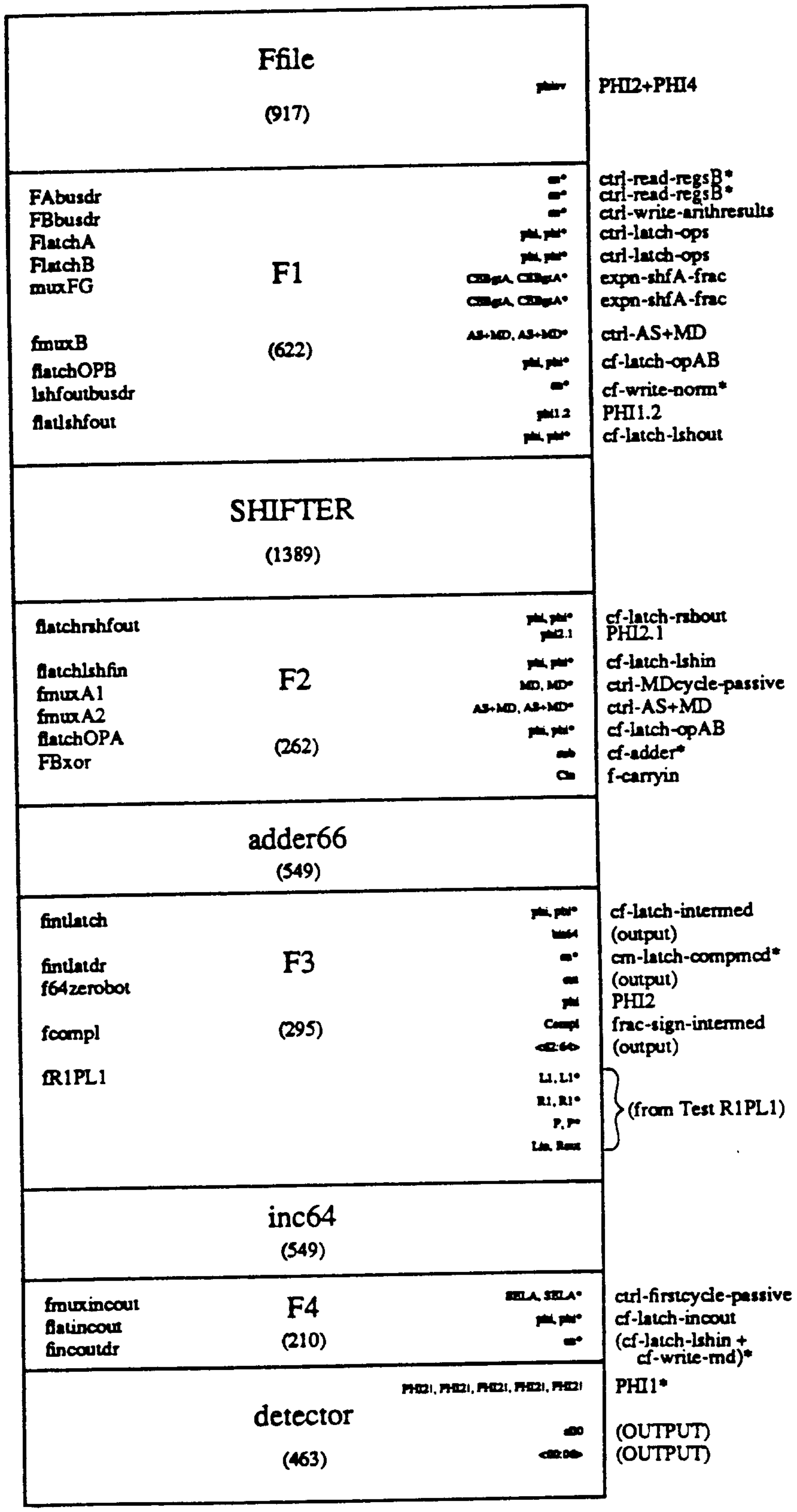
WORST CASE CONTROL INFORMATION FOR MULTIPLY/DIVIDE LAYOUT

BLOCK	SIGNAL (layout name)	Cap (fF)	Buffer	Delay (ns)	Label
IN	cm-latch-compmcd (phi)	2869	60	7.70	61,64
IN	cm-latch-compmcd* (phi*)	2988	60	6.36	62,63
IN	ctrl-latch-ops (phi)	2869	60	11.73	57,60,65,69,70
IN	ctrl-latch-ops* (phi*)	2988	60	10.41	58,59,66,68
IN	cm-clear-div (clr)	6438	60	35.01	67
MUX	cm-latch-slave (phi)	6334	60	24.23	39,44,49,54
MUX	Booth sel-1 (sel-1)	2930	60	3.93	38,43,48,53
MUX	Booth sel+1 (sel+1)	2856	60	3.87	40,45,50,55
MUX	Booth sel-2 (sel-2)	3001	60	3.98	41,46,51,56
MUX	Booth sel+2 (sel+2)	2930	60	3.93	37,42,47,52
LATCH	cm-latch-slave (phi)	3183	60	25.08	30,32,34,36
LATCH	cm-latch-slave* (phi*)	3895	60	28.84	29,31,33,35
CSA	cm-latch-master (phi)	11254	60	18.52	25,26,27,28
PP	cm-latch-master (phi)	3213	60	17.46	14,24
PP	cm-latch-master* (phi*)	3901	60	15.94	13,23
PP	cm-latch-slave (phi)	3213	60	23.62	8,16
PP	cm-latch-slave* (phi*)	3901	60	27.78	7,15
PP	cm-clear (clr)	7429	60	38.72	12,22
PP	ctrl-gate-PPS/C-fracBus (en)	7410	60	8.72	11,21
PP	ctrl-mulop (shr8)	3875	60	3.89	9,19
PP	ctrl-mulop* (shl2)	3943	60	1.53	10,20
PP	cm-mux-dvr (enbusA)	3936	60	8.24	17
PP	cm-mux-dvr* (enPPSm)	3888	60	7.44	18
QUO	cm-latch-master (MA)	3575	60	15.57	2,5
QUO	cm-latch-slave (SL)	2832	60	22.93	3,6
QUO	ctrl-gate-quotient-fracBus (en)	6500	60	8.07	1,4

