The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLSI Multiprocessor

Richard R. Duncombe

Department of Electrical Engineering
and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

Microprocessor architecture is evolving rapidly as silicon integrated circuits increase in density. On-chip cache memories are becoming an established feature in 32-bit microprocessor designs because they significantly improve performance. Microprocessor performance is degraded by bus contention between instruction and data memory traffic. On-chip instruction caches reduce this contention problem by supplying many of the instructions executed by the microprocessor. The SPUR instruction unit is a direct mapped cache with 512 bytes or 128 instructions. It is organized in sub-blocks to provide efficient instruction fetching and prefetching from the external memory. The SPUR instruction unit is controlled by two finite state machines: one for instruction fetching and one for instruction prefetching. These control functions are implemented using PLA's and standard logic cells. The standard cells are implemented in domino logic to meet speed and area constraints. SPICE simulations indicate that the slowest signal delay path in the instruction unit is 14.7 ns. The SPUR instruction unit contains 39,400 transistors and occupies 4200 x 6000 μm^2 in a 2 μm technology. Area and speed metrics for alternative instruction units indicate that implementations with either larger sub-blocks or two-way associativity will satisfy the SPUR CPU speed requirements. A two-way set-associative implementation would consume approximately 20% more silicon area.

August 8, 1986

TABLE OF CONTENTS

1. INTRODUCTION

Microprocessor architecture is evolving rapidly to incorporate advances in integrated circuit technology. The most dramatic progress in technology has been achieved in the density of transistors on a single chip. Computer architects must evaluate new alternatives to determine the optimum way to use this increased density.

1.1 On-Chip Cache Memory Research

Research indicated that on-chip cache memories could provide significant performance benefits before technology would support their implementation. Smith commented that more research was necessary to best design on-chip caches [Smith82]. Goodman proposed that small local cache memories were essential for single chip or single board central processor units (CPUs) [Good83]. He argued that the bandwidth between the CPU and memory system would limit performance unless local cache memory was provided. Furthermore, the local cache must not further burden the bus resources. Goodman concluded that the amount of data transferred between the local cache and external memory system should be reduced in designs limited by bus bandwidth. Trace driven simulations by Hill demonstrated benefits of on-chip cache memories in several different computer architectures [Hill84]. Hill also presented data to support 'sub-block' cache memory organizations. Sub-blocks refer to the amount of data that is transferred between the local cache and the external memory system. Two or more sub-blocks and an address tag are contained in each block in the cache. Hill demonstrated that the use of sub-blocks in on-chip cache design reduces bus traffic. However, sub-blocks also reduce the hit ratio of the cache.

1.2 On-Chip Cache Memory Implementations

A number of recent microprocessor designs have implemented on-chip cache memories. Two examples are reviewed in this section. The Motorola MC68020 32-bit microprocessor contains a 256-byte instruction cache [Mac84]. The MC68020 instruction cache improves performance by reducing the number of accesses that must be made to the external memory system. The

MC68020 instruction cache only supports basic features. It is a direct mapped cache and does not support prefetching from memory. However, even this basic design has a significant impact on performance. System measurements indicate that 64% of the instructions executed by the MC68020 are contained in the instruction cache [Mac85]. Furthermore, the overall system performance improves by 30% when the instruction cache is enabled.

The Zilog Z80000 microprocessor contains a 256-byte cache memory that supports more complex features [Phil85]. For example, this cache may be configured in one of three ways: instruction cache, data cache, or a unified cache (instructions and data). It also supports fully associative mapping between the instructions address and cache blocks. Trace driven simulations of this cache indicate that the hit ratio should be 60% to 70%. Overall performance of the microprocessor should improve by 20% to 30% [Phil85].

1.3 MS Report Outline

This MS report describes an on-chip instruction cache designed for the SPUR project at U.C. Berkeley. The objective of the SPUR (Symbolic Processing Using RISCs) project is to design and build a multiprocessor workstation [Hill85]. A custom 32-bit RISC CPU, cache controller, and floating-point processor are being developed to support the multiprocessing Lisp environment. The instruction cache on the SPUR CPU is referred to as 'the instruction unit' to avoid confusion with the external cache memory in the SPUR system.

The information in this MS report is organized from high-level architectural descriptions to low-level circuit implementation details. Chapter 2 first provides a brief review of the SPUR CPU microarchitecture. The architecture of the instruction unit and its interface to the CPU are then described. The final section in Chapter 2 presents five examples to clarify the operation of the instruction unit. Chapter 3 focuses on the implementation of the SPUR instruction unit. State diagrams illustrate the two finite state machines that exist within the instruction unit. The next section describes the implementation philosophy for the instruction unit control. The final three sections of Chapter 3 discuss the circuit implementations of the control, instruction buffer, tag

comparison, and prefetcher functional blocks. Finally, Chapter 4 examines several implementation parameters including: memory array size, sub-block size, and associativity.

2. THE SPUR INSTRUCTION UNIT ARCHITECTURE

The SPUR instruction unit architecture has been influenced by previous research at U.C. Berkeley. For example, a VLSI cache chip was designed and implemented as a U.C. Berkeley graduate class project [Hill82], [Patt83]. Hill studied trace driven simulations of cache memories suitable for on-chip implementations [Hill84]. The instruction unit architecture was further refined when the functional description of the CPU was written [Kong86a], [Kong86b].

I was not at U.C. Berkeley to participate in the evolution of this architecture. Therefore, I will not address many of the issues and ideas that were considered during its development. However, I describe the final instruction unit architecture in this chapter. In the next chapter I will combine the implementation issues that I faced with this architectural information to provide a cohesive description of the project.

2.1 Overview of the SPUR CPU

This section provides an overview of the SPUR CPU microarchitecture. This information is needed to support the subsequent sections that examine the instruction unit architecture and implementation. The CPU microarchitecture will be documented in a dissertation by Shing Kong. The implementation issues for the execution unit will be written by Dave Lee and Wook Koh. The clock generation circuits and PLA templates will be documented by Deog-Kyoon Jeong.

CPU Functional Description: At the highest level, the SPUR CPU contains two functional units: the execution unit (EUnit) and the instruction unit (IUnit). These two functional units contain separate control blocks as in a traditional board-level CPU implementation. The SPUR CPU, however, takes advantage of the EUnit and IUnit integration by sharing four wide busses. The pin count for these busses alone (102 pins) uses most of the available SPUR CPU signal pins. The instruction unit interface is presented in section 2.3.

The block diagram in Figure 2-1 shows the key functional blocks of the CPU. The EUnit data path is naturally divided into two parts. The lower datapath performs most of the register to register instructions and initiates branch and trap activities. The upper datapath maintains

the program counter and register window pointers. It also calculates the destination address for all compare and branch instructions. The following paragraphs describe each block in Figure 2-1 from left to right and bottom to top [Kong86a].

Reg_File: A 138 word by 40 bit register file which supports dual port reads and single port writes. This register file is organized as 8 overlapping register windows with 32 registers in each window.

Pipe_Reg: This block contains three datapath latches. Each latch is 40-bits wide. Two latches store data read and written from the register file to make it available during the proper pipeline stage. The other latch stores data that will be transferred to the Data_bus. The information on the Data_bus is written to the external cache memory.

In_Ext: This block inserts and extracts bytes for tag manipulation.

Shift: This is a 3-bit shifter for the ALU.

ALU: Performs A+B, A-B, A XOR B, A AND B, and A OR B.

UPSW_KPSW: Special Registers for the User Process Status Word and the Kernal Process Status Word.

Br_Cond: Uses ALU information to evaluate the conditions for the compare and branch instructions.

Trap: Generates trap requests for unusual conditions.

IUnit: The IUnit receives instruction requests on the program counter bus (PC_bus) from the EUnit. The IUnit sends instructions on the instruction bus (Ins_bus) to the Mst_Ctr block. The IUnit also initiates fetches and prefetches on the Add_bus and loads instructions from the Data_bus.

Mst_Ctr: The master control block decodes the instruction OP codes into high-level control signals for the rest of the CPU. High-level control signals from the Mst_Ctr do not directly connect to the data path circuits. These signals

provide communication between the Mst_Ctr block and other local control blocks within the CPU. Since the CPU supports a four stage pipeline, the master control must deliver each signal for an instruction during the correct cycle. Further decoding and control is performed locally in many of the other blocks. This will be clarified further in chapter 3 when the IUnit control is described.

Cache_Intf: This block communicates with the external cache controller chip.

CWP_SWP: Special registers for the Current Window Pointer and the Saved Window Pointer in the register file.

Special_PC: Three special program counters are maintained in this block: TrapPC, CallPC, and KernalPC.

PC_Logic: This block maintains the special program counters and the pipeline program counters. It contains a 30-bit adder to calculate the destination address for all the compare and branch instructions.

Pipeline_PC: The program counter for each instruction in the execution pipeline is stored in this block.

Execution Unit Pipeline: The pipeline design in the EUnit influenced almost all other parts of the CPU microarchitecture. A four stage pipeline was selected to ideally allow one instruction to execute every cycle. Instruction unit misses and external cache misses prevent the ideal case from being achieved [Hill85]. A four phase non-overlapping clocking scheme was selected to support the pipeline. The pipeline stages are listed below [Kong86a]:

Ins_Fet: Instruction is fetched from the IUnit to the EUnit.

Execution: The instruction is executed. Instructions that access memory calculate the effective address during this cycle.

Memory: All instructions that access memory do so in this cycle. This is a null cycle for other instructions.

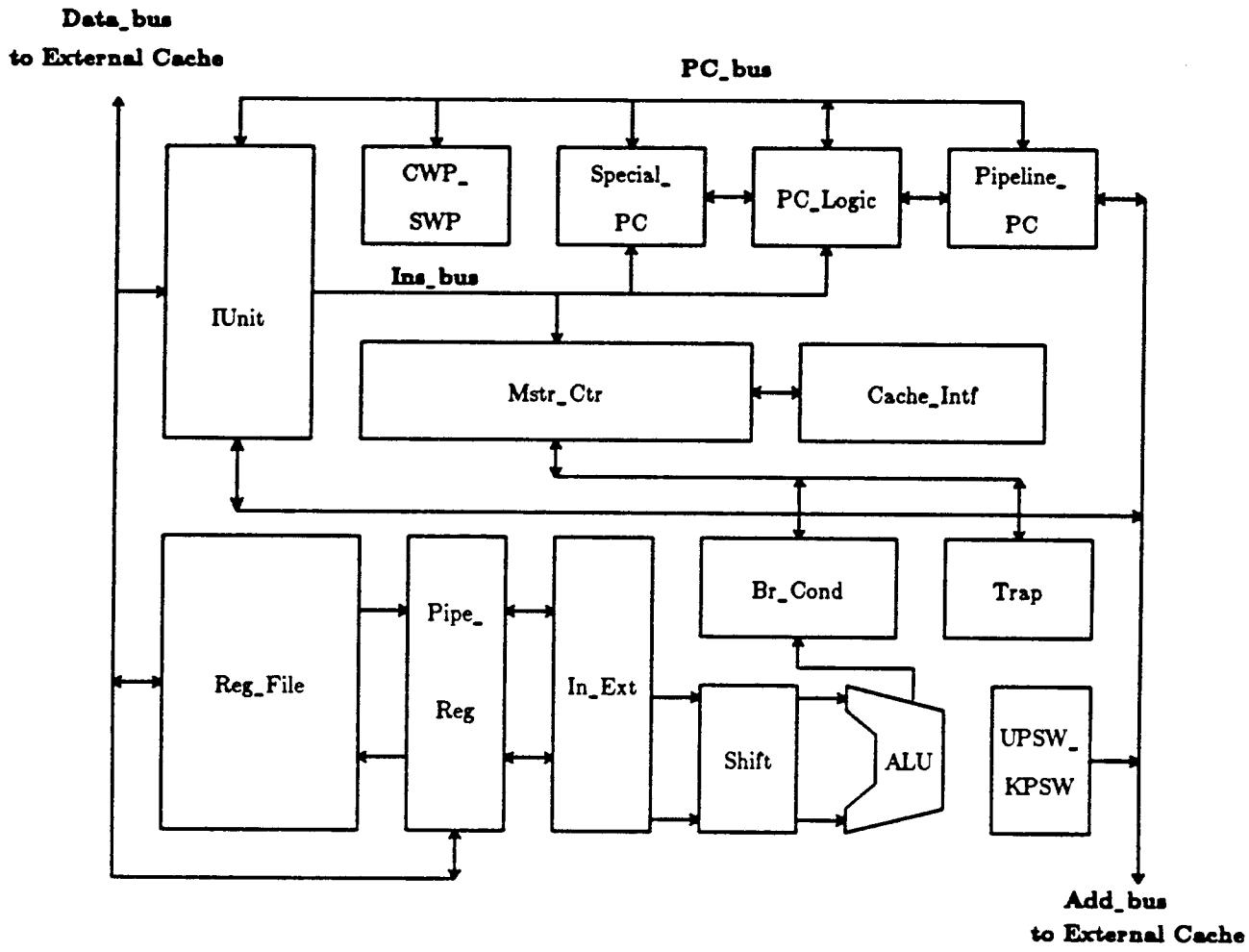


Figure 2-1: CPU Block Diagram

Reg_Write: The register file is written into during this cycle.

Pipeline Suspension: Two suspension mechanisms are provided in the SPUR EUnit pipeline: global and partial [Kong86a]. The IUnit responds differently for each case. In the former, everything in the pipeline is 'frozen'. The IUnit continues to send out the last instruction repeatedly until the suspension ends. Partial suspension allows the instructions already in the pipeline to complete execution. The IUnit feeds an internally generated 'MISS' instruction until the suspension ends.

Trap Request Handling: The trap request handling mechanism in the EUnit microarchitecture is also coordinated with the IUnit. When a trap occurs, the instructions in the first three

pipeline stages are 'killed' [Kong86a]. The IUnit then issues two internally generated instructions to the EUnit. First, a TRAP_CALL instruction initiates execution of the trap handler software. The next instruction executed, READ_PC, is also issued by the IUnit. This internally generated instruction saves the program counter for the 'killed' instruction. The trap handler software begins to execute after the READ_PC instruction.

2.2 SPUR Instruction Unit Architecture

The primary motivation for an on-chip instruction cache is to reduce contention for the external cache [Hill85]. If the CPU was dependent on the external cache for both instructions and data references, the pipeline would have to stall on every data reference. Other options such as two cache ports on the CPU or using an interleaved external cache were rejected because of the increased hardware cost [Hill86a]. The on-chip instruction unit relieves this external cache bottleneck by providing the illusion of a parallel memory port for the execution unit. Therefore, the execution unit may get an instruction and a data reference in the same cycle as long as there is a hit in the IUnit. A second benefit is that the latency time for instructions is reduced for the CPU. In the SPUR implementation, the IUnit has a latency time of approximately one half cycle. The external cache has a one cycle latency time. The CPU control takes advantage of this reduced latency. Control signals sent to the IUnit from the CPU Mst_Ctr block arrive approximately one half cycle later than signals sent to the external cache.

Instruction Unit Organization: The IUnit is a 512-byte, direct mapped cache. It is organized into 16 blocks with each block containing 32 bytes. The cache contains 128 instructions (4-bytes per instruction). Therefore, each block contains 8 instructions which are referred to as 'sub-blocks'. Several advantages of using sub-blocks on small on-chip caches memories have been reported [Good83], [Hill84]. Perhaps the most significant advantage is that the memory traffic is reduced because only required sub-blocks are loaded. This is important for the SPUR IUnit because instruction memory traffic coinciding with data traffic forces the EUnit pipeline to stall. Unfortunately, sub-blocks also cause the miss ratio to increase because several misses may occur

in the same block [Hill84].

Instruction Virtual Address Fields: The IUnit divides the instruction address into three address fields. The implementation of the instruction memory array and tag comparison logic is based on the following address fields:

1. Sub-Block Address Field <3 bits> - selects one of 8 sub-blocks
2. Block Address Field <4 bits> - selects one of 16 blocks
3. Address Tag Field <23 bits> - highest order instruction address bits that identify the present address tag for each block.

The advantages of using on-chip instruction cache memories have been demonstrated in both simulations and system measurements [Good83], [Hill84], [Mac85]. Implementing an on-chip cache introduces several system problems that must be overcome. For example, self modifying programs may alter the instruction stream stored in the main memory. The SPUR architecture avoids this problem by requiring that software invalidates the IUnit before executing a modified instruction stream. A similar problem is encountered when software changes the mapping of process-specific addresses to global addresses [Hill86b]. This typically occurs as a result of a context switch. The SPUR architecture solves this problem by requiring software to invalidate the IUnit whenever this mapping is changed.

The CPU design is simplified because the SPUR architecture forbids modifications in the virtual address space without software intervention Without this restriction, the EUnit pipeline and the IUnit would have to check for self modifying code [Hill86b]. In other words, the EUnit would have to determine whether instructions already in the pipeline had been modified and the IUnit would have to determine if it had 'stale' instructions cached.

Prefetching on Misses: The sub-block organization in the IUnit supports a prefetching algorithm that is initiated by misses. When a Miss occurs, the EUnit is passed a 'MISS' instruction from the IUnit and a memory fetch is initiated. The external cache returns one instruction or sub-block so that the EUnit may resume operation. The IUnit will then try to prefetch the

remainder of the block in sequential order. In the ideal case, the prefetcher will prevent any additional misses within the same block. The conditions that must be satisfied in the ideal case are: (1) the EUnit is executing sequential code in the same block, and (2) the EUnit must make no data references to the external cache. The prefetcher does not burden the CPU to external cache interface because it is overridden by either an instruction miss or a data reference. More detailed sequences of operation are described in section 2.4.

The 'prefetch-on-miss' algorithm and sub-block architecture implemented in the SPUR IUnit have several notable advantages and disadvantages. The primary advantage of only fetching a sub-block is that the instruction miss can be satisfied more quickly by the external cache [Hill85]. This assumes that the CPU package has a limited number of pins available and that a block is always larger than a single instruction. The first assumption was confirmed in the SPUR CPU implementation as the limited package pins had to be carefully allocated. Alternative data bus widths between the IUnit and external cache are discussed in chapter 4. The second assumption is also safe when considering the overhead for the address tag implementation. This overhead is reduced by a factor of n when a group of n sub-blocks are used [Hill84].

The disadvantages of supporting sub-blocks are increased silicon area and decreased speed. Each sub-block must have a valid bit in addition to its block valid bit. I estimate that 4% of the total IUnit area was used by the sub-block valid bits and associated circuitry. Speed is reduced because the sub-block valid bit must be read from the instruction array and sent to the IUnit control block. This is the critical signal delay path in the entire IUnit implementation (14.7 ns typical). The critical signal delay path would decrease by 2.2 ns if the sub-block valid bit was not necessary. Simulation results are presented in section 3.4.

The SPUR IUnit prefetching algorithm continuously prefetches from one demand miss to the next demand miss. When the prefetcher reaches the end of a block, it continues prefetching at the beginning of the same block. At first glance, this algorithm could be viewed as wasteful since the prefetcher may actually prefetch sub-blocks that are already valid (that it previously fetched or prefetched). However, the prefetcher is always overridden when there is real work to do like

instruction misses or data references, and the implementation is simplified because only a 3-bit incrementer is required.

2.3 Instruction Unit Interface

The instruction unit interface serves several purposes. It coordinates the two finite state machines in the IUnit, fetch and prefetch, with the CPU master control finite state machine. The interface also allows the IUnit to monitor the external cache controller chip. Finally, the interface provides four wide busses for instruction address and data movement. Previous work at U.C. Berkeley has helped refine this interface. A single-chip instruction cache explored CPU to cache interfaces so that on-chip caches could be effectively integrated when technology would permit [Patt83],[Hill84].

The control signals and busses that make up the IUnit interface are listed in Table 2-1. All signals or busses in the interface pass through the EUnit. However, the Cache_Busy and Cache_Data_Valid signals are initiated by the external cache (ECache). To avoid confusion, I will always describe the state of signals as either true or false. If a signal is true, then the action or condition indicated by that signals name is presently valid. For example, if I state that Cache_Busy is true, this indicates that the external cache is presently busy.

2.4 Instruction Unit Operation Examples

This section will illustrate the architectural information presented so far with some examples. These examples emphasize the IUnits operation. Therefore, simplifications are often made about the EUnit or the external cache. The internal activity within the IUnit is also simplified in order to avoid unnecessary detail. The internal operation is further discussed in Chapter 3.

Each of the following five examples include a verbal description and a table. The table summarizes the IUnit operation during a sequence of CPU cycles. The IUnit inputs and outputs refer to the signals and busses presented in the previous section. Internal actions taken by the IUnit

Signal or Bus Name	Functional Description
Execution Unit to Instruction Unit Signals	
Reset_IUnit	Resets the IUnit from any previous state
IUnit_KPSW_Set	Single Bit in the KPSW sets the enabled condition
Prefetch_KPSW_Set	Single Bit in the KPSW sets the prefetch condition
Pipeline_Not_Suspended	Indicates no Global Pipeline Suspension in EUnit
Load_OPCode	Load Instruction is being executed by the CPU
Store_OPCode	Store Instruction is being executed by the CPU
lowTOup_OPCode	Lower and upper data paths are using the Add_bus
Invalidate_OPCode	Invalidate IUnit Instruction is being executed by the CPU
Invalidate_Trap	EUnit is servicing a Trap that invalidates the IUnit
External Cache to Instruction Unit Signals	
Cache_Busy	Indicates that the ECache is busy
Cache_Data_Valid	Indicates that the ECache data is valid on the Data_bus
Instruction Unit to Execution Unit Signals	
Fetch_Request	Indicates that the IUnit is fetching an instruction
Prefetch_Request	Indicates that the IUnit is prefetching an instruction
Instruction Unit / Execution Unit Busses	
PC_bus <30 bits>	Transfers program counter from EUnit to IUnit
Add_bus <30 bits>	Connects IUnit and EUnit to the ECache address pads
Ins_bus <32 bits>	Transfers instructions from the IUnit to the EUnit
Data_bus <40 bits>	Connects IUnit and EUnit to the ECache data pads

Table 2-1: Instruction Unit Interface

are also indicated in the table. For clarity, all the sequence examples assume that the IUnit starts in the same state: FET_normal and PF_prefetch. The IUnit is in the FET_normal state if the last instruction requested by the EUnit was a hit. The PF_prefetch state indicates that the IUnit is actively trying to prefetch sequential instructions. These instruction unit states are further described in section 3.1.