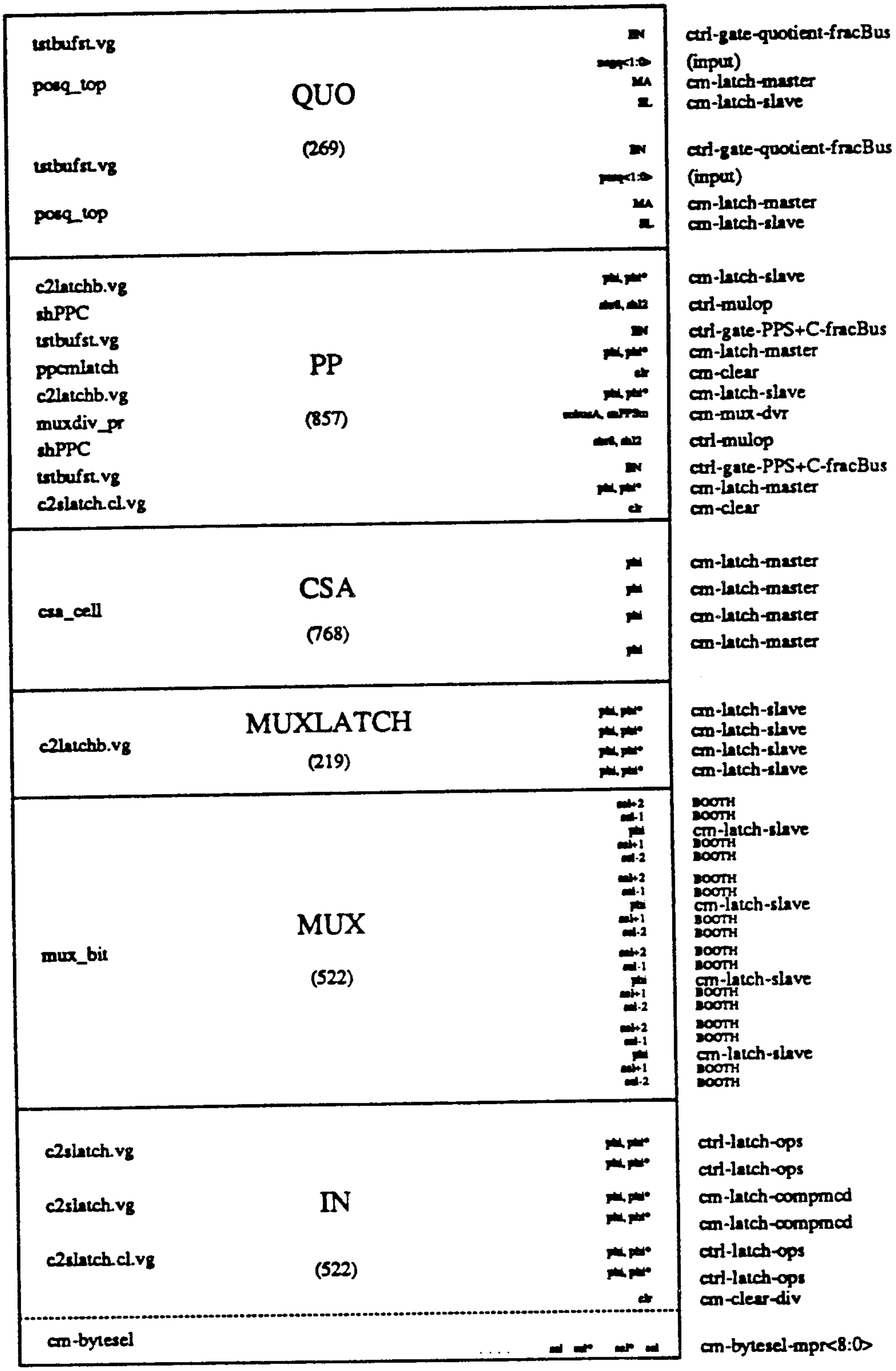
EXPONENT CONTROL SIGNAL PLACEMENT

Eregfile	Efile (860)	[illegible]	PHI2+PHI4
EAbusdr	E1	ar*	ctrl-read-regsB*
EBbusdr	(234)	ar*	ctrl-read-regsB*
ebusBinv		ar*	ctrl-write-arithresults*
AbarB17	AsubB17	phi1, phi1*	PHI1
	(388)	EB<16>, EA<16>	(output)
		EB-EA<16>	(output)
MuxBgtA17	E2	BgtA, BgtA*	expn-BgtA
Ege128	(100)	Ege128	(output)
		phi1	ce-latch-expndiff
	BsubA17	BA[illegible]<0:6>	(output)
AbarB	(366)	phi1, phi1*	PHI1
EAlatch		phi1, phi1*	ctrl-latch-ops
EBlatch		phi1, phi1*	ce-latch-expnB
MuxOPG	E3	EB>EA, EB>EA*	ce-mux-opG
latchOPG		phi1, phi1*	ce-latch-opGL
Eshiftcvt		shift-cvt<5:0>	(output)
muxOPL1	(368)	MD+AS, MD+AS*	ctrl-MD+AS
		frac-zerodetect<6:0>	(input)
muxOPL2		1cyc, 1cyc*	ctrl-firstcycle-passive
latchOPL		phi1, phi1*	ce-latch-opGL
SubXor17	EGsubEL (353)	CEadd	ce-adder*
		Cin	e-carryin
muxEdest1		arith, arith*	ctrl-arithop
mux17Edest2	E4	0det, 0det*	frac-zerodet
muxEdest3		1cyc, 1cyc*	ctrl-firstcycle-passive
Ereslatch	(217)	phi1, phi1*	ce-latch-dest
Eresbusdr		ar*	(ce-dest-to-busB + ctrl-write-arithresults)*

FRACTION CONTROL SIGNAL PLACEMENT



MULDIV CONTROL SIGNAL PLACEMENT



APPENDIX F: SIMULATION DATA

1. SPICE

1. Extract circuit description from within Magic: ":ext"
2. Obtain .sim file from ext2sim: "ext2sim -R -c 1e-18"
3. Remove any attributes from .sim file, such as the Cr:In$ labels used in CRYSTAL. In vi: "%s/d=Cr:In$//g", etc.
4. Obtain .spice file from sim2spice: "sim2spice -d ~/misc/def"
5. Add input and model cards to .spice file.
6. Submit .spice file on EROS: "spice < file.spice > file.out"

Typical ~/misc/def file:

```
def p P scmosII
def n N scmosII
set Vdd 1 scmosII
set GND 0 scmosII
set P 1
set N 0
```