IUnit Operation Example - Ideal Instruction Miss: An ideal instruction miss refers to the shortest possible sequence to fetch a missed instruction. This corresponds to a two cycle delay in the SPUR CPU. In order for this ideal sequence to occur, the external cache must not be busy. This operation example is described in full to clarify some of the cryptic notations used in Table 2-2.

Cycle 1: During phi1 of cycle 1, the EUnit requests an instruction from the IUnit. The program counter (ins_add[i]) is the address of the next instruction needed by the EUnit. The IUnit detects that it does not contain this instruction (Miss) during phi2. The IUnit then initiates a partial pipeline suspension in the EUnit by sending an internally generated MISS instruction (ins[MISS]) on the Ins_bus. During phi4, the instruction unit fetches the missed instruction with address (ins_add[i]) from the external cache.

Cycle 2: The IUnit waits during phi1 and phi2 for the Data_Valid signal from the external cache. A second internal MISS instruction is sent out during phi3. In phi4 the IUnit accepts the fetched instruction (ins[i]) from the external cache and writes it into its memory array. It also initiates a prefetch for the next sequential instruction (ins_add[i]+1).

Cycle 3: This cycle demonstrates a Hit in the IUnit. The EUnit again supplies the address of the program counter (ins_add[i]) to the IUnit. Since this instruction was just fetched from memory, the IUnit detects a Hit in phi2. During phi3, the requested instruction is sent to the EUnit thereby ending the partial pipeline suspension. Finally, in phi4, the IUnit accepts the prefetched instruction (ins[i]+1) from the external cache and writes it into its memory array. It again initiates a prefetch for the next sequential instruction (ins_add[i]+2) from the ECache.

Instruction Unit Inputs		Instruction Unit Outputs
Cycle 1		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i] on the PC_bus	phi1	
	phi2	IUnit detects a Miss for ins_add[i]
ECache is not Busy	phi3	IUnit sends ins[MISS] to EUnit on the Ins_bus
	phi4	IUnit sends ins_add[i] to ECache on the Add_bus

Cycle 2		state(FET_memPending, PF_waiting)
IUnit ignores the PC_bus	phi1	
	phi2	IUnit waiting for ECache Data
ECache data is valid	phi3	IUnit sends ins[MISS] to EUnit on the Ins_bus
IUnit accepts ins[i] from ECache on the Data_bus	phi4	IUnit sends ins_add[i]+1 to ECache on the Add_bus

Cycle 3		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i] on the PC_bus	phi1	
	phi2	IUnit detects a Hit for ins_add[i]
ECache data is valid	phi3	IUnit sends ins[i] to EUnit on the Ins_bus
IUnit accepts ins[i]+1 from ECache on the Data_bus	phi4	IUnit sends ins_add[i]+2 to ECache on the Add_bus

Table 2-2: IUnit Operation Example - Ideal Instruction Miss

IUnit Operation Example - Invalidate: The EUnit can invalidate the IUnit either by executing an invalidate instruction or servicing a trap that causes invalidation. The IUnit overwrites all block valid tags in phi2 of the cycle that receives the invalidate request. Table 2-3 shows that the instruction unit recovers quickly from an invalidate request. In the ideal case, an invalidate request only suspends the pipeline for two cycles. This is because the IUnit may start a fetch in phi4 of the cycle that receives the invalidate request.

Instruction Unit Inputs		Instruction Unit Outputs
Cycle 1		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i] on the PC_bus EUnit Invalidates IUnit	phi1	
	phi2	All block valid tags are invalidated
ECache is not Busy	phi3	IUnit sends ins[MISS] to EUnit on the Ins_bus
IUnit ignores Data_bus	phi4	IUnit sends ins_add[i] to ECache on the Add_bus
Cycle 2		state(FET_memPending, PF_waiting)
IUnit ignores the PC_bus	phi1	
	phi2	IUnit waiting for ECache Data
ECache data is valid	phi3	IUnit sends ins[MISS] to EUnit on the Ins_bus
IUnit accepts ins[i] from ECache on the Data_bus	phi4	IUnit sends ins_add[i]+1 to ECache on the Add_bus
Cycle 3		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i] on the PC_bus	phi1	
	phi2	IUnit detects a Hit for ins_add[i]
ECache data is valid	phi3	IUnit sends ins[i] to EUnit on the Ins_bus
IUnit accepts ins[i]+1 from ECache on the Data_bus	phi4	IUnit sends ins_add[i]+2 to ECache on the Add_bus

Table 2-3: IUnit Operation Example - Invalidate

IUnit Operation Example - Reset Followed by a Hit: The EUnit can put the IUnit into a known state from any previous state by asserting the Reset signal. The IUnit starts a sequence of actions as soon as Reset is true. The reset mechanism in the instruction unit is used to support several system level activities. For example, traps, page faults, and system power-up sequences use the IUnit reset feature.

The operation example, shown in Table 2-4, illustrates several other features supported by the IUnit during resets. First, a reset request from the EUnit does not invalidate the instructions cached in the IUnit. It does force the IUnit to ignore any pending fetches or prefetches. Secondly, this example demonstrates the sequence of internally generated instructions sent from the IUnit to the EUnit. During phi2 of cycle 1, the EUnit requests the IUnit to reset. In phi3 of the same

cycle, the IUnit sends a TRAP_CALL instruction to the EUnit. The READ_PC instruction is sent to the EUnit during phi3 of cycle n. The EUnit discontinued the reset on phi2 of this cycle. The final IUnit feature that this example illustrates is the prefetch-on-miss algorithm. Cycle n+1 was chosen to be an instruction Hit even though this is unlikely. The prefetcher remains idle until a miss, invalidation, or global suspension occurs. In most typical sequences, however, a miss will occur after a reset. Therefore, the prefetcher will resume its prefetching activities.

Instruction Unit Inputs		**Instruction Unit Outputs**
Cycle 1		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i] on the PC_bus	phi1	
EUnit Resets IUnit	phi2	
	phi3	IUnit sends ins[TRAP_CALL] to EUnit on the Ins_bus
IUnit ignores Data_bus	phi4	
Cycle n		state(FET_reset, PF_reset)
IUnit ignores the PC_bus	phi1	
EUnit discontinues IUnit Reset	phi2	
	phi3	IUnit sends ins[READ_PC] to EUnit on the Ins_bus
IUnit ignores Data_bus	phi4	
Cycle n+1		state(FET_normal, PF_idle)
EUnit requests ins_add[i] on the PC_bus	phi1	
	phi2	IUnit detects a Hit for ins_add[i]
	phi3	IUnit sends ins[i] to EUnit on the Ins_bus
IUnit ignores Data_bus	phi4	Prefetcher idle until a Miss, Invalidate or Global Suspension occurs

Table 2-4: IUnit Operation Example - Reset Followed by a Hit

IUnit Operation Example - Trap: The trap sequence, shown in Table 2-5 is quite similar to the the preceding reset example. The EUnit uses the reset mechanism to initiate the sequence of internally generated instructions: TRAP_CALL and READ_PC. In this example, the IUnit detects a miss after the READ_PC instruction. This is a more realistic case since it is unlikely that the trap handler instructions would be cached by the IUnit.

Instruction Unit Inputs		Instruction Unit Outputs
Cycle 1		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i] on the PC_bus	phi1	
EUnit Resets IUnit	phi2	
	phi3	IUnit sends ins[TRAP_CALL] to EUnit on the Ins_bus
IUnit ignores Data_bus	phi4	

Cycle 2		state(FET_reset, PF_reset)
IUnit ignores the PC_bus	phi1	
EUnit discontinues IUnit Reset	phi2	
	phi3	IUnit send ins[READ_PC] to EUnit on the Ins_bus
IUnit ignores Data_bus	phi4	

Cycle 3		state(FET_normal, PF_idle)
EUnit requests ins_add[i] on the PC_bus	phi1	
	phi2	IUnit detects a Miss for ins_add[i]
ECache is not Busy	phi3	IUnit sends ins[MISS] to EUnit on the Ins_bus
IUnit ignores Data_bus	phi4	IUnit sends ins_add[i] to ECache on the Add_bus

Cycles 4 & 5	Same as Cycles 2 & 3 in Ideal Miss (Table 2-2)

Table 2-5: IUnit Operation Example - Trap

IUnit Operation Example - Global Pipeline Suspension The final IUnit operation example is shown in Table 2-6. This sequence illustrates the interaction between the EUnit and the IUnit during a global pipeline suspension. Cycle 1, which is an IUnit Hit, sends the requested instruction (ins[i]) to the EUnit on phi3. The IUnit latches this instruction internally on phi2. During phi1 of cycle 2, the EUnit signals a global pipeline suspension. The IUnit responds by repeatedly sending out the last instruction (ins[i]) to the EUnit on phi3. The prefetcher could remain active during a global pipeline suspension as shown in cycle 2. This is unlikely, however, because any other request for the external cache disables the prefetcher. In cycle n, after the pipeline suspension has ended, the IUnit resumes normal operation.

Instruction Unit Inputs		Instruction Unit Outputs
Cycle 1		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i] on the PC_bus	phi1	
	phi2	IUnit detects a Hit for ins_add[i]
ECache is not Busy	phi3	IUnit sends ins[i] to EUnit on the Ins_bus
	phi4	IUnit sends ins_add[i]+1 to ECache on the Data_bus

Cycle 2		state(FET_normal, PF_prefetch)
EUnit signals Global Pipeline Suspension	phi1	
	phi2	IUnit does not read a new instruction
ECache data is valid	phi3	IUnit sends ins[i] to EUnit on the Ins_bus
IUnit accepts ins[i]+1 from ECache on the Data_bus	phi4	IUnit sends ins_add[i]+2 to ECache on the Add_bus

Cycle n		state(FET_normal, PF_prefetch)
EUnit requests ins_add[i]+1 on the PC_bus EUnit ends Global Pipeline Suspension	phi1	
	phi2	IUnit detects a Hit for ins_add[i]+1
ECache Data is valid	phi3	IUnit sends ins[i]+1 to EUnit on the Ins_bus
IUnit accepts ins[i]+2 from ECache on the Data_bus	phi4	IUnit sends ins_add[i]+3 to ECache on the Add_bus

Table 2-6: IUnit Operation Example - Global Pipeline Suspension

3. THE SPUR INSTRUCTION UNIT IMPLEMENTATION

The architectural decisions described in the previous chapters take on different meanings when the implementation issues are also considered. The purpose of this chapter is to highlight the implementation of the SPUR CPU instruction unit. It serves as a benchmark for future instruction cache implementations. A layout plot of the IUnit is shown in Figure A-1 in Appendix A. The information in this chapter is arranged in a hierarchical fashion for the convenience of both casual readers and those hungry for details. First, a state machine description of the IUnit is provided to clarify its operation. The IUnit is then dissected into four parts: control, instruction buffer, tag comparison, and prefetcher.

3.1 Instruction Unit State Description

Two finite state machines exist within the IUnit: the fetch finite state machine (Fetch_FSM) and the prefetch finite state machine (Prefetch_FSM). The input signals to these two finite state machines are shown in Table 3-1. The first seven input signals (Reset > Flush) are simple derivatives of the IUnit interface signals presented in section 2.3. The Miss signal is derived from two signals internal to the IUnit: Block_Miss and Instruction_Miss. These signals are discussed further in sections 3.4 and 3.5. The final signal, Starting_Prefetch, is the only connection between the Fetch_FSM and the Prefetch_FSM. Starting_Prefetch is used to implement the prefetch-on-miss algorithm. The Fetch_FSM tells the Prefetch_FSM that it can begin prefetching after a Miss, Flush, or a Global_Suspension are true.

In both the Fetch_FSM and Prefetch_FSM, the next state is determined by the input signals in Table 3-1 and the previous state. A programmed logic array (PLA) determines the next state during phi4. A new state officially begins on every phi1. Two sets of dynamic latches are used in conjunction with the PLA's. The first set of latches captures the next state on phi4. The second set of latches, which hold the present state, are updated on phi1. The PLA's for the Fetch_FSM and the Prefetch_FSM are shown in Figures A-2 and A-3 respectively in Appendix A. The finite state machine descriptions for both PLA's are included in Appendix B.

Signal Name	Origin of Signal
Reset	Same as Reset_IUnit signal
IUnit_Enable	Latched version of IUnit_KPSW_Set signal
Prefetch_Enable	Latched version of Prefetch_KPSW_Set signal
Global_Suspension	Latched version of Pipeline_Not_Suspended
Memory_Busy	Cache_Busy AND NOT(Cache_Data_Valid) OR (Load_OPCode OR Store_OPCode OR lowTOup_OPCode)
Data_Valid	Latched version of Cache_Data_Valid signal
Flush	Invalidate_OPCode OR Invalidate_Trap
Miss	Block_Miss OR Instruction_Miss
Starting_Prefetch	Signal from the Fetch_FSM to the Prefetch_FSM

Table 3-1: Instruction Unit Control Input Signals

Fetch Finite State Machine: The Fetch_FSM controls most of the activities in the instruction unit. The state of the Fetch_FSM is combined with other inputs to determine whether the IUnit transfers instructions to the EUnit (Ins_bus), fetches instructions from the external cache (Add_bus), or accepts data from the external cache (Data_bus). The Fetch_FSM is controlled by seven of the input signals in Table 3-1: Reset, IUnit_Enable, Global_Suspension, Memory_Busy, Data_Valid, Flush, and Miss. The Fetch_FSM consists of five states as shown in Figure 3-1. Each state is briefly described below.

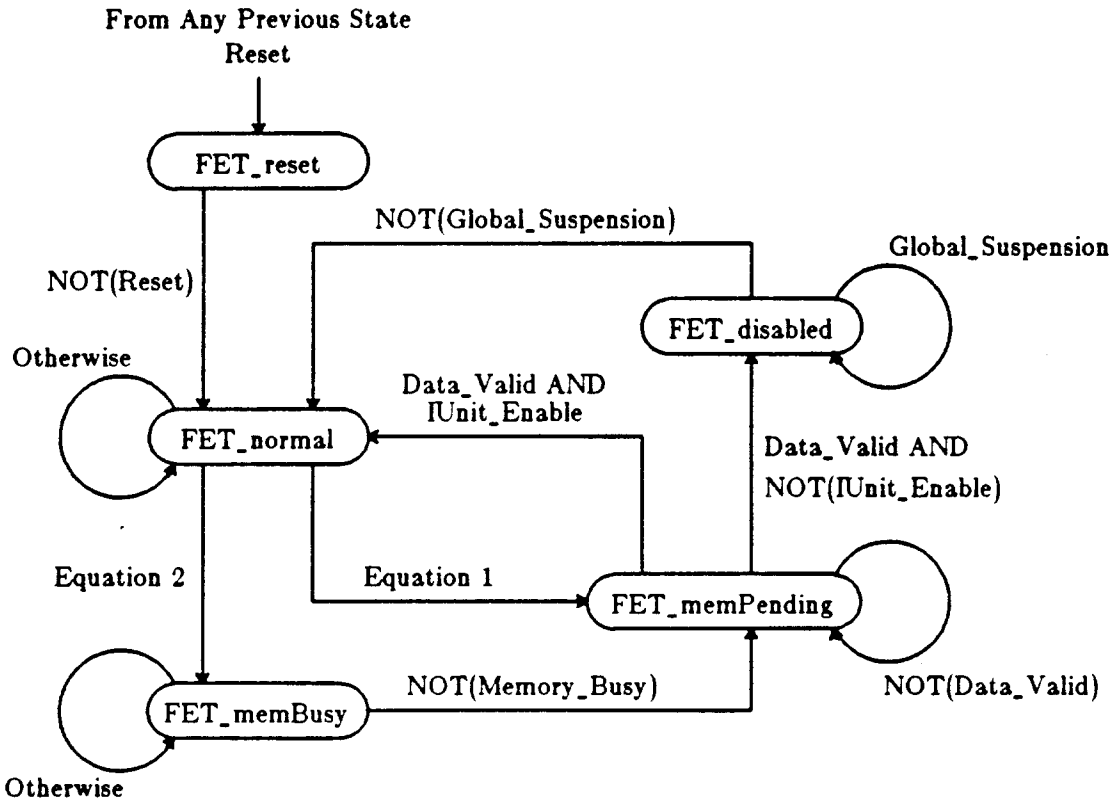
FET_reset: The Reset signal forces the Fetch_FSM into the FET_reset state from any previous state. It will remain in this state as long as Reset is true. The IUnit engages in the following activities while in the FET_reset state:

1. The IUnit ignores the program counter from the EUnit (PC_bus).
2. The IUnit does not read or write any instructions or address tags from its memory arrays. Previous fetches or prefetches from the ECache are ignored.
3. The IUnit sends out the READ_PC internal instruction repeatedly to the EUnit (Ins_bus) during phi3. The TRAP_PC instruction has already been sent out on the previous cycle. In other words, the TRAP_PC signal is generated after the Reset signal is true and before the IUnit moves into the FET_reset state.
4. No Fetches are initiated by the IUnit.
5. The Fetch_FSM keeps Starting_Prefetch false so that the Prefetch_FSM does not initiate any prefetches.

FET_normal: The Fetch_FSM moves to the FET_normal state on the cycle after the Reset signal is false. Changes on the Reset, Global_Suspension, Miss, or Flush signals move the Fetch_FSM out of the FET_normal state. The IUnit performs the following activities while

in the FET_normal state:

1. The IUnit latches the program counter from the EUnit (PC_bus). In effect, the IUnit has accepted an instruction request from the EUnit.
2. The IUnit may read and write instructions and address tags from its memory arrays. These actions are also dependent on other input signals. For example, the IUnit only writes information into the instruction memory array when Data_Valid is true.
3. The IUnit passes one of three instructions to the EUnit while in the FET_normal state. It passes the requested instruction to the EUnit (Ins_bus) if it detects a hit. It sends a MISS instruction if a miss occurs or a TRAP_PC instruction if the Reset signal is true.
4. The IUnit initiates fetches from the external cache (Add_bus) if Miss or Flush are true and Memory_Busy is false.
5. The Fetch_FSM keeps Starting_Prefetch false until Miss, Flush or Global_Suspension are true.



Equation 1: NOT(Global_Suspension) AND (Miss OR Flush)
AND Memory_Busy

Equation 2: NOT(Global_Suspension) AND (Miss OR Flush)
AND NOT(Memory_Busy)

Figure 3-1: Fetch State Diagram

FET_memBusy: The Fetch_FSM moves to this state if Miss or Flush are true and the external cache is busy (Memory_Busy=true). The IUnit activities while in this state are:

1. The IUnit ignores the program counter from the EUnit (PC_bus).
2. The IUnit does not read or write any instructions or address tags from its memory arrays.
3. The IUnit passes the MISS instruction to the EUnit (Ins_bus).
4. Fetches may be initiated by the IUnit during this state if the Memory_Busy signal is false. For example, assume a Miss occurred in the IUnit during the previous cycle and Memory_Busy was true. Therefore, the Fetch_FSM moved into the FET_memBusy state. The IUnit initiates a fetch as soon as Memory_Busy is false.

FET_memPending: The Fetch_FSM moves to this state if Miss or Flush are true, and the external cache is not busy (Memory_Busy=false). In this state, the IUnit is waiting for a fetch request from the external cache to be satisfied. The IUnit activities in the FET_memPending state are:

1. The IUnit ignores the program counter from the EUnit (PC_bus).
2. The IUnit does not read any instructions or address tags from its memory arrays. However, the IUnit may write into its instruction array if Data_Valid is true.
3. The IUnit passes the MISS instruction to the EUnit (Ins_bus).
4. No fetches may be initiated by the IUnit during this state. The IUnit is presently waiting for its last fetch to be satisfied.
5. The Fetch_FSM signals the Prefetch_FSM that it can start making sequential prefetches.

FET_Disabled: The Fetch_FSM moves to this state if the IUnit_Enable signal is false. The FET_disabled state permits the CPU to function without the IUnit. The only activity that the IUnit performs during this state is to pass instructions to the EUnit (Ins_bus). No prefetches are initiated while the IUnit is in this state. The IUnit may pass either the last requested instruction, a MISS or a TRAP_PC.

Prefetch Finite State Machine: The state of the Prefetch_FSM determines whether any sequential prefetching will be done by the IUnit. The Prefetch_FSM PLA has six inputs: Reset, IUnit_Enable, Prefetch_Enable, Memory_Busy, Flush, and Starting_Prefetch. The five states in the Prefetch_FSM are shown in Figure 3-2.

PF_reset & PF_disabled: The Prefetch_FSM is forced into the PF_reset state from any other state when the Reset signal is true. The Prefetch_FSM will move into the PF_disables state

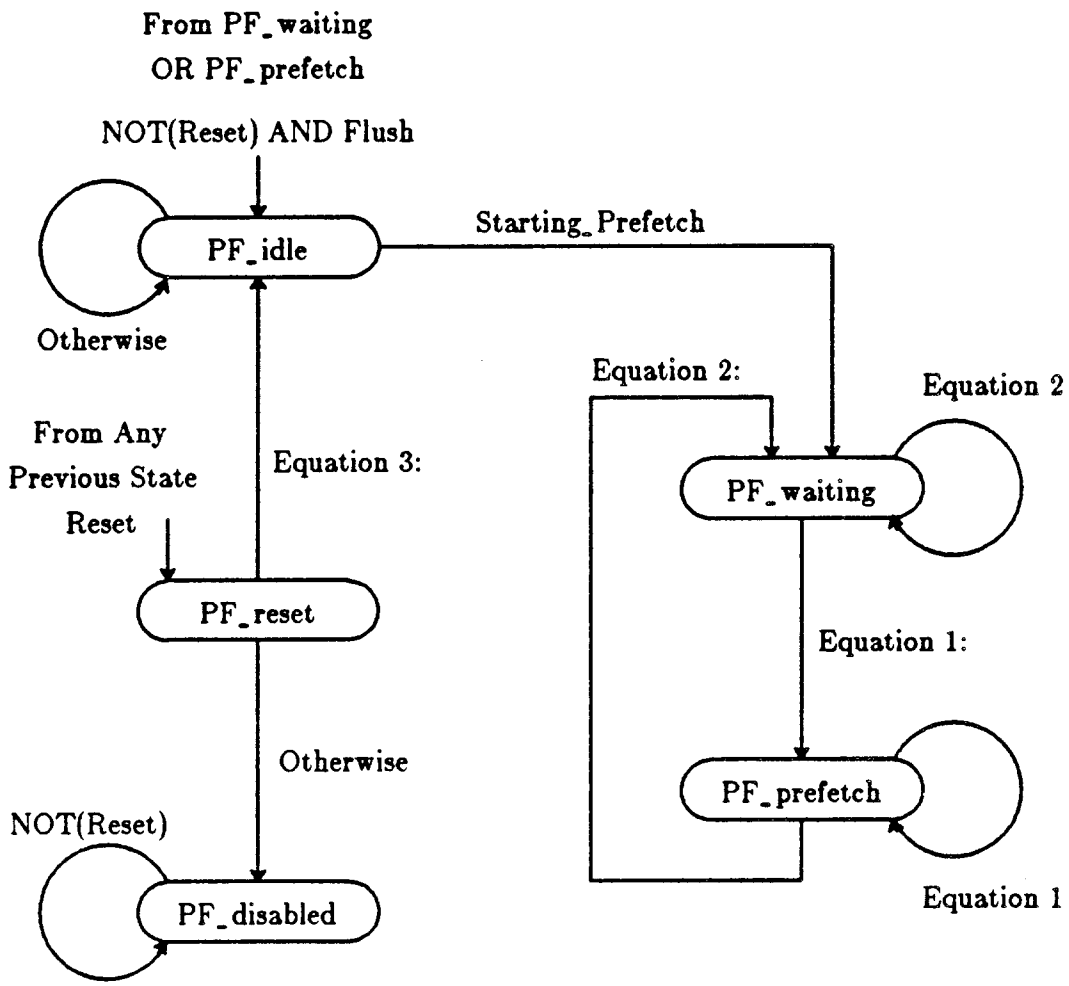
after a reset if IUnit_Enable or Prefetch_Enable is false. No Prefetching is initiated from either of these states.

PF_idle: The idle state supports the prefetch-on-miss algorithm. The prefetcher will remain in this state until the Fetch_FSM receives a Miss, Flush, or a Global_suspension. While in the idle state, the Prefetch_FSM repeatedly latches in the address of the last instruction that was fetched by the IUnit (Add_bus). It increments this address so that it is prepared to prefetch the next sequential instruction if requested by the Fetch_FSM (Starting_Prefetch=true).

PF_waiting & PF_prefetch: The Prefetch_FSM is attempting to prefetch sequential instructions while in either of these states. In both cases, the Prefetch_FSM latches in the address of the last instruction that was either fetched or prefetched. It increments this address so that it may initiate a prefetch. The Prefetch_FSM moves between these two states when either the Memory_Busy signal and the Starting_Prefetch signal change. A new prefetch is initiated on every cycle when both the Memory_Busy and the Starting_Prefetch signals are false. Prefetches are initiated by sending the next sequential instruction address to the external cache (Add_bus) and asserting the Prefetch_Request signal. The Prefetch_FSM also writes incoming prefetched instructions into the instruction memory array when Data_Valid is true. It will continue in this mode indefinitely until either Flush or Reset are true.

3.2 Control Implementation Philosophy

Computer architects have often complained about the difficulties of control design in computers. A familiar adage states that the control design only occupies 10% of the computer hardware but requires 90% of the total design time. Fortunately, this has not been the case in the SPUR CPU design. Careful planning on the CPU functional level has helped segment the control blocks into manageable pieces [Kong86a], [Kong86b]. The IUnit control implementation occupies 8% of the total IUnit area and only required 50% of the total design time.



Equation 1: NOT(Starting_Prefetch OR Memory_Busy)

Equation 2: Starting_Prefetch OR Memory_Busy

Equation 3: NOT(Reset) AND IUnit_Enable AND Prefetch_Enable

Figure 3-2: Prefetch State Diagram

Design Constraints: The implementation philosophy that I pursued was shaped by several characteristics of the CPU design. The CPU floor plan was designed so that the upper and lower datapaths were approximately the same width. The height of the cells in each datapath were matched so that circuit designs could be shared. The height of the static memory cell used in both the register file and the IUnit also match the datapath cells. The area allocated for the IUnit was also an integral part of the CPU floor plan. The CPU upper and lower datapath layouts were

already complete, and fixed the IUnit width dimension. Therefore, I was forced to establish some tight pitches that influenced my approach. Secondly, there is a high probability that logic changes will be made in the IUnit control. Therefore, the layout was planned to support these changes. Finally, the IUnit control must issue low-level control signals during all four clock phases. This constraint limited the amount of logic that could be implemented in PLA's.

Generic IUnit Control Implementation: The two finite state machines PLA's within the IUnit are quite similar. This also holds true for the rest of their control functional blocks. Figure 3-3 shows a generic block diagram which illustrates either the Fetch_FSM or the Prefetch_FSM control functions [Kong86b]. The IUnit control inputs come from four sources: the EUnit, the instruction memory array, the tag comparison block, and the other finite state machine. Section 3.3 lists the input and output signals for both finite state machines. The Input Logic & Latches in Figure 3.3 are conventional static gates. The State PLA's were also simple to implement because generation tools and templates had already been developed [Scott86], [Jeon86]. Therefore, the output logic implementation is the focus of the remainder of this section.

Output Logic Constraints: The output logic block was the most challenging part of the IUnit control implementation. Several constraints complicated its design. First, a PLA could not be used to implement the output logic because the output signals are used during the same phase that they are evaluated. The targeted performance for this output logic was under 4 ns. Since the logic equations in this block are likely to change, I tried to implement a layout that was structured and general. I hoped to implement the output logic with standard logic cells that could be fit together in mosaics of all possible patterns. Ideally, changing one standard cell would not effect the layout of any other cell. I also hoped to bus all input signals into this standard cell mosaic so that any logic gate could be connected to any desired input. Unfortunately, the layout area required for this completely general approach was not practical. The output logic would be 2.5 times taller than my available height.

Output Logic Implementation: In my final implementation strategy I made tradeoffs between speed, area, and a completely general approach. The output logic equations vary widely

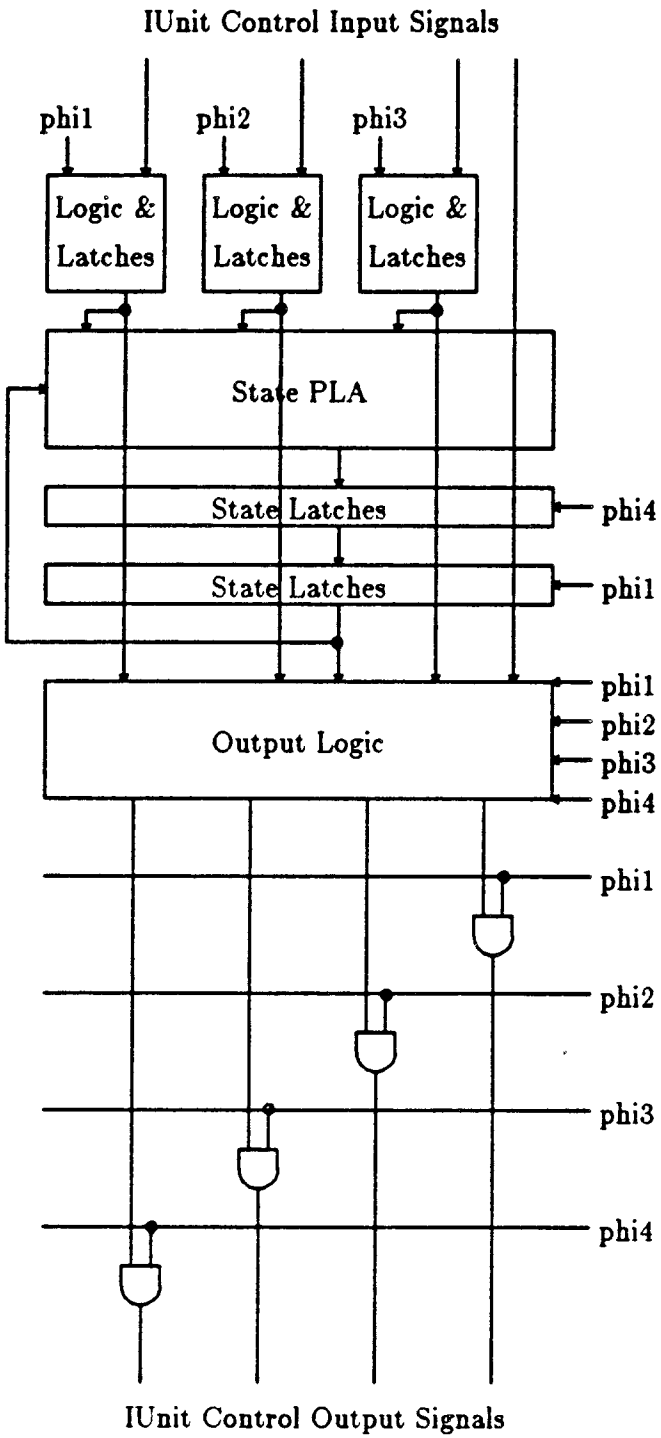


Figure 3-3: Generic IUnit Control Block Diagram

in complexity. The simplest equations are just a buffered version of the present state. The most complex output logic equation is shown below (MISS_To_InsBus). The 'FET_...' inputs signals identify the present state of the Fetch_FSM. Therefore, only one of the following signals are true:

FET_normal, FET_disabled, FET_memBusy, or FET_memPending. In the actual implementation, all signals proceeded by a NOT operator were inverted prior to the output logic gate.

```
MISS_To_InsBus= {FET_normal AND NOT(Reset) AND NOT(Global_Suspension)
                 AND (Miss OR Flush)}
                 OR {FET_disabled AND NOT(Reset) AND Global_Suspension}
                 OR {(FET_memBusy OR FET_memPending) AND NOT(Reset)}
```

Domino Logic Standard Cells: This complex logic output equation emphasizes several important points. If a completely general implementation approach were adopted, the same amount of area should be reserved for all output logic equations. This approach consumed far too much area. A circuit implementation must be selected that minimizes both delay time and area. Domino logic gates were selected to best satisfy these criteria. Figure 3-4 shows a domino logic gate implementation for the MISS_To_InsBus output logic equation. Two AND*OR domino logic gates were used in this implementation. All the output logic equations use at most two series domino logic gates to ensure adequate performance. No more than 5 NMOS devices are connected in series in any logic gates to avoid threshold voltage shifts due to an excessive backgate voltage. Layouts plots for two standard domino logic gates are shown in Figure A-4 in Appendix A: a 4 Input AND Gate and a 7 Input AND*OR Gate.

Standard Cell Mosaic Implementation: I made several implementation tradeoffs in my domino standard cells to improve the layout density. I did not bus all input signals through the mosaic. Instead, I left several free routing channels to carry signals required at a later time. A typical output logic mosaic is shown in Figure 3-5. The input signals are bussed through the logic cells. This corresponds to the top and bottom of Figure 3-5. I shared area between outputs signals so that complex logic equations could use excess space from simple logic equations. This technique is illustrated in Figure 3-5. Each output is allocated enough room for two logic cells. For example, Output Signal 1 uses a 4 Input AND Gate and a 2 Input OR Gate. Output Signal 2 only requires one 3 Input AND Gate, therefore, one of its logic cells is free. The most complex

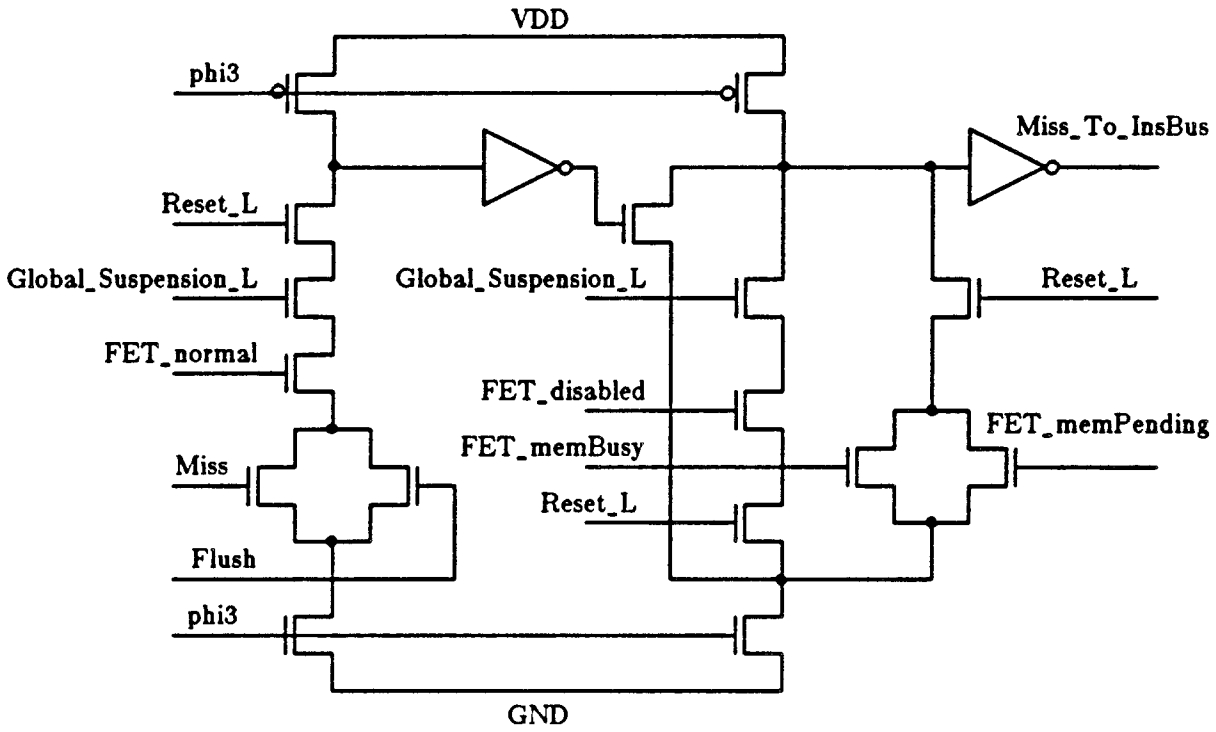


Figure 3-4: Domino Output Logic Gate

logic gate that can fit into a single cell height is a 4 Input AND. More complex AND*OR gates can be implemented by using two single cell heights. Sharing area between output signals that have complex and simple logic equations is also demonstrated in Figure 3-5. For example, Output Signal 3 only uses one 3 Input AND Gate so it can contribute the space occupied by one logic cell to another output. In this case, Output Signal 4 has a more complex logic equation that requires one 6 Input AND*OR Gate and one 4 Input AND Gate. Therefore, it uses the free space available from Output Signal 3. In the extreme case, an output logic equation requires two AND*OR gates. Fortunately, there are some outputs that do not require any logic cells. Sharing area between output signals was effectively used to reduce the area of the output logic mosaic.

Limitations of the Output Logic Mosaic: There are several disadvantages with the output logic mosaic approach that I implemented. First, changing the logic equation for one output may effect other outputs. Hopefully, most logic changes will only require modifying the input

Logic Cells	Logic Cells	Output Buffer
4 Input AND Gate	2 Input OR Gate	Output Signal 1
free cell	3 Input AND Gate	Output Signal 2
6 Input AND*OR Gate	3 Input AND Gate	Output Signal 3
	4 Input AND Gate	Output Signal 4
7 Input AND*OR Gate	5 Input AND*OR Gate	Output Signal 5
		Output Signal 6

Figure 3-5: Typical Output Logic Mosaic

connections. In addition, there would be little difficulty substituting a simpler output logic gate for a more complex gate. However, problems could arise if more complex logic gates are needed. In the worst case, a new custom logic gate could be packed into the available space. The general purpose logic cells that I designed keep most of the metal 1 routing channels open. This makes them easier to connect to any of the available inputs. If necessary, this routing area could be packed with additional transistors. The second limitation of this approach is that the logic gate inputs must be connected to the input signals by hand. This is an error prone and time consuming process. I'm confident that software tools could be developed to automate the output logic generation. Finally, it may be possible to redefine the control architecture to capture more of the problem into the state PLA's. The PLA's are quite area efficient in comparison to the output logic and are already supported by automatic generation tools. One possible approach is to use PLA's to perform all the output logic functions that can be evaluated one phase earlier than necessary. Domino output logic could then be reserved for only those parts of the output logic that must be evaluated during the final phase.

3.3 Fetch and Prefetch Control Implementations

This section summarizes the implementation details of the Fetch_FSM and the Prefetch_FSM. The layout of the IUnit control occupies 8% of the total area in the IUnit and contains approximately 2300 transistors. The generic control block diagram, presented in the previous section, proved to be impractical in several cases. For example, the regularity of my output logic implementation had to be abandoned to achieve adequate performance margins for some output signals. This section is divided into three parts. First, a complete list of input and output signals for the Fetch_FSM and Prefetch_FSM is reviewed. The output logic mosaics implemented for both of these finite state machines is then presented. Finally, SPICE simulation results for the worst case speed path in the control block are discussed.

Fetch and Prefetch Input and Output Signals: A complete list of the input signals used by the Fetch_FSM and Prefetch_FSM are shown in Table 3-2. Input signals used by the IUnit control arrive during all four phases. However, input signals that arrive during phi4 are latched on the following phi1 to avoid race conditions between these signals and the PLA's. All of the input signals in Table 3-2 have been introduced in previous sections except the Write_Fetch signal.

The Write_Fetch signal connects the Fetch output logic to the Prefetch output logic. Write_Fetch is important because potentially it is the slowest signal path in the IUnit control design. The speed path is illustrated by the two following logic equations:

```
Write_Fetch        = {NOT(Reset_IUnit) AND Data_Valid AND FET_memPending}

Write_Instruction

                   = {NOT(Reset) AND NOT(Flush)}
                   {AND Data_Valid AND PF_prefetch}
                   {OR Write_Fetch}
```

Signal or Bus Name	Signal Timing Information			
	phi1	phi2	phi3	phi4
EUnit and ECache Inputs to IUnit Control (see Table 2-1)				
Enable_KPSW_Set				*Changes
Prefetch_KPSW_Set				*Changes
Pipeline_Not_Suspended				*Changes
Load_OPCode	Changes			
Store_OPCode	Changes			
lowTOup_OPCode	Changes			
Invalidate_OPCode	Changes			
Invalidate_Trap	Changes			
Reset_IUnit		Changes		
Cache_Busy		Changes		
Cache_Data_Valid			Valid	
PC_bus <30 bits>	EUnit		Pre-Charge	
Address_bus <30 bits>	Pre-Charge	EUnit	Pre-Charge	EUnit or IUnit
Data_bus <40 bits>	EUnit	Pre-Charge	ECache	Pre-Charge
IUnit Derivatives of EUnit and Ecache Inputs (see Table 3-1)				
IUnit_Enable	Changes			
Prefetch_Enable	Changes			
Global_Suspension	Changes			
Reset		Changes		
Flush		Changes		
Memory_Busy		Changes		
Data_Valid			Changes	
IUnit DataPath Inputs to IUnit Control				
Block_Miss		Valid		
Instruction_Miss		Valid		
Fetch_FSM Outputs to Prefetch_FSM Inputs				
Starting_Prefetch			Valid	
Write_Fetch				Valid

*Signals arrive on phi4 but are not used until the following phi1.

Change - indicates that the signal is always asserted. It only changes during the designated phase.

Valid - indicates that the signal may only be asserted true during the designated phase. Signal is false during all other phases.