Model parameters used:

```
*****************************************************************************
*        TYPICAL Device parameters for the HP CMOS40 Process
*
*                    Released 2/6/86 by Rich Duncombe
*       NOTE:  These parameters are intended for digital design only.
*
*****************************************************************************
*
* Use N and P models for W >= 4U and L <= 2U
*
/.MODEL N NMOS LEVEL=2 VTO= 0.75 KP=76.0U GAMMA=.40 LAMBDA=.025 TOX=25N
+ NSUB=4E16 TPG=+1 XJ=.25U LD=.20U UEXP=.16 VMAX=5.5E4 JS=1000U
+ CGSO=220P CGDO=220P CJ=230U CJSW=260P CGBO=400P
*
/.MODEL P PMOS LEVEL=2 VTO=-0.75 KP=27.0U GAMMA=.50 LAMBDA=.045 TOX=25N
+ NSUB=2.0E16 TPG=-1 XJ=.20U LD=.05U UEXP=.15 VMAX=9.0E4 JS=1000U
+ CGSO=220P CGDO=220P CJ=670U CJSW=215P CGBO=400P
*
*****************************************************************************
```

2. CRYSTAL

1. Extract circuit description from within Magic: ":ext"
2. Obtain .sim file from ext2sim "ext2sim -R -c 1e-18"
3. Start up CRYSTAL: "crystal"
4. Read in typical source file: "source file.crystal"

Typical source file (for exp-control.sim):

```
source crys_parm
build exp-control.sim
alias exp-control.al
inputs *PHI *cycle *clock *ctrl-start *ctrl-AS *ctrl-mulop
outputs <1:35>
watch 1 2
options graphics magic
options bus 12
options watchpaths 10
capacitance 1.6  1
capacitance 1.6  2
set 0 PHI3&
set 0 PHI2&
set 0 PHI4&
delay PHI1& 0 0
critical -g gatequo.mcrit
critical 1w
```

CRYSTAL parameters used:

```
! **********************************************************************
! crystal paramter release V2.1(2.10.86)
! based on HPCMOS40 1.6um Process
! extracted by Wook Koh        mail problems to wookkoh@kim
! **********************************************************************
tran nchan slopeparmsdown 0.000,8000,0.7;0.113,8500,0.7;0.271,9500,0.7;0.771, 11500, 0.7; 2.527, 15500, 1.0; 7.534, 2
tran nchan slopeparmsup 0.000, 8000, 0.7; 0.113, 8500, 0.7; 0.271, 9500, 0.7; 0.771, 11500, 0.7; 2.527, 15500, 1.0; 7.534
tran pchan slopeparmsdown 0.000, 20000, 0.8; 0.488, 25000, 0.9; 1.599, 35000, 1.0; 4.800, 55000, 1.5; 47.828, 194000, 5
tran pchan slopeparmsup 0.000, 20000, 0.8; 0.488, 25000, 0.9; 1.599, 35000, 1.0; 4.800, 55000, 1.5; 47.828, 194000. 5.0;
```

3. MOSSIM

1. Extract circuit description from within Magic: ":ext"
2. Obtain .sim file from ext2sim: "ext2sim -R -c 1e-18"
3. Obtain .ntk file from sim2ntk: "sim2ntk file"
4. Start up MOSSIM: "Mossim"
5. Read in .ntk description: "read file"
6. Read in typical source file: "source file"

sim2ntk file:

```
#!/bin/csh -f
if ($#argv != 1) then
echo "Usage:  sim2ntk file"
exit 1
endif
onintr end
set file=$1.sim
rm -r $1.ntk
echo sim $1.sim      >! temp.$$
echo type vdd:i gnd:i >> temp.$$
echo stren ratio:3.0  >> temp.$$
echo size ratio:3.0  >> temp.$$
echo write $1.ntk     >> temp.$$
echo quit            >> temp.$$
convert < temp.$$
end:
/bin/rm -f temp.$$
```

Typical source file (counter.src):

```
copy counter.cpy
comment ******************************************************
comment       Logical Simulation for 5-bit Counter
comment ******************************************************
comment
switch explain:1
clock phi1:10000000 phi1*:01111111 phi2:00100000 phi2*:11011111 -
phi3:00001000 phi3*:11110111 phi4:00000010
force Vdd:1 GND:0
vector clear* (clr&phi3)*
prefix 5bitinc_0
vector A incbit_3/ai incbit_2/ai incbit_1/ai incbit_0/ai incbit0_0/ai
unprefix
vector /b S s4 s3 s2 s1 s0
vector /b SPHI1 s4_phi1 s3_phi1 s2_phi1 s1_phi1 s0_phi1
watch /* /b 5bitinc_0/A S SPHI1 clear* start busy inc cout
set /b 5bitinc_0/A:00000
set /b clear*:0 start:0 busy:0
```