Table 3-2: Fetch and Prefetch Input Signals

The Write_Fetch signal would normally be evaluated during phi4 because the Data_Valid signal does not change until phi3. When using domino logic gates, it is essential that none of the inputs make a high to low transition during the evaluation period. My standard domino gates avoid this problem by always evaluating the logic function on the phase after the last signal changes. The slow signal path arises because the Write_Instruction signal must wait for the Write_Fetch signal. Therefore, this speed path essentially suffers from two delays through the output logic: one for the Fetch_FSM and the second for the Prefetch_FSM.

This signal delay path was removed by using a static logic gate for the Write_Fetch signal. This technique works because the Data_Valid signal is valid midway through phi3. The Write_Fetch signal has enough time (15 ns) to propagate through the 3 input static gate and settle at the input of the Write_Instruction output logic gate before the beginning of phi4.

The Fetch_FSM and Prefetch_FSM output signals, listed in Table 3-3, provide direct control of the instruction memory array, tag comparison logic, and prefetcher. These output signals are described in subsequent sections of this chapter.

Signal or Bus Name	Signal Timing Information			
	phi1	phi2	phi3	phi4
Instruction Memory Array Control Signals				
Invalidate_Block		Valid		
Enable_Block		Valid		
Read_Instruction		Valid		
Instruction_To_InsBus			Valid	
MISS_To_InsBus			Valid	
TRAPCALL_To_InsBus			Valid	
READPC_To_InsBus			Valid	
Ins_bus <32 bits>			IUnit	
Read_MemLatch				Valid
Write_Instruction				Valid
Tag Comparison Logic Control Signals				
Load_FetchPC	Valid			
Invalidate_Tag		Valid		
Bypass_Tag_Decoder		Valid		
Write_Tag				Valid
Add_bus <30 bits>	Pre-Charge	EUnit	Pre-Charge	EUnit or IUnit
Prefetcher Control Signals				
FetchPC_To_AddBus				Valid
IncrementedPC_To_AddBus				Valid
Load_ReferencePC				Valid
Fetch_FSM Outputs to Prefetch_FSM Inputs				
Starting_Prefetch			Valid	
Write_Fetch				Valid
Outputs to the Execution Unit				
Fetch_Request			Valid	
Prefetch_Request			Valid	

Valid - indicates that the signal may only be asserted true during the designated phase. Signal is false during all other phases.

Table 3-3: Fetch and Prefetch Output Signals

Fetch and Prefetch Output Logic Mosaics: The output logic implementation followed the structured standard cell approach whenever possible. Actually, only the Write_Fetch signal needed to use a static output gate. However, several other output signals had potential race

problems. For example, the Fetch_Request and Prefetch_Request output signals are evaluated during phi3. Both these signals are derived from the Memory_Busy signal. This signal does not become valid until late in the phi2 cycle. Extra attention had to be given to this delay path to ensure that Memory_Busy was valid before the beginning of phi3.

The output logic mosaics for the Fetch and Prefetch finite state machines are shown in Figure 3-5 and Figure 3-6 respectively. The Fetch output logic supports 14 output signals in the same number of standard cell rows. The Prefetch output logic contains six output signals in eight logic cell rows. The area consumed by the Fetch output logic mosaic set the critical dimension of the IUnit control block. Figures A-5 and A-6 in Appendix A show the layout plots of the Fetch and the Prefetch output logic mosaics respectively. Layout plots for the entire Fetch_FSM and Prefetch_FSM blocks are shown in Figures A-7 and A-8. The output logic consumes about 50% of the total control area in both of these plots.

Logic Cells	Logic Cells	Output Buffer
4 Input AND Gate	6 Input AND*OR Gate (d4AND2OR)	Fetch_Request
fet_free Cell		Load_FetchPC
5 Input AND*OR Gate (d4AND1OR)	3 AND*OR Gate	Starting_Prefetch
	fet_free Cell	READPC_To_InsBus
4 Input AND Gate	6 Input AND*OR Gate (d4AND2OR)	FetchPC_To_AddBus
2 Input OR Gate		TRAPCALL_To_InsBus
fet_free Cell	5 Input AND*OR Gate (d4AND1OR)	Write_Tag
4 Input AND Gate		Read_MemLatch
5 Input AND*OR Gate (d4AND1OR)	7 Input AND*OR Gate (d3AND4OR)	MISS_To_InsBus
		Invalidate_Block
fet_free Cell	6 Input AND*OR Gate (d5AND1OR)	Instruction_To_InsBus
3 Input AND Gate		Read_Instruction
fet_free Cell	fet_free Cell	Enable_Block
fet_free Cell	3 Input Static AND Gate	Write_Fetch

Figure 3-6: Fetch Output Logic Mosaic

Logic Cells	Logic Cells	Output Buffer
2 Input OR Gate	4 Input AND Gate	Prefetch_Request
2 Input AND Gate	2 Input AND Gate	Invalidate_Tag
pf_free Cell	2 Input AND Cell	Bypass_Tag_Decoder
4 Input AND Gate	2 Input OR Gate	Write_Instruction
5 Input AND*OR Gate (d4AND1OR)	2 Input OR Gate	Load_ReferencePC
	4 Input AND Gate	free_OB Cell
2 Input OR Gate	2 Input OR Gate	IncrementedPC_ToAddBus
4 Input AND Gate	pf_free Cell	free_OB Cell

Figure 3-7: Prefetch Output Logic Mosaic

Control Design Critical Signal Delay Path: The critical signal delay path in the IUnit control implementation is in the output logic. The output logic is the only block within the IUnit control that must evaluate outputs during every phase. Furthermore, these output signals must be used during the same phase that they are evaluated.

SPICE simulations were performed on the MISS_To_InsBus output because it uses the most complex domino logic gate. The schematic for this gate is shown in Figure 3-4. SPICE simulations estimate a 2.9 ns (typical) delay time from the beginning of the evaluation phase to a valid output signal. The delay time is measured from the rising edge of the clock to the rising edge of the output signal. The delay time between these rising edges is measured at 2.5 volts. The SPICE simulation data is included in Appendix C.

3.4 Instruction Buffer Implementation

The instruction buffer is a special purpose on-chip memory. It stores 128 instructions and their associated valid tags in 4224 static memory cells. The instruction buffer occupies 77% of the total area in the IUnit and contains approximately 32,000 transistors. A layout plot of the instruction buffer is shown is Figure A-9 in Appendix A. The control of the instruction buffer is simplified with the four phase clocking scheme implemented throughout the SPUR CPU. Two phases are available for precharging, one phase is available for a read operation and one phase is available for a write operation. This section briefly describes the instruction buffer on the block diagram level. The timing of these blocks and their control signals is also summarized. Finally, constraints that influenced the instruction buffer implementation and SPICE simulation results for the critical signal delay path are presented.

Instruction Buffer Functional Description The functional blocks in the instruction buffer are shown in Figure 3-8 [Kong86b]. The IArray is a 128 x 33 array of static memory cells. The Decoders can access any single instruction in the IArray or one entire block (8 instructions). This feature is used to invalidate an entire block in one access. The MemoryDataLatch stores data from the Data_bus. It writes this data into the IArray one the next phase. Data read from the IArray is stored in the InsReg. The InsMux is a four-way multiplexer that controls which instruction is sent to the EUnit on the Ins_bus. The IBRead, IBWriteM, and IBWriteS registers store addresses for read or write operations. The 'M' and 'S' subscripts denote master and slave latches. The WordValid block contains the logic that writes and reads the sub-block valid bit from the last row of memory cells. It informs the IUnit control whether an Instruction_Miss has occurred during a read operation.

Instruction Buffer Timing: The control signals necessary to support read and write operations are simplified by having four clock phases available. The timing for the functional blocks is summarized in Table 3-4. The IArray is pre-charged during phi1 and phi3. New addresses for read and write operations are sent during phi1 and phi3 from the IBRead and IBWriteS registers to the Decoders. Normally, the InsReg is only updated on phi2 after a read

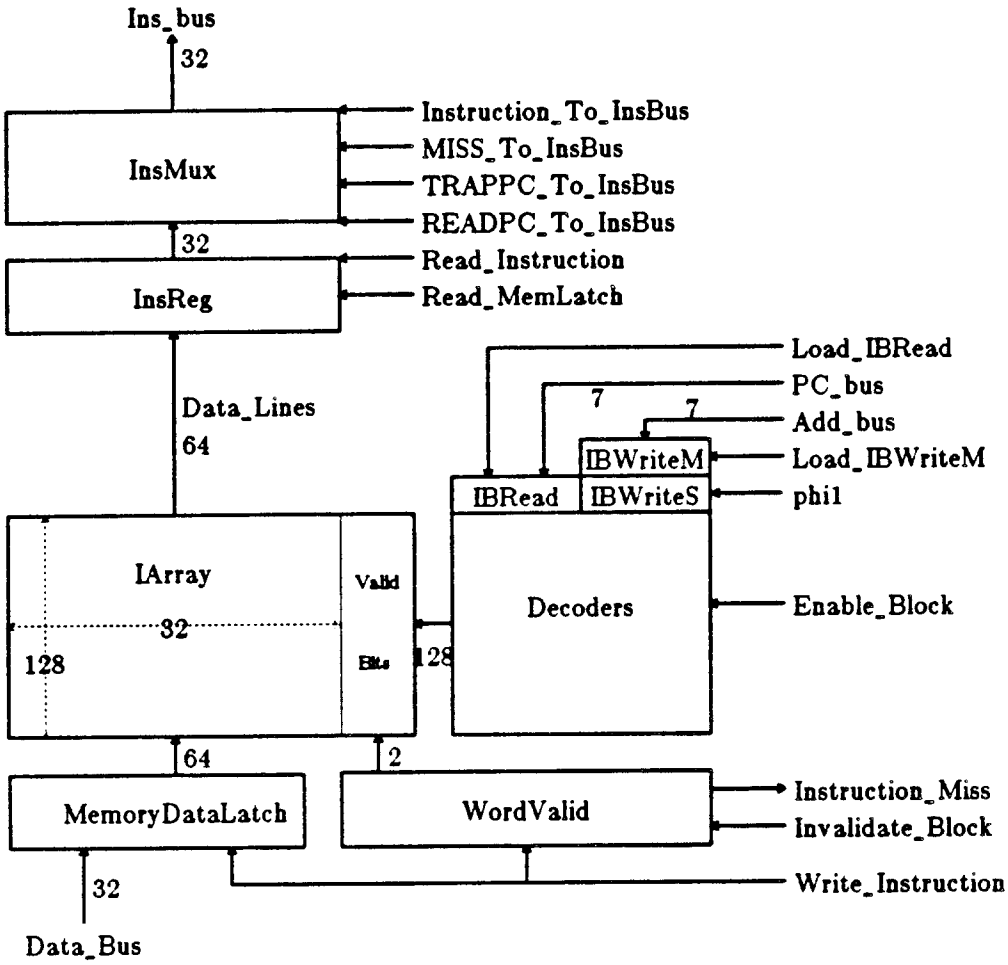


Figure 3-8: Instruction Buffer Block Diagram

operation. The control implementation provides an additional signal so the InsReg can latch the data being written into IArray if desired. This feature could be used for debugging or could allow chips with defects in the IArray to still be used. The InsMux sends one of four instructions to the EUnit over the Ins_bus. This instruction may be the instruction stored in the InsReg or three hard wired instructions: MISS, READ_PC, or TRAP_CALL.

The control signal timing is also shown in Table 3-4. The purpose of most of these signals is self evident from their name and connection in Figure 3-8.

Instruction Buffer Implementation Constraints: Several implementation decisions within the CPU had a significant impact on the instruction buffer implementation. Dave Lee had already completed layout of the lower datapath in the EUnit. Most of the lower datapath was

Signal or Bus Name	Signal Timing Information			
	phi1	phi2	phi3	phi4
Instruction Memory Array Functional Blocks				
IArray	Pre-Charge	Read	Pre-Charge	Write
Decoders	New Address	Access IArray	New Address	Access IArray
MemDataLatch	Pre-Charge IArray		Latch Data_bus	Write IArray
InsReg		Latch IArray		{Latch IArray}
InsMux			Drive Ins_bus	
IBRead	Latch PC_bus Drive Address			
IBWrite			Drive Address	*Latch Add_bus
WordValid	Pre-Charge	Read Valid Bit	Pre-Charge	Write Valid Bit
Instruction Memory Array Signals				
Invalidate_Block		Valid		
Enable_Block		Valid		
Read_Instruction		Valid		
Instruction_Miss		Valid		
Instruction_To_InsBus			Valid	
MISS_To_InsBus			Valid	
TRAPCALL_To_InsBus			Valid	
READPC_To_InsBus			Valid	
Read_MemLatch				Valid
PC_bus <30 bits>	EUnit		Pre-Charge	
Add_bus <30 bits>	Pre-Charge	EUnit	Pre-Charge	EUnit or IUnit
Data_Bus <32 bits>	EUnit	Pre-Charge	ECache	Pre-Charge
Instruction_bus <32 bits>			IUnit	

* Signals arrive on phi4 but are not used until the following phi1.

{} Indicates that this feature is included for debugging, not normal operation.

Valid - indicates that the signal may only be asserted true during the designated phase. Signal is false during all other phases.

Table 3-4: Instruction Buffer Timing

implemented to be consistent with the height of the register file. The register file contains 5520 memory cells and measures 4094 x 5729 μm^2. Wook Koh was also well into the upper datapath layout before I started working on the IUnit. Functional blocks in the upper datapath were designed assuming the IUnit would be approximately 6000 x 4000 μm^2. Any drastic change in the size or the aspect ratio of the IUnit would force implementation changes in many other functional

blocks. Alternative memory array implementations are discussed in Chapter 4.

Instruction Buffer Critical Signal Delay Path: The critical signal delay path in the instruction buffer includes reading the sub-block valid bit from the IArray and driving it to the IUnit control block. This is the slowest signal delay path in the IUnit. SPICE simulation results show that the total time for this delay path is 14.7 ns (typical). A read operation from one of the instruction memory cells into the InsReg takes 12.5 ns (typical). Therefore, the sub-block valid bit increases the critical signal delay path in the IUnit by 17.6%. The SPICE simulation data and assumptions are included in Appendix C. The delay time is broken into three components below:

Select Line Delay	= 4.2 ns (select line = 2.5v)
Data Line Delay	= 7.3 ns (data line = 2.5v)
Miss Line Delay	= 3.2 ns (Instruction_Miss = 2.5v)

3.5 Tag Comparison & Prefetcher Implementations

The tag comparison and prefetcher support additional datapath functions in the IUnit. This block occupies 15% of the total area in the IUnit and contains approximately 4600 transistors. A layout plot of the Tag Comparison and Prefetcher block in shown in Figure A-10 in Appendix A. This section provides a functional description of these blocks and timing information. SPICE simulation results are presented at the end of this section.

Tag Comparison & Prefetcher Functional Description: The tag comparison block maintains the address tag information for the IUnit. This block compares its address tags against requested instruction addresses and informs the IUnit control whether a hit or miss occurs. The tag comparison block must also support its 384 bit static memory cells. The prefetcher block latches instruction address that are being fetched or prefetched on the Add_bus and increments the three lowest order bits. This incremented instruction address is sent on the Add_bus when a prefetch is initiated.

The functional blocks within the tag comparison and prefetcher are shown in Figure 3-9. The TArray is a 16 x 24 array of static memory cells identical to the IArray cells. The decoders are also similar the instruction buffer decoders. The Comparator compares the address tag stored in the TArray with the tag stored in the FetchPC latch. FetchPC holds a program counter address tag. It also handles precharging in the TArray.

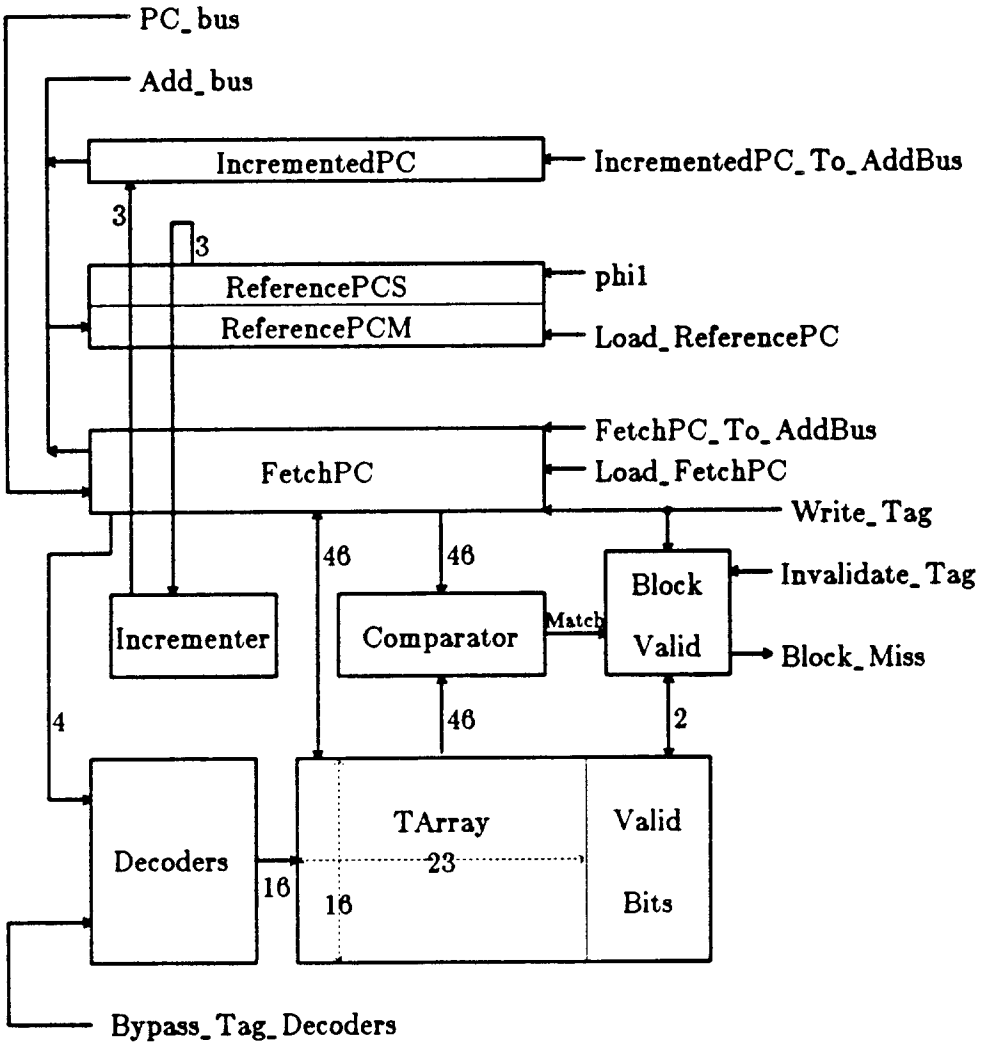


Figure 3-9: Tag Comparison and Prefetcher Block Diagram

The BlockValid cell OR's the results of the Comparator with the block valid bit read from the TArray. It informs the IUnit control block whether a Block_Miss has occurred. Only four functional blocks are used in the prefetcher: ReferencePCM, ReferencePCS , Incrementer, and

IncrementedPC. Both the ReferencePCM and ReferencePCS are 30-bit latches. The Incrementer is a simple wrap-around counter. It increments the three lowest order bits of the instruction address stored in the ReferencePCS latch. The IncrementedPC is a simple buffer that writes the incremented instruction address on the Add_bus.

Tag Comparison and Prefetcher Timing: The timing of the functional blocks and signals is summarized in Table 3-5. The purpose of these signals is self evident from their names and connections in Figure 3-9.

Signal or Bus Name	Signal Timing Information			
	phi1	phi2	phi3	phi4
Tag Comparison and Prefetcher Functional Blocks				
TArray	Pre-Charge	Read	Pre-Charge	Write
Decoders	New Address	Access TArray	New Address	Access TArray
Comparator		Compares Tags		
FetchPC	Latch PC_bus Drive Decoder		Drive Decoder	Drive Add_bus
ReferencePC				*Latch Add_bus
Incrementer		Increment Address		
IncrementedPC				Drive Add_bus
BlockValid	Pre-Charge	Read Valid Bit	Pre-Charge	Write Valid Bit
Tag Comparison and Prefetcher Signals				
Load_FetchPC	Valid			
Invalidate_Tag		Valid		
Bypass_Tag_Decoder		Valid		
Block_Miss		Valid		
FetchPC_To_AddBus				Valid
IncrementedPC_To_AddBus				Valid
Load_ReferencePC				Valid
Write_Tag				Valid
Address_bus <30 bits>	Pre-Charge	EUnit	Pre-Charge	EUnit or IUnit
PC_bus	EUnit		Pre-Charge	

* Signals arrive on phi4 but are not used until the following phi1.

Valid - indicates that the signal may only be asserted true during the designated phase. Signal is false during all other phases.

Table 3-5: Tag Comparison and Prefetcher Timing

Comparator Implementation: A two level domino logic implementation of the comparator was used to minimize its speed delay and area. Figure 3-10 shows the schematic of the comparator cell. The first level of domino logic gate provides a single bit comparison. The Not_Bit_Match signal, shown in Figure 3-10, indicates whether the tag bit and the data bit are the same. All 23-bits are OR'ed together in the second level of domino logic gate. The size of the two series devices that implement the final OR function (m1 & m2) were optimized to give a fast transition on the Match line. The speed of this signal was found to reach a maximum with devices of 40 μm to 45 μm wide. A layout plot of the Comparator is shown in Figure A-11 in Appendix A.

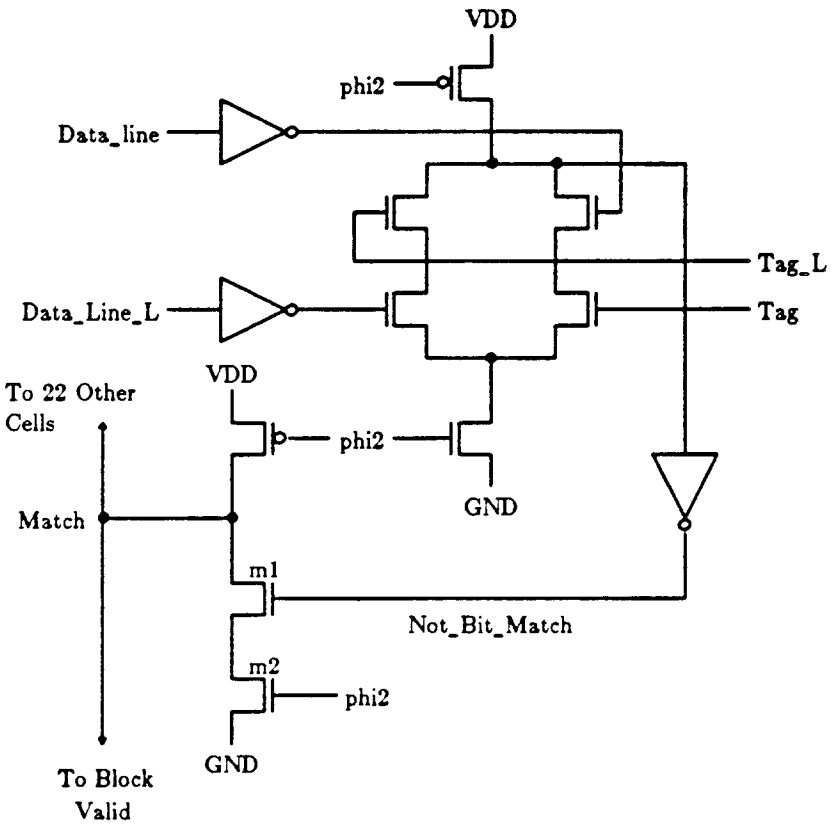


Figure 3-10:Comparator Schematic

Tag Comparison Critical Signal Delay Path: The critical signal delay path in the tag comparison block occurs during a tag read and comparison operation. The signal delay path involves the following signals: Decoders select line, memory cell data line, tag comparison Match

signal, and the Block_Miss signal. The SPICE simulation results show that the total time for this signal delay path is 9.3 ns (typical). SPICE simulation data and assumptions are included in Appendix C. The delay time is broken into its four components below:

Select Line Delay - 2.5 ns (select line = 2.5v)

Data Line Delay - 1.7 ns (data line = 2.5v)

Comparator Delay - 3.2 ns (Match = 2.5v)

BlockValid Delay - 1.9 ns (Block_Miss = 2.5v)

4. ALTERNATIVE INSTRUCTION UNIT IMPLEMENTATIONS

In this chapter, I examine instruction unit implementations that are similar to the SPUR IUnit. I varied three IUnit design parameters: memory array size, sub-block size, and associativity. None of these alternatives would require any functional changes in the SPUR CPU. However, many of these alternatives would require large portions of the CPU layout to change. In this chapter, I assume that the CPU layout is not a factor. In actual practice, this assumption could not be further from the truth. Organizing a complex VLSI chip into an efficient implementation is one of the most difficult tasks that chip designers face. Poor planning often forces portions of of the design to be re-implemented. As I stated previously in section 3.2, the SPUR instruction unit size and aspect ratio were determined when the chip floor plan was established. This chapter also assumes that the speed of the CPU is limited by the instruction unit. In the SPUR CPU, the critical signal delay path occurs during a read in the register file.

I selected memory array size, sub-block size, and associative implementation alternatives because they have been shown to significantly impact the performance of cache memories [Smith82], [Good83], [Hill84]. Furthermore, these alternatives do not require drastic changes in the control complexity of the IUnit. I compare these alternative implementations using two metrics: silicon area and critical signal delay time.

4.1 Memory Array Size

Cache size has a direct impact on hit ratio [Smith82], [Hill84]. The improvement in hit ratio is more pronounced in small cache memories such as the SPUR IUnit. Increasing the cache size in an on-chip memory is accomplished by simply expanding the instruction memory array. However, both the silicon area and speed of the cache are affected by changes in the memory array size.

The critical signal delay times for five array sizes are shown in Table 4-1. These speed estimates are based on SPICE simulations similar to the those discussed in section 3.4.

Memory Array Size	Critical Signal Delay Path			
	Select Line Delay (ns)	Data Line Delay (ns)	Miss Line Delay (ns)	Total Delay (ns)
Cache Size = 64 Instruction				
64 x 33 Array	4.2	4.4	2.7	11.3
Cache Size = 128 Instruction				
64 x 65 Array	4.9	4.8	2.6	12.4
128 x 33 Array (SPUR)	4.2	7.3	3.2	14.7
Cache Size = 256 Instruction				
128 x 65 Array	4.9	7.7	4.1	16.8
256 x 33 Array	4.2	13.0	5.0	21.2

Table 4-1: Critical Signal Delay vs Memory Array Size

Table 4-1 illustrates that the critical signal delay path in the IUnit is directly related to the memory array size and aspect ratio. This delay path includes the select line, data line, and miss line delays. The delay times in Table 4-1 show that the select delay time does not vary more than 17% even when select line capacitance is doubled. The decoders have a fixed internal delay time of 2.8 ns. The large output drivers in the decoders minimize the delay due to select line capacitance. Alternatively, the data line delay varies by a factor of three in Table 4-1. This delay time is due to the memory cell discharging the precharged data line. More elaborate sensing schemes could reduce this delay time for the slower alternatives. However, these sense amp designs would either require additional power, area or clock generation complexity depending on the approach used. The final delay component, the Miss line delay, is affected by the fall time of the data line. The memory array implemented in the SPUR IUnit is 2.3 ns slower than a square 64 x 65 memory array. However, if one instruction sub-blocks are used in the 64 x 66 array, then some of this performance improvement is lost. An additional pass gate would be necessary to select between the two miss signals. This would reduce the 2.3 ns advantage to 1.2 ns. Array size and sub-block combinations are summarized in section 4.4.

Increasing the number of instructions in the SPUR IUnit to 512 instructions is not practical with current technology using a static memory array. Research at U.C. Berkeley has investigated

using one-transistor dynamic memory cells for on-chip cache memories [Chil85]. A test structure was designed to evaluate the sensitivity of these memory cells in a standard 2 um logic process. Simulations showed that the sensitivity of the design could tolerate the process variations expected. Dynamic memory clocking circuits that can tolerate wide process variations have also been investigated [Tham85]. Dynamic memory implementations are beyond the scope of this report because significant changes in the IUnit control design would be necessary to support the refresh operations.

4.2 DataPath Width / Sub-block Size

The sub-block size in the SPUR instruction unit was constrained by the number of package pins that could be allocated to the data bus between the CPU and the external cache (32-bits). Preliminary trace driven simulations show that the number of data references from the EUnit may reach 30% [Hill86b]. This will impact the IUnits performance because prefetches are blocked by EUnit data references.

Several implementation alternatives could reduce the number of prefetches that are blocked. The sub-block size and CPU to external cache datapath could be increased to 64-bits. The instruction unit could then fetch and prefetch two instructions on each access to the external cache. Another alternative is to add a second memory port to the CPU. Prefetches would never be blocked in this case. This alternative is not considered further because the present IUnit implementation would work with a second external cache memory port. It would add a significant amount of complexity to the external cache implementation.

Cache Size = 128 Instructions; Sub-Block = 2 Instructions:

This implementation alternative would change both the instruction array and tag array dimensions. The instruction array would be organized as 64 sub-blocks with two instructions and one valid bit per sub-block (64 x 65). The number of blocks in the tag array would drop from 16 to 8. Several different designs could be used to select one of the two instructions in the sub-blocks. This problem is simplied in the SPUR IUnit because a multiplexer (InsMux) and control

signals are already available in the instruction buffer. Both instructions could be read from the instruction array and latched (InsReg) during phi2. The InsMux could simply be expanded to a five-way multiplexer to perform the final selection during phi3. The Instruction_To_InsBus signal could be combined with the lowest order address bit to determine which instruction is transferred to the Ins_bus. Timing during phi3 is not critical so there is no penalty from using a five-way multiplexer. This multiplexor may not be included in future IUnit implementations. Simulated delay times for two-way, four-way, and eight-way multiplexors are listed below:

Two-way multiplexor:
Data to Data Delay = 1.0 ns
Signal to Data Delay = 1.0 ns

Four-way multiplexor:
Data to Data Delay = 1.2 ns
Signal to Data Delay = 1.1 ns

Eight-way multiplexor:
Data to Data Delay = 1.4 ns
Signal to Data Delay = 1.3 ns

An implementation with 128 instructions and 2 sub-blocks would differ from the present IUnit in the following ways.

Area: The area of the IUnit would decrease by 8%. This is due to a 50% reduction in the tag array and sub-block valid bits.

Speed: The critical signal delay path in this instruction array is identical to the the results shown in Table 4-1 for a 64 x 65 memory array. The delay time for this alternative is 16% lower than the present IUnit implementation (From 14.7 ns to 12.4 ns). The critical delay time through the tag comparison block only decreases by 4% (From 9.3 ns to 8.9 ns) despite the fact that the data line capacitance is reduced by 50%. However, the data line delay is a small component of the total tag comparison delay time (see section 3.5).

Cache Size = 256 Instructions; Sub-Block = 2 Instructions:

Doubling the cache size with 2 instruction per sub-block would give an instruction array with dimensions 128 x 65. The following changes in area and speed would occur:

Area: The area would increase by 1.7 times. This IUnit would measure 6000 lambda wide by 7700 lambda high. Assuming the rest of the CPU layout remained the same, an IUnit this size could be incorporated in a technology with a scaling factor of lambda = 0.7 μm.

Speed: The critical signal delay path in the IUnit would increase by 14% (From 14.7 ns to 16.8 ns). This is caused by the select line capacitance doubling. The tag comparison signal delay path is identical to the SPUR IUnit in this example (16 blocks).

4.3 Two-way Associativity

Associativity is an important parameter in cache design because it forces design tradeoffs between hit ratio, cost, and performance [Smith82]. Trace driven simulations have shown that hit ratios do not improve significantly for cache designs with associativity greater that four or eight [Smith82], [Good83]. Simulations for small cache memories indicate that most of the benefit in hit ratio is obtained with a two-way associative cache [Hill86b].

This section presents several two-way associative cache implementations for the SPUR IUnit. The two options that I examine are: 128 vs 256 instructions. In both cases, I do not include area estimates for the replacement algorithm implementation. Simple random replacement algorithms could be implemented in less that 2% of the total IUnit area. A least-recently-used algorithm would affect the total IUnit area by approximately 4% to 8%. Neither of these replacement algorithms would impact the critical signal delay path in the IUnit.

Cache Size=128 Instructions; Associativity=2

Several features of the SPUR IUnit implementation are useful for a two-way associative implementation. For example, instructions are read from the IUnit during phi2 but are not sent to the EUnit until the following phase. The four-way instruction multiplexer in the SPUR IUnit is also useful in a two-way associative design. It can simply be expanded to a five-way multiplexor to support selection between the two instruction paths.

A block diagram showing a simple two-way associative implementation is shown in Figure 4-1. The instruction arrays are identical to the 64 x 33 array described in section 4.1. Miss signals are generated from each of these arrays, Instruction1_Miss and Instruction2_Miss. Block miss signals are generated by the two tag comparison blocks. The Block1_Miss and Block2_Miss signals shown in Figure 4-1 is analogous to the Block_Miss signal in the IUnit. All four miss signals would be latched by the control block at the end of phi2. The IUnit control would be complicated slightly to accommodate these new signals. Two additional static latches and static gates are necessary. The output logic must add an extra output signal for the InsMux. The following changes in area and speed result in this two-way associative design.

Area: The area of this two-way associative IUnit implementation would be about 17% larger than the SPUR IUnit. The area for the two instruction arrays is 12% larger than a single array. The area for two tag arrays that contain 8 blocks is 50% larger than one single array that contains 16 blocks.

Speed: In general, two-way associative caches are slower than direct mapped caches. Delay is added because instructions must pass through a multiplexer to select one of two instructions. This multiplexor must be clocked by one of the miss signals : Instruction1_Miss or Instruction2_Miss. Therefore, a series time delay (miss delay + multiplexor) is present in a typical two-way associative cache that is not present in a direct mapped cache. This series delay path is already present in the SPUR IUnit, therefore, a two-way associative implementation will not be inherently slower. The critical speed delay path in the two-way associative IUnit is the same as a direct mapped implementation except that one additional signal delay occurs in the control block. Despite this additional gate delay, the critical signal delay path in the two-way associative design is 9% faster than the present IUnit (From 14.7 ns to 12.9 ns). The data line capacitance in each memory array is reduced by 50% which more than offsets the additional gate delay. The delay time introduced by using a five-way multiplexor is not significant. The IUnit has an entire phase to transfer the instruction through the multiplexor onto the Ins_bus. The

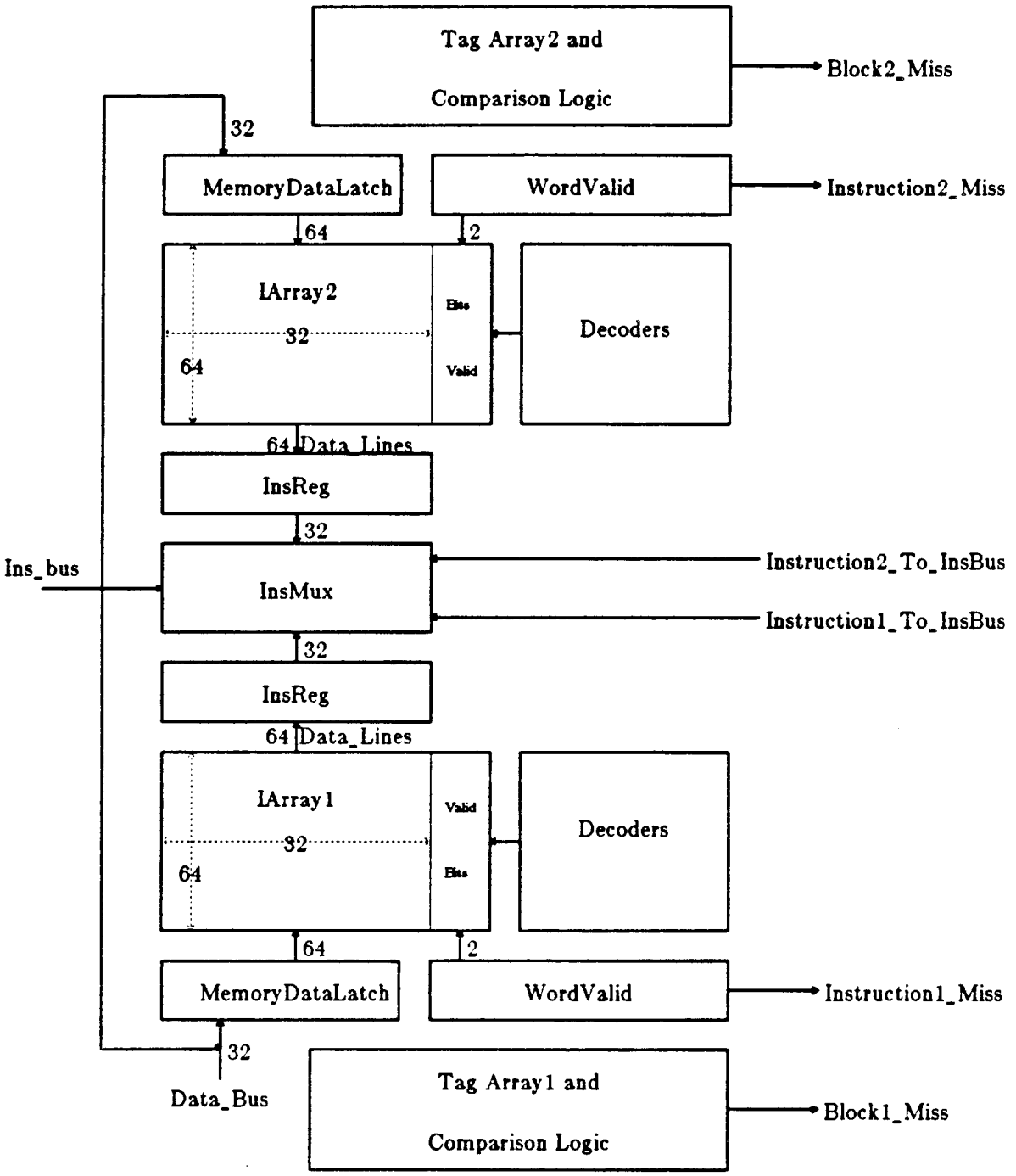


Figure 4-1:Two-Way Associative IUnit Implementation

present delay would increase by about 25% (From 7.5 ns to 9.3 ns). However, this time is significantly faster than the critical signal delay path.

Cache Size= 256; Associativity=2

The same design discussed in the previous example would also support a cache with twice the number of instructions. The following changes in area and speed would result:

Area: Two memory arrays with dimensions 64 x 66 dimensions would provide the best implementation. The Tag arrays are identical in size to the IUnit. The total area of this implementation would be 2.0 times larger than the present IUnit.

Speed: The critical signal delay of this two-way implementation is equivalent to the present IUnit. The key parameter is the data line capacitance. Using two separate memory arrays for the associative implementation reduces the data line capacitance by 50%. This configuration has two series delays: a pass gate to select one miss signal from each memory array and the gate delay in the IUnit control.

4.4 Alternative Implementation Summary

IUnit implementation alternatives are summarized in Table 4-2. All of the implementations with 128 instructions were faster than the SPUR IUnit implementation. In all cases, the speed improvement was due to reduced data line capacitance. Either the aspect ratio of the memory array changed or the memory array was split in half. The speed estimates in Table 4-2 indicate that either an IUnit with two instruction sub-blocks or a two-way associative implementation would have been fast enough for the SPUR CPU.

The difference in area between the two-way associative alternative and the other two option was more pronounced. In fact, it would not have been possible to fit the two-way associative implementation in a 2 μm version of the CPU. It would fit onto the 1.6 μm version of the CPU.

IUnit Configuration			Normalized Implementation Metrics	
Number of Sub-Blocks	Degree of Associativity	Number of Memory Arrays	Silicon Area	Critical Signal Delay
Array Dimension 64 x 33				
1	1	1	0.62	0.77
1	1	2	1.08	0.91
2 (2)	1	2	1.05	0.77
1	2	2	1.17	0.91
Array Dimension = 64 x 65				
1 (1)	1	1	0.97	0.91
2 (2)	1	1	0.92	0.84
1 (1)	1	2	1.8	1.01
2 (2)	1	2	1.7	0.95
1 (1)	2	2	2.1	1.01
Array Dimension = 128 x 33				
1 (SPUR)	1	1	1.0	1.0
Array Dimension = 128 x 65				
1 (1)	1	1	1.8	1.25
2 (2)	1	1	1.7	1.14
Array Dimension = 256 x 33				
1	1	1	1.8	1.44

(1) These implementations would require an array (n x 66) to support the two sub-block valid bits.

(2) This area estimate does not include the extra pad and bus area.

Table 4-2: Implementation Metric Summary

5. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

I believe that the density of integrated circuit technology will continue to advance into the next decade. Therefore, many of the design tradeoffs that we are now facing will change. For example, I think that microprocessors will routinely have both on-chip data and instruction caches. The size of these on-chip caches will continue to be an important tradeoff.

I have formed several opinions about the design of on-chip cache memorys during this project. On-chip cache memory designs must be optimized for a synchronous environment. Unlike board level cache designs, improving speed in an on-chip instruction cache does not necessarily improve the performance of the entire chip. A better approach is to design the on-chip cache to be slightly faster than the critical signal delay path in the entire chip. Architectural features that improve the hit ratio of the cache should then be considered. Any additional control complexity must also be taken into consideration.

Control design continues to be the most time consuming and error prone part of IC design. I think the IUnit control design could have been simplified with several custom software tools. For example, the individual cells in the output logic mosaics could have been placed automatically. More importantly, a software tool could analyze the logic equations and move non-critical equations into the PLA. This tool must understand timing relationships in order to partition the logic for either the PLA or the domino output logic. Creating these tools was beyond the scope of this report, but I believe that this would be an interesting research topic.

In conclusion, I implemented an on-chip instruction cache for the SPUR CPU. The instruction unit contains approximately 40 unique circuits and layouts. A variety of circuit design techniques were used from simple static gates to dynamic logic gates and latches. The simulated critical signal delay time of the instruction unit is less than 15 ns (typical). The layout efficiency of the instruction memory array was approximately 62% which is similar to industry standard static memory chips. The design and layout took approximately 4 man-months (40 hour weeks).

ACKNOWLEDGEMENTS

The SPUR project was well established when I joined the group one year ago. Members of the research team were busy molding their architectural ideas into a final design. Despite the fast paced nature of the project, I found the group members sincerely interested in sharing their ideas with me. I would particularly like to thank the CPU design team: Shing Kong, Dave Lee, and Wook Koh. They have provided invaluable technical guidance during my participation on this project. John Keller deserves credit for introducing me to the SPUR project and providing encouragement throughout the year. I also want to thank Mark Hill for contributing to this research paper with both ideas and manuscript reviews.

My research advisors, David Hodges and Randy Katz, have helped direct both my course work and involvement on the SPUR CPU. Without their guidance, and extremely quick readings of this report, I would not have finished my MSEE program in one year.

I would also like to acknowledge Hewlett Packard for supporting me through the Resident Fellowship Program and the management at the Northwest IC Division. The moral support of Bob Tillman, Skip Rung, Sam Angelos and many others helped me strive for this goal.

Finally, my strength (and sanity) throughout the entire year has been preserved by my wife, Alesia. She has helped smooth out some of the discouraging times and enhance the good times. Most importantly, she gave birth to our son Scott this year, who brings me a smile on every day.

REFERENCES

[Chil85] Brian Childers: "On-Chip Memory For Microprocessors", MS Report, Department of Electrical Engineering & Computer Science, University of California, Berkeley, 26 pages, Jan. 1985.

[Good83] James Goodman: "Using Cache Memory to Reduce Processor- Memory-Traffic", Proc. Tenth International Symposium on Computer Architecture, pp 124-131, June 1983.

[Hill82] Mark Hill, Dimitris Lioupis, Chris Nyberg, Tim Sippel: "RISC Cache Project", Graduate class project for CS292x, Computer Science Division, Department of Electrical Engineering & Computer Science, University of California, Berkeley, 17 pages, June 1982.

[Hill84] M.D. Hill: "Experimental Evaluation of On-Chip Microprocessor Cache Memories", Proc. Eleventh International Symposium on Computer Architecture, June 1983.

[Hill85] M.D. Hill et. el.: "SPUR: A VLSI Multiprocessor Workstation", Computer Science Division, Department of Electrical Engineering & Computer Science, University of California, Berkeley, 1985.

[Hill86a] M.D. Hill: Informal implementation notes on SPUR Instruction Unit, U.C. Berkeley, 1986.

[Hill86b] M.D. Hill: private communication, U.C. Berkeley, 1986.

[Jeong86] D.K. Jeong: Developed PLA templates compatible with MPLA, U.C. Berkeley, 1986.

[Kong86a] S. Kong, R. Duncombe, D. Lee, W. Koh: "The SPUR CPU: An Architectural Description", Version 2.0, Computer Science Division, Department of Electrical Engineering & Computer Science, University of California, Berkeley, 43 pages, June 1986.

[Kong86b] S. Kong: N.2 Description of the SPUR CPU, U.C. Berkeley, 1986.

[Mac84] D. MacGregor, D. Mothersole, and B. Moyer: "The Motorola MC68020", IEEE Micro, August 1984, pp. 101-118.

[Mac85] D. MacGregor, Jon Rubenstein: "A Performance Analysis of MC68020-based Systems, IEEE Micro, pp. 50-70, December 1985.

[Patt83] D.A. Patterson, P. Garrison, M.D. Hill, et al: "Architecture of a VLSI Instruction Cache for a RISC", Proc. Tenth International Symposium on Computer Architecture, pp 108-116, June 1983.

[Phil85] D. Phillips: "The Z80000 Microprocessor", IEEE Micro, pp. 23-36, December 1985.

[Scott86] W. Scott, R. Mayo, G Hamachi, J. 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.

[Smith82] A.J. Smith: "Cache Memories", Computing Surveys, pp 473-530, Sept. 1982.

[Tham85] K.S. Tham: "A Self-Timed One Transistor Dynamic Ram", MS Report, Department of Electrical Engineering & Computer Science, University of California, Berkeley, 26 pages, Jan. 1985.

APPENDIX A: LAYOUT PLOTS

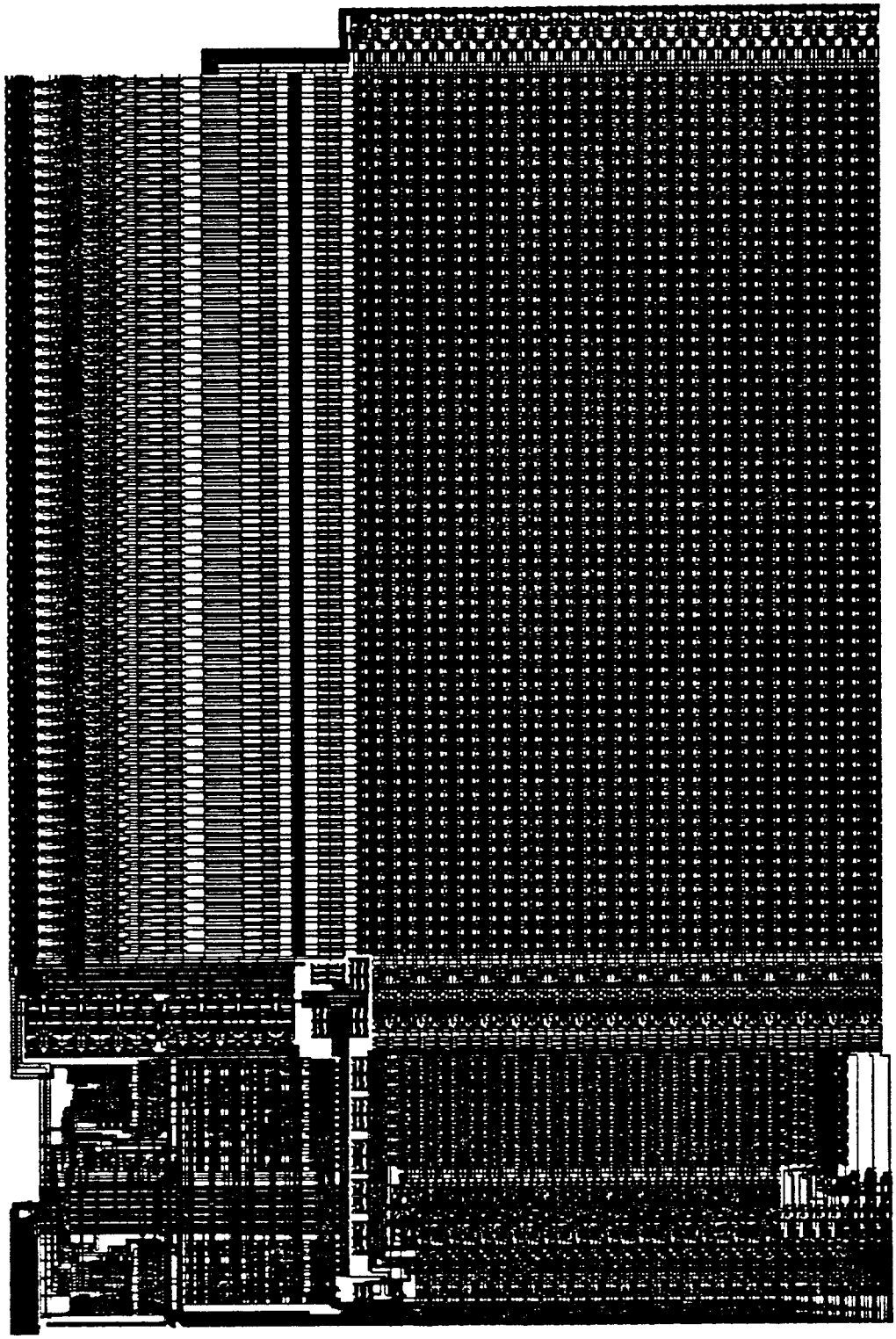


Figure A-1:Instruction Unit Layout Plot

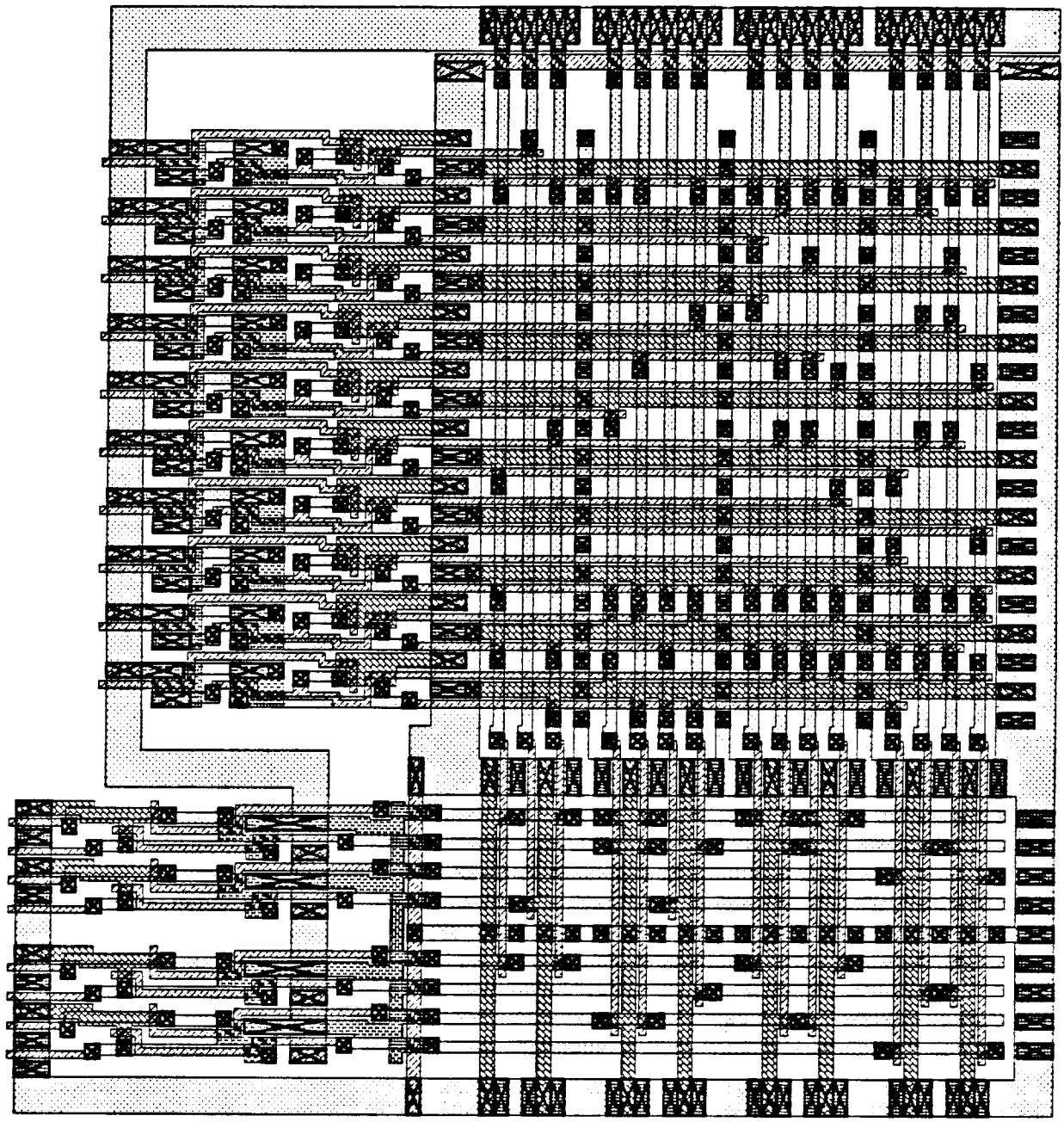


Figure A-2:Fetch State PLA Layout Plot

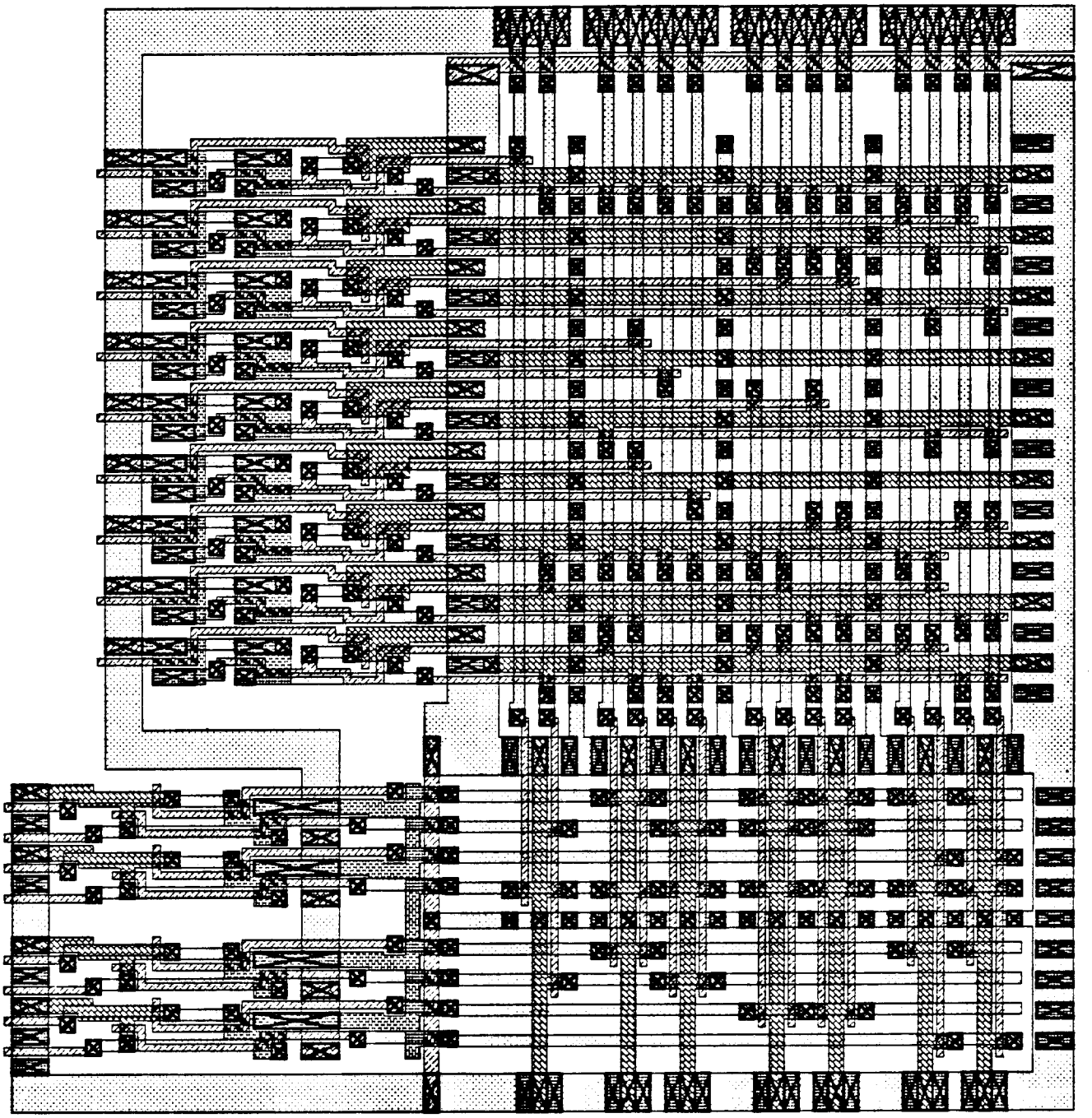


Figure A-3:Prefetch State PLA Layout Plot

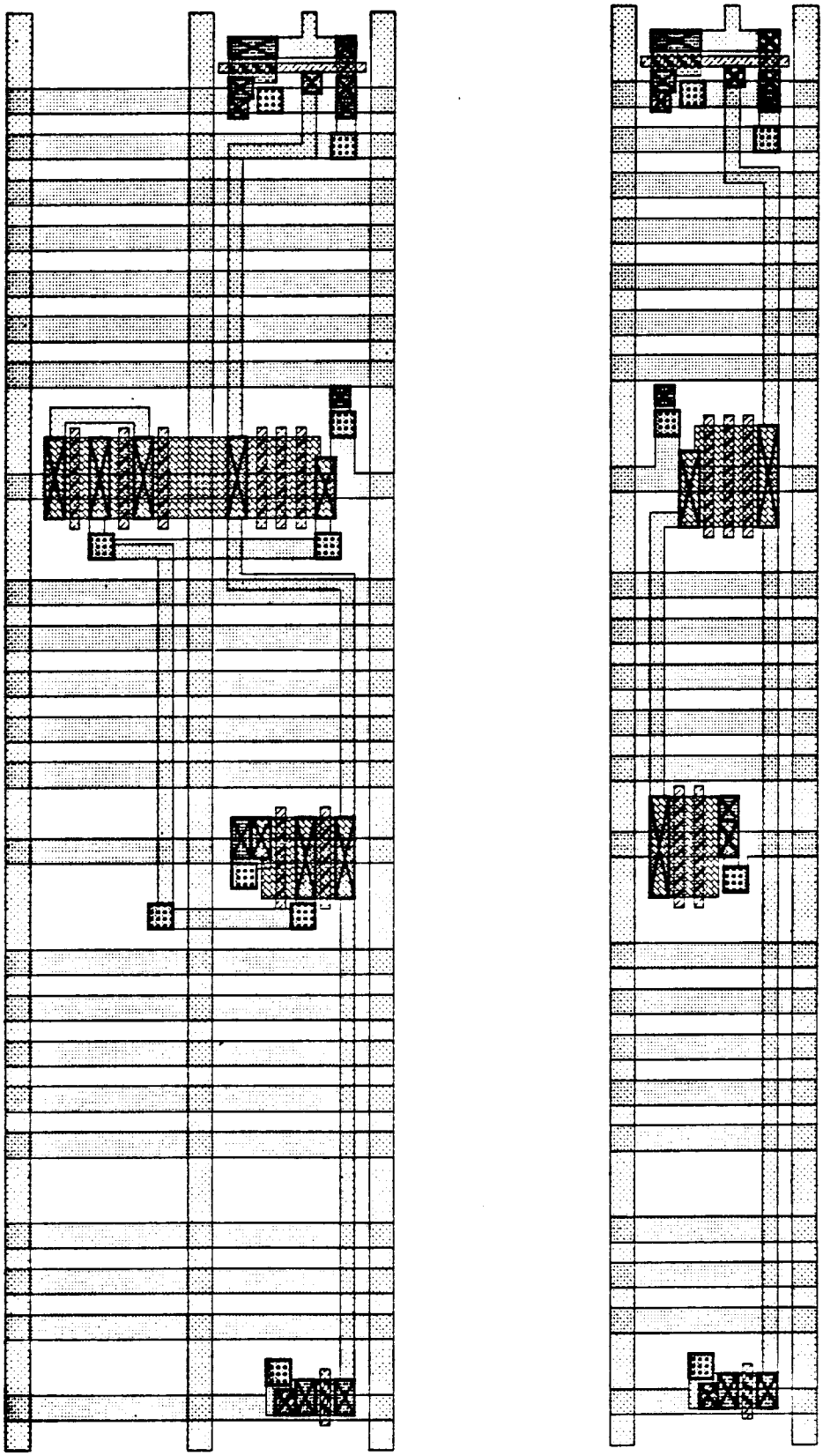


Figure A-4:Domino Output Logic Gate Layout Plot

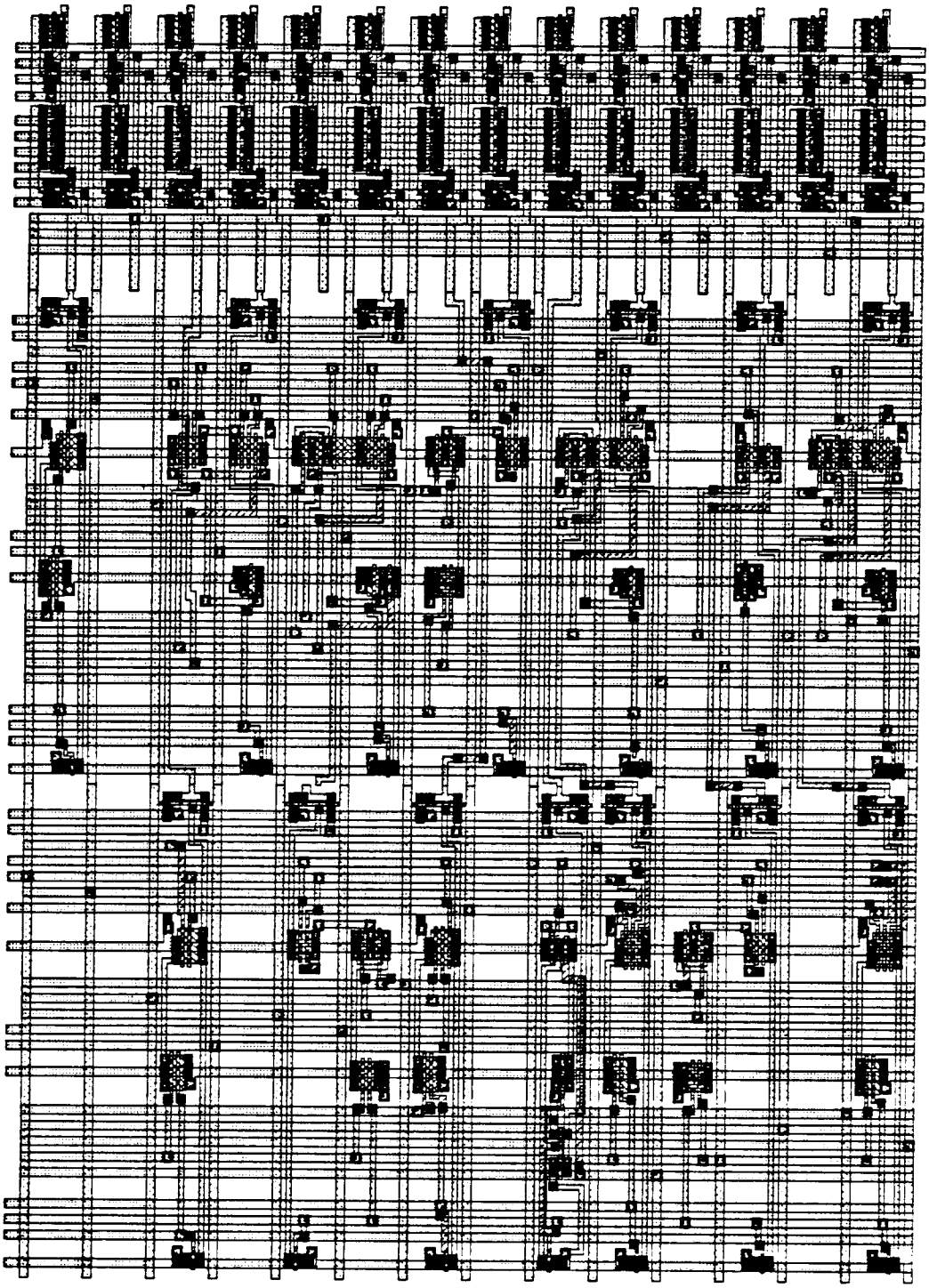


Figure A-5:Fetch Output Logic Layout Plot

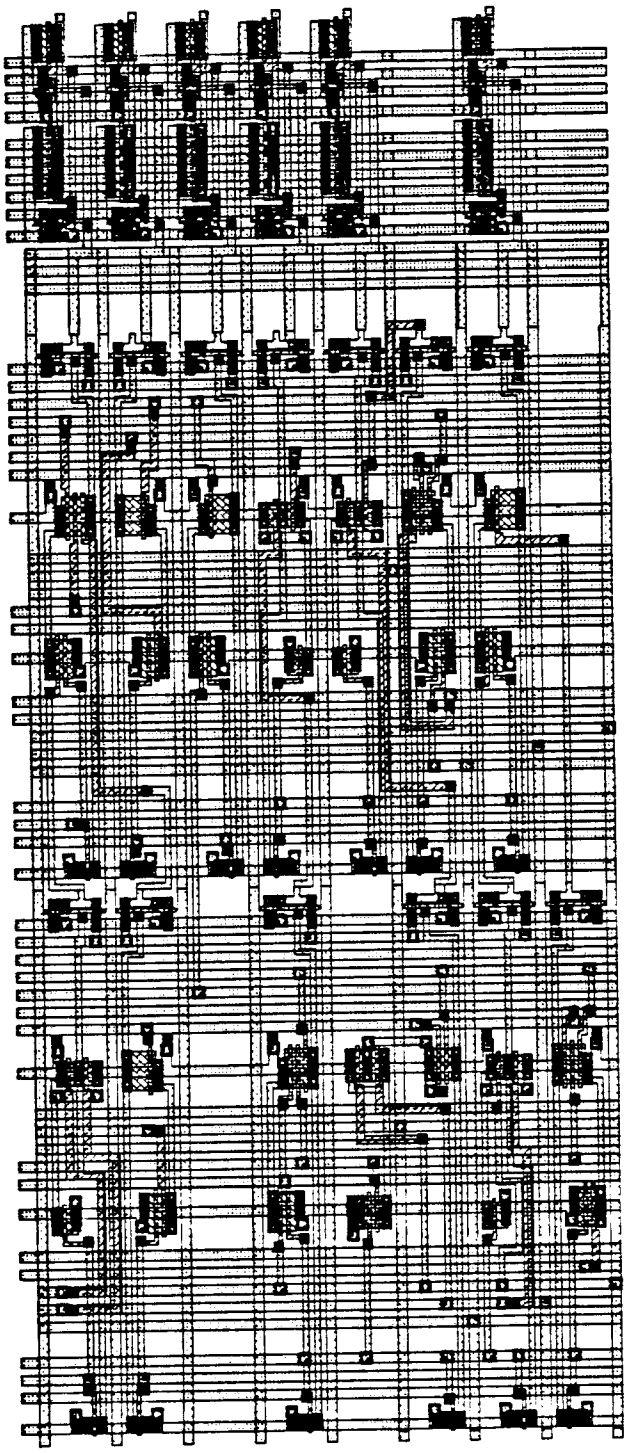


Figure A-6:Prefetch Output Logic Layout Plot

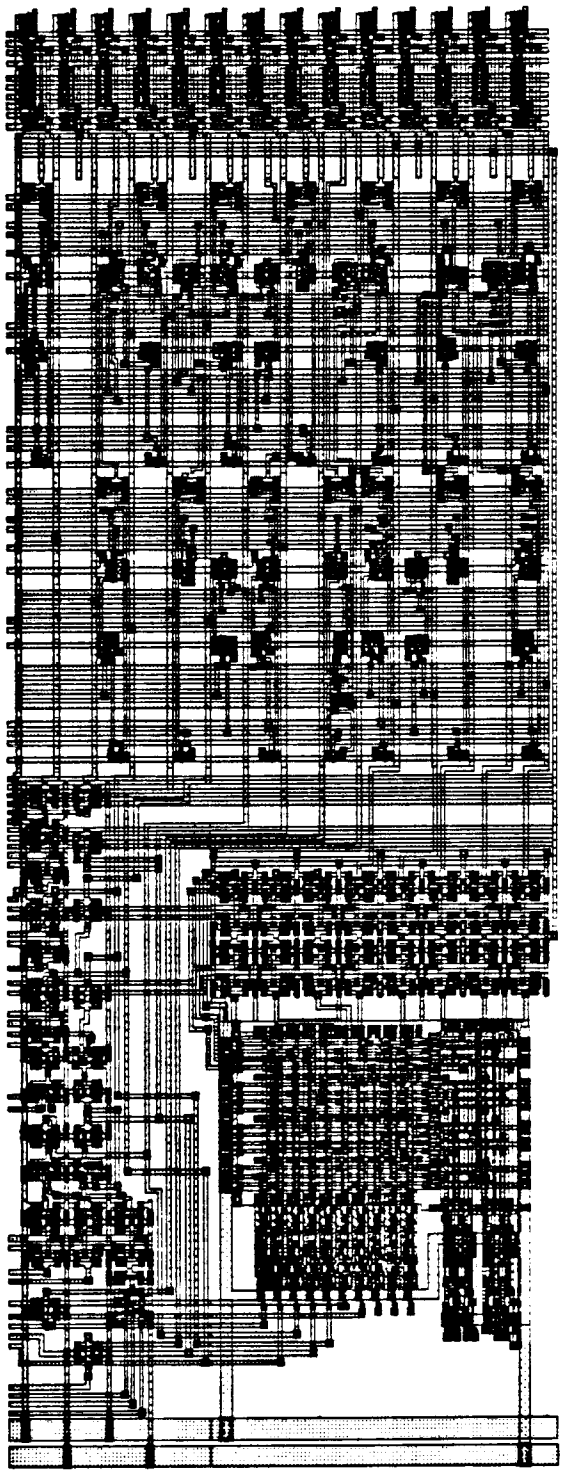


Figure A-7:Fetch Control Layout Plot

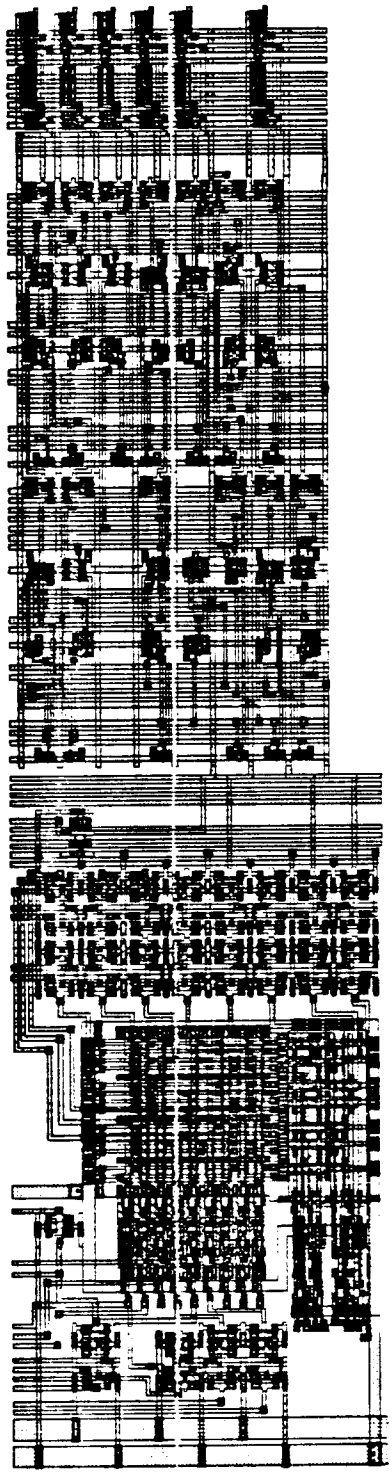


Figure A-8:Prefetch Control Layout Plot

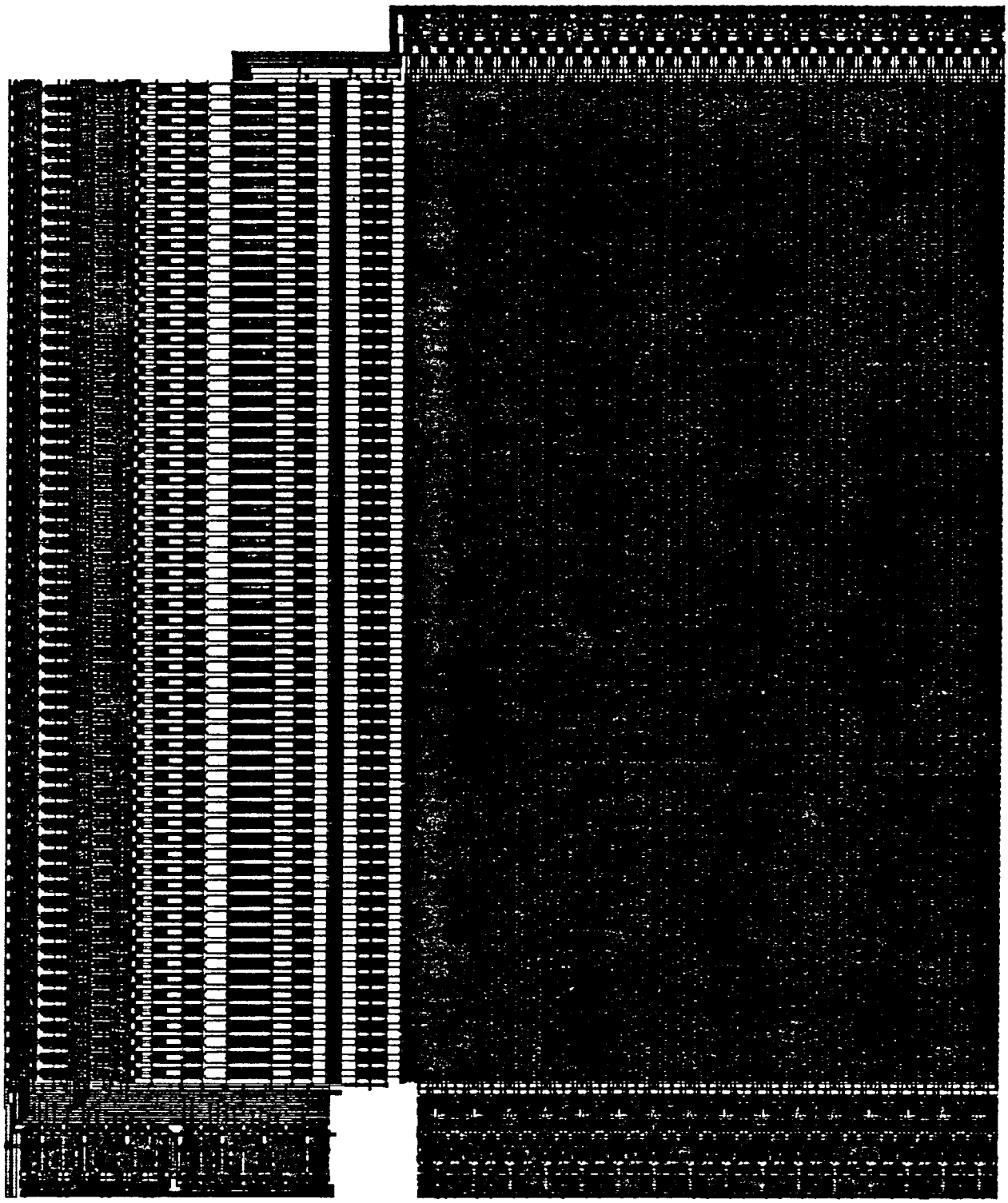


Figure A-9:Instruction Buffer Layout Plot

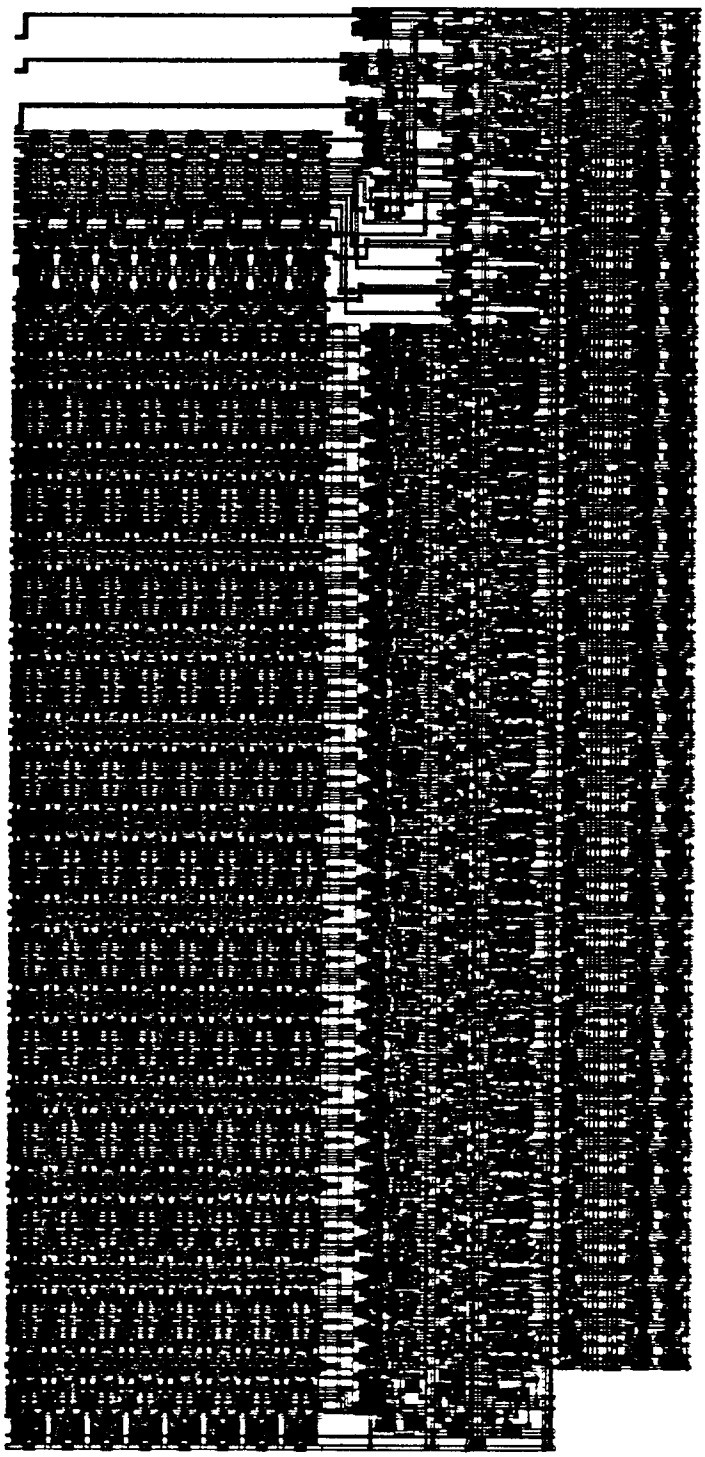


Figure A-10:Tag Comparison and Prefetcher Layout Plot

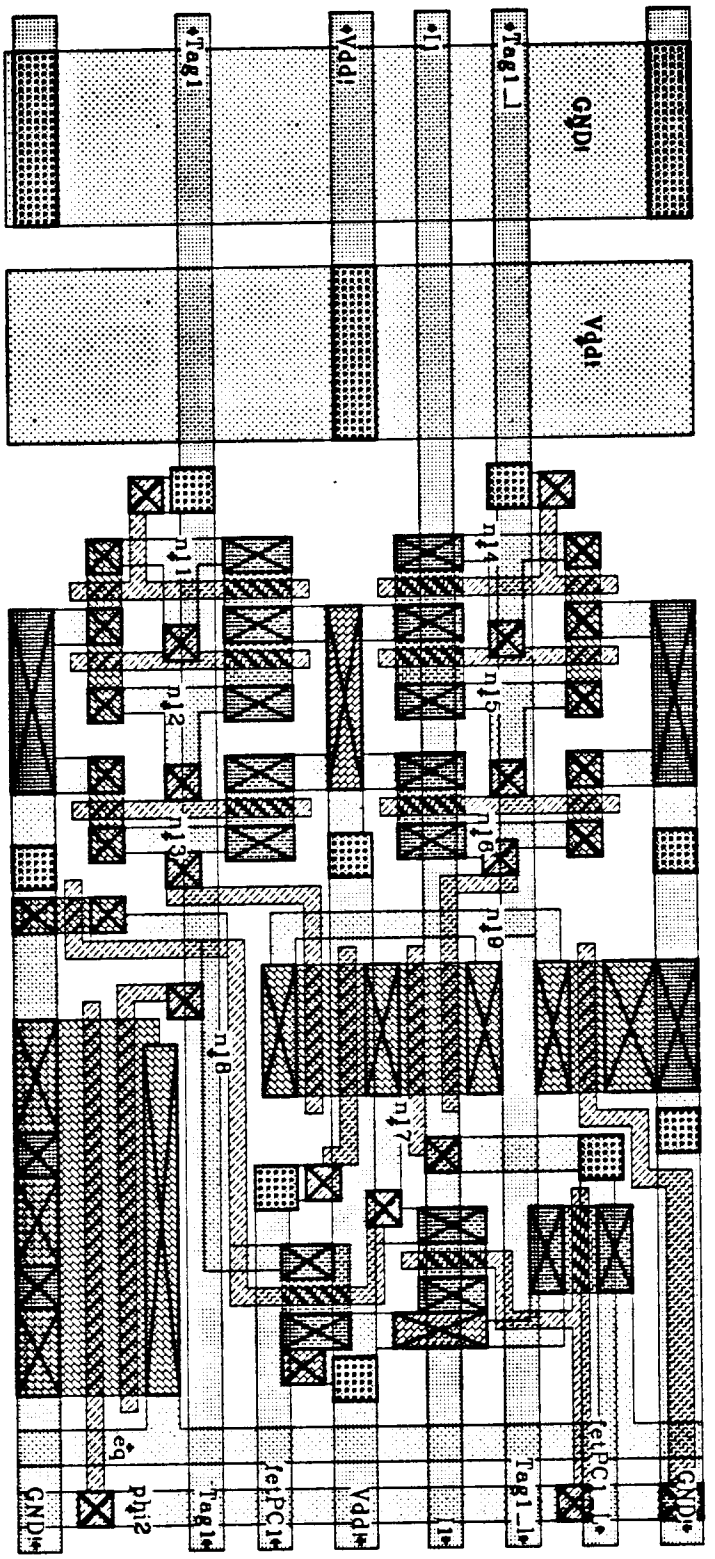


Figure A-11:Comparator Layout Plot

APPENDIX B: PLA DESCRIPTIONS

FETCH PLA ALGORITHM:

```
--7 INPUTS / 5 OUTPUTS (not including 3 state inputs and outputs)

INPUTS:  resetIB_C2 flush_VC2 ibMiss_C2 memBusy_VC3 LatchDataValid_C3
         LatchNotSpd_C1 LatchIuEn_C1;

OUTPUTS: resetIB normal membusy mempend disabled;

--Always reset to fet_reset when resetIB_C2 is asserted.

RESET ON resetIB_C2 TO fet_reset (resetIB);

--Describe state transitions.

fet_reset:      IF NOT resetIB_C2 THEN fet_normal (normal)
                ELSE LOOP (resetIB);

fet_normal:     CASE (flush_VC2 ibMiss_C2 memBusy_VC3 LatchNotSpd_C1 )
                  1?11 => fet_membusy (membusy);
                  0111 => fet_membusy (membusy);
                  1?01 => fet_mempend (mempend);
                  0101 => fet_mempend (mempend);
                ENDCASE => LOOP (normal);

fet_membusy:    IF NOT memBusy_VC3 THEN fet_mempend (mempend)
                ELSE LOOP (membusy);

fet_mempend:    CASE (LatchDataValid_C3 LatchIuEn_C1)
                  10 => fet_disable (disabled);
                  11 => fet_normal (normal);
                ENDCASE =>LOOP (mempend);

fet_disable :   IF LatchNotSpd_C1 THEN fet_normal (normal)
                ELSE LOOP (disabled);
```

PREFETCH PLA ALGORITHM:

```
--6 INPUTS / 5 OUTPUTS (not including 3 state inputs and outputs)

INPUTS:   resetIB_C2 flush_VC2 memBusy_VC3 LatchIuEn_C1
          LatchStartingPF_C3 LatchIuPre_C1;

OUTPUTS:  resetPF idle disabled waiting prefetch;

--Always reset to pf_reset when resetIB_C2 is asserted.

RESET ON resetIB_C2 TO pf_reset (resetPF);

--Describe state transitions.

pf_reset:   CASE (resetIB_C2 LatchIuEn_C1 LatchIuPre_C1)
                1?? => LOOP (resetPF);
                011 => pf_idle (idle);
            ENDCASE => pf_disabled (disabled);

pf_idle:    CASE (resetIB_C2 flush_VC2 LatchStartingPF_C3)
                1?? => pf_reset (resetPF);
                01? => LOOP (idle);
                001 => pf_waiting (waiting);
            ENDCASE => LOOP (idle);

pf_disabled:    IF resetIB_C2 THEN pf_reset (resetPF)
            ELSE LOOP (disabled);

pf_waiting: CASE (resetIB_C2 flush_VC2 LatchStartingPF_C3 memBusy_VC3)
                1??? => pf_reset (resetPF);
                01?? => pf_idle (idle);
                001? => LOOP (waiting);
                0001 => LOOP (waiting);
            ENDCASE => pf_prefetch (prefetch);

pf_prefetch:    CASE (resetIB_C2 flush_VC2 LatchStartingPF_C3 memBusy_VC3)
                1??? => pf_reset (resetPF);
                01?? => pf_idle (idle);
                001? => pf_waiting (waiting);
                0001 => pf_waiting (waiting);
            ENDCASE => LOOP (prefetch);
```

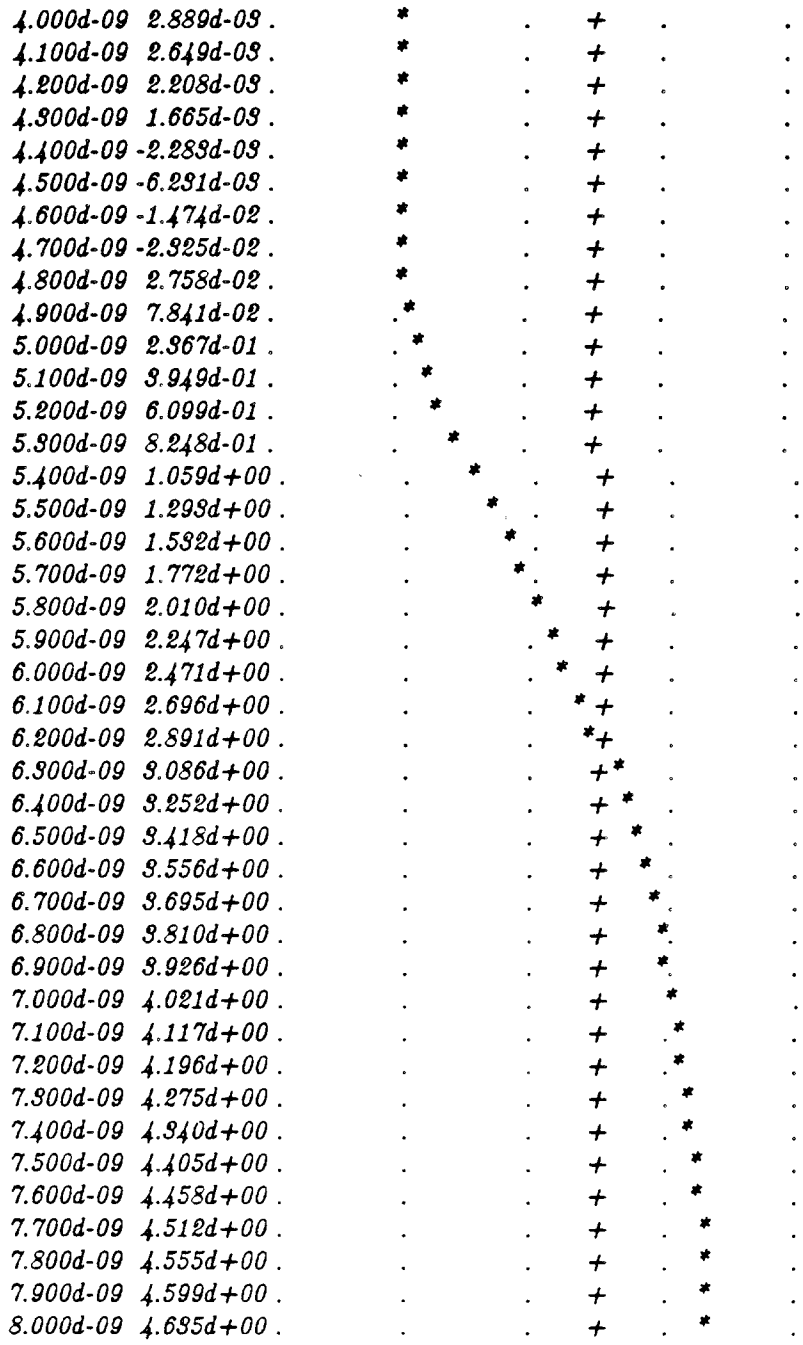
APPENDIX C: SPICE SIMULATION DATA

SPICE PARAMETER FILE

```
**********************************************************************
*       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
*
August+ 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
*
August+ 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
*
* Use NBIG and PBIG models for W >= 4U and L >= 2U
*
August+ NSUB=4E16 TPG=+1 XJ=.25U LD=.20U UEXP=.02 VMAX=5.5E4 JS=1000U
+ CGSO=220P CGDO=220P CJ=230U CJSW=260P CGBO=400P
*
August+ NSUB=2.0E16 TPG=-1 XJ=.20U LD=.05U UEXP=.05 VMAX=9.0E4 JS=1000U
+ CGSO=220P CGDO=220P CJ=670U CJSW=215P CGBO=400P
*
* Use NMIN and PMIN models for W <= 4U and L <= 2U
*
August+ NSUB=4E16 TPG=+1 XJ=.25U LD=.20U UEXP=.18 VMAX=5.5E4 JS=1000U
+ CGSO=220P CGDO=220P CJ=230U CJSW=260P CGBO=400P
*
August+ 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
*
**********************************************************************
```

DOMINO LOGIG GATE: MISS_To_InsBus Signal

```
1********08/06/86 ******** SPICE 2G.6   9/15/89 ********02:09:41*****

0DOMINO CRITICAL SPEED PATH (VDD=5, TYP PARAMS)

0****   TRANSIENT ANALYSIS              TEMPERATURE =   27.000 DEG C

0******************************************************************************


0LEGEND:

 *: MISS_To_InsBus
 +: phi3
X
    TIME     V(21)

(*)---------- -2.000d+00    0.  d+00    2.000d+00    4.000d+00  6.000d+00
                 ----------------------------

(+)---------- 0.  d+00    2.000d+00    4.000d+00    6.000d+00  8.000d+00
                 ----------------------------
 1.000d-09 -2.081d-05 +          *          .          .          .
 1.100d-09 -1.664d-05 +          *          .          .          .
 1.200d-09 -1.251d-05 +          *          .          .          .
 1.300d-09 -8.624d-06 +          *          .          .          .
 1.400d-09 -5.047d-06 +          *          .          .          .
 1.500d-09 -1.640d-06 +          *          .          .          .
 1.600d-09  1.552d-06 +          *          .          .          .
 1.700d-09  4.642d-06 +          *          .          .          .
 1.800d-09  7.603d-06 +          *          .          .          .
 1.900d-09  1.096d-05 +          *          .          .          .
 2.000d-09  1.480d-05 +          *          .          .          .
 2.100d-09  5.671d-04 . +        *          .          .          .
 2.200d-09  9.149d-04 .   +      *          .          .          .
 2.300d-09  1.213d-03 .    +     *          .          .          .
 2.400d-09  1.313d-03 .     +    *          .          .          .
 2.500d-09  1.414d-03 .      +   *          .          .          .
 2.600d-09  1.462d-03 .       + *          .          .          .
 2.700d-09  1.509d-03 .       + *          .          .          .
 2.800d-09  1.538d-03 .         X          .          .          .
 2.900d-09  1.567d-03 .         * +        .          .          .
 3.000d-09  1.601d-03 .         *  +       .          .          .
 3.100d-09  1.635d-03 .         *   +      .          .          .
 3.200d-09  1.701d-03 .         *    +     .          .          .
 3.300d-09  1.766d-03 .         *     +    .          .          .
 3.400d-09  1.856d-03 .         *      +   .          .          .
 3.500d-09  1.946d-03 .         *       +  .          .          .
 3.600d-09  2.114d-03 .         *        + .          .          .
 3.700d-09  2.281d-03 .         *         .+          .          .
 3.800d-09  2.491d-03 .         *         .  +        .          .
 3.900d-09  2.700d-03 .         *         .   +       .          .
```


4.000d-09 2.889d-03
4.100d-09 2.649d-03
4.200d-09 2.208d-03
4.300d-09 1.665d-03
4.400d-09 -2.283d-03
4.500d-09 -6.231d-03
4.600d-09 -1.474d-02
4.700d-09 -2.325d-02
4.800d-09 2.758d-02
4.900d-09 7.841d-02
5.000d-09 2.367d-01
5.100d-09 3.949d-01
5.200d-09 6.099d-01
5.300d-09 8.248d-01
5.400d-09 1.059d+00
5.500d-09 1.298d+00
5.600d-09 1.532d+00
5.700d-09 1.772d+00
5.800d-09 2.010d+00
5.900d-09 2.247d+00
6.000d-09 2.471d+00
6.100d-09 2.696d+00
6.200d-09 2.891d+00
6.300d-09 3.086d+00
6.400d-09 3.252d+00
6.500d-09 3.418d+00
6.600d-09 3.556d+00
6.700d-09 3.695d+00
6.800d-09 3.810d+00
6.900d-09 3.926d+00
7.000d-09 4.021d+00
7.100d-09 4.117d+00
7.200d-09 4.196d+00
7.300d-09 4.275d+00
7.400d-09 4.340d+00
7.500d-09 4.405d+00
7.600d-09 4.458d+00
7.700d-09 4.512d+00
7.800d-09 4.555d+00
7.900d-09 4.599d+00
8.000d-09 4.635d+00

IBUFFER CRITICAL SIGNAL DELAY PATH:

1********08/08/86 ******** SPICE 2G.6 3/15/83 ********04:48:46******

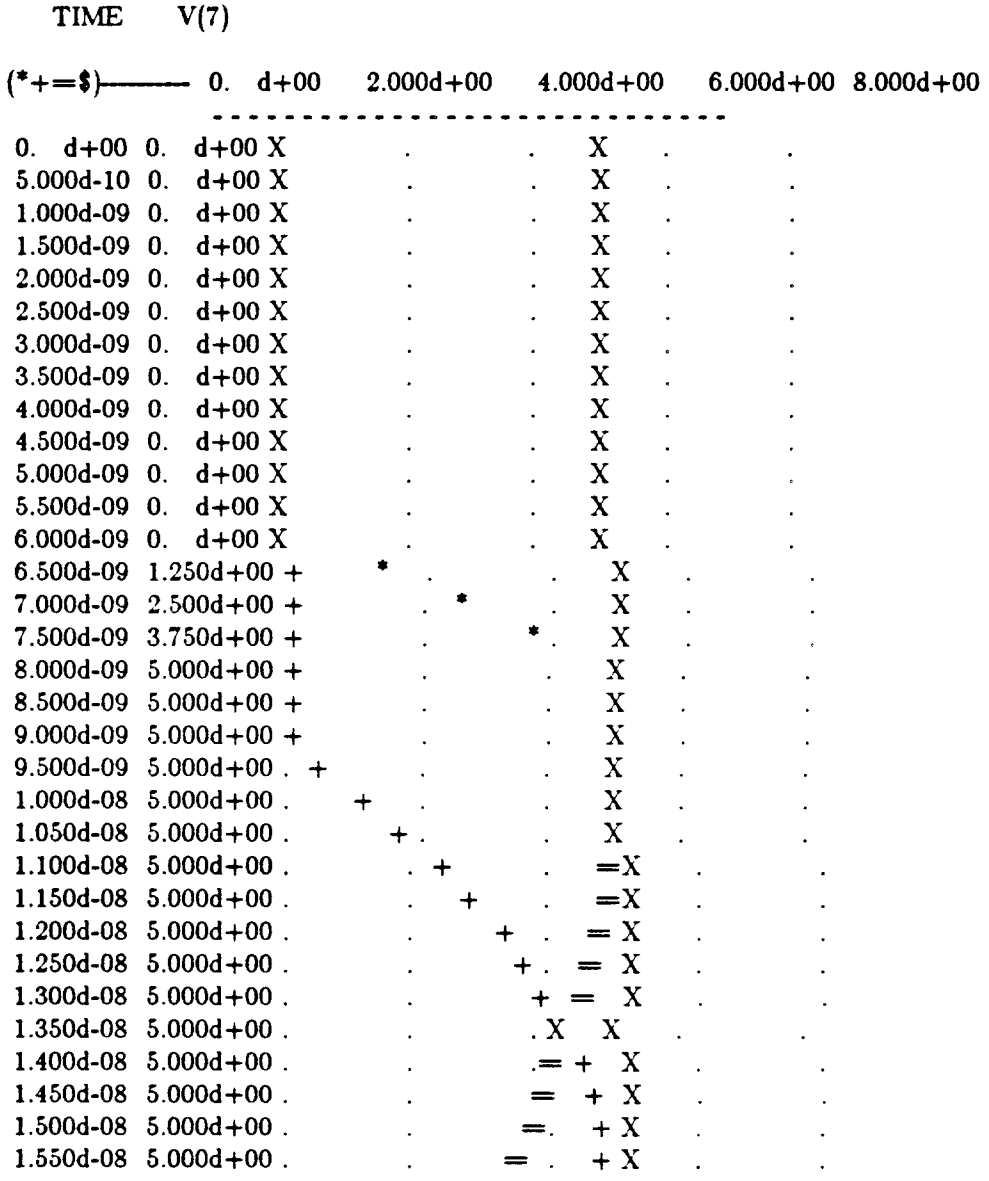
0IBUFFER CRITICAL SPEED PATH (VDD=5, TYP PARAMS)

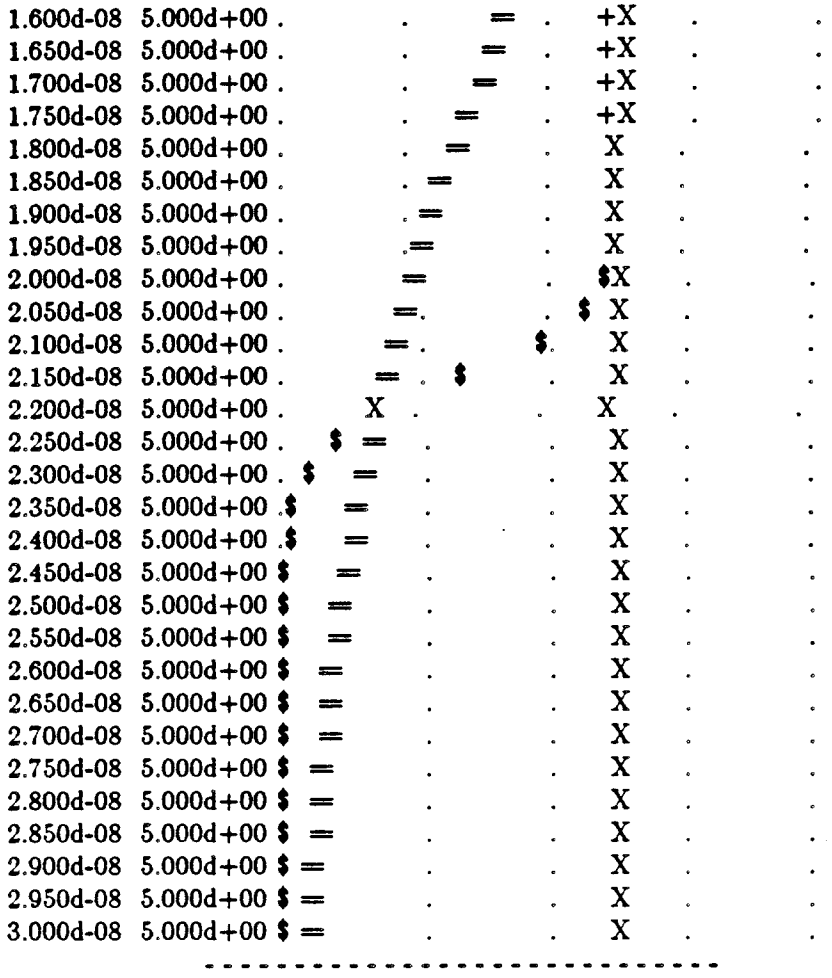
0**** TRANSIENT ANALYSIS TEMPERATURE = 27.000 DEG C

0**********

0LEGEND:

*: phi2phi4
+: Select Line
=: Data Line
$: Instruction_Miss Line
X




1.600d-08 5.000d+00
1.650d-08 5.000d+00
1.700d-08 5.000d+00
1.750d-08 5.000d+00
1.800d-08 5.000d+00
1.850d-08 5.000d+00
1.900d-08 5.000d+00
1.950d-08 5.000d+00
2.000d-08 5.000d+00
2.050d-08 5.000d+00
2.100d-08 5.000d+00
2.150d-08 5.000d+00
2.200d-08 5.000d+00
2.250d-08 5.000d+00
2.300d-08 5.000d+00
2.350d-08 5.000d+00
2.400d-08 5.000d+00
2.450d-08 5.000d+00
2.500d-08 5.000d+00
2.550d-08 5.000d+00
2.600d-08 5.000d+00
2.650d-08 5.000d+00
2.700d-08 5.000d+00
2.750d-08 5.000d+00
2.800d-08 5.000d+00
2.850d-08 5.000d+00
2.900d-08 5.000d+00
2.950d-08 5.000d+00
3.000d-08 5.000d+00

TAG COMPARISON CRITICAL SPEED PATH

*1********08/08/86 ******** SPICE 2G.6 3/15/83 ********05:05:47******

0TAG CRITICAL SPEED PATH (VDD=5, TYP PARAMS)

*0**** TRANSIENT ANALYSIS TEMPERATURE = 27.000 DEG C*

*0**********

0LEGEND:

**: phi2phi4*
+: Select Line
=: Data Line
$: Match Line
0: Block_Miss Line

```
(*$)---------- 0.  d+00    2.000d+00    4.000d+00    6.000d+00  8.000d+00
               ------------------------------

(+=0)---------- -2.000d+00    0.  d+00    2.000d+00    4.000d+00  6.000d+00
               ------------------------------
0.  d+00  0.  d+00 *        X        .    $    .    =    .
5.000d-10 0.  d+00 *        X        .    $    .    =    .
1.000d-09 0.  d+00 *        X        .    $    .    =    .
1.500d-09 0.  d+00 *        X        .    $    .    =    .
2.000d-09 0.  d+00 *        X        .    $    .    =    .
2.500d-09 0.  d+00 *        X        .    $    .    =    .
3.000d-09 0.  d+00 *        X        .    $    .    =    .
3.500d-09 0.  d+00 *        X        .    $    .    =    .
4.000d-09 0.  d+00 *        X        .    $    .    =    .
4.500d-09 0.  d+00 *        X        .    $    .    =    .
5.000d-09 0.  d+00 *        X        .    $    .    =    .
5.500d-09 0.  d+00 *        X        .    $    .    =    .
6.000d-09 0.  d+00 *        X        .    $    .    =    .
6.500d-09 1.250d+00 .     * X        .    $    .    =    .
7.000d-09 2.500d+00 .       X  *     .    $    .    =    .
7.500d-09 3.750d+00 .       X     *  .    $    .    =    .
8.000d-09 5.000d+00 .       X        .    X    .    =    .
8.500d-09 5.000d+00 .       0  +     .    X    .    =    .
9.000d-09 5.000d+00 .       0     +  .    X    .    =    .
9.500d-09 5.000d+00 .       0        .  + X    .    =    .
1.000d-08 5.000d+00 .       0        .    X +  .  =         .
1.050d-08 5.000d+00 .       0        .    X    =+.         .
1.100d-08 5.000d+00 .       0        .    X    .+          .
1.150d-08 5.000d+00 .       0        .  =  X    .  +        .
1.200d-08 5.000d+00 .       0        =  .  X    .    +      .
1.250d-08 5.000d+00 .       0     =     .    X    .    +    .
1.300d-08 5.000d+00 .       0  =        .    X    .    +    .
1.350d-08 5.000d+00 .       0 =         .  $ *    .    +    .
1.400d-08 5.000d+00 .       0 =      $  .    *    .    +    .
1.450d-08 5.000d+00 .       0=$         .    *    .    +    .
```

```
1.500d-08  5.000d+00 .      $  0=        .     *     .     +     .
1.550d-08  5.000d+00 .   $     =0        .     *     .     +     .
1.600d-08  5.000d+00 .  $      =      0  .     *     .     +     .
1.650d-08  5.000d+00 . $       =         .     *  0  .     +     .
1.700d-08  5.000d+00 .$        =         .     *     .  0  +     .
1.750d-08  5.000d+00 .$        =         .     *     .     X     .
1.800d-08  5.000d+00 $         =         .     *     .     X     .
1.850d-08  5.000d+00 $         =         .     *     .     X     .
1.900d-08  5.000d+00 $         =         .     *     .     X     .
1.950d-08  5.000d+00 $         =         .     *     .     X     .
2.000d-08  5.000d+00 $         =         .     *     .     X     .
                    ----------------------------------------------
```