

**VLSI DESIGN OF A TWIN REGISTER FILE FOR
REDUCING THE EFFECTS OF CONDITIONAL
BRANCHES IN A PIPELINED
ARCHITECTURE**

By

MANISH SHAH

Bachelor of Engineering

Maharaja Sayajirao University

Baroda, India

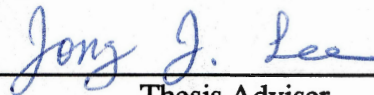
1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1992

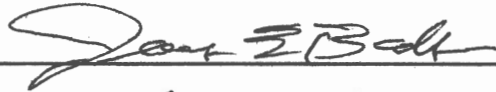
Thesis
1992
S525v

VLSI DESIGN OF A TWIN REGISTER FILE FOR
REDUCING THE EFFECTS OF CONDITIONAL
BRANCHES IN A PIPELINED
ARCHITECTURE

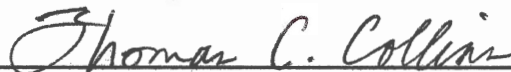
Thesis Approved:



Thesis Adviser







Dean of the Graduate College

PREFACE

Pipelining is a major organizational technique which has been used by computer engineers to enhance the performance of computers. Pipelining improves the performance of computer systems by exploiting the instruction level parallelism of a program. In a pipelined processor the execution of instructions is overlapped, and each instruction is executed in a different stage of the pipeline. Most pipelined architectures are based on a sequential model of program execution in which a program counter sequences through instructions one by one.

A fundamental disadvantage of pipelined processing is the loss incurred due to conditional branches. When a conditional branch instruction is encountered, more than one possible paths are following the instruction. The correct path can be known only upon the completion of the conditional branch instruction. The execution of the next instruction following a conditional branch cannot be started until the conditional branch instruction is resolved, resulting in stalling of the pipeline. One approach to avoid stalling is to predict the path to be executed and continue the execution of instructions along the predicted path. But in this case an incorrect prediction results in the execution of incorrect instructions. Hence the results of these incorrect instructions have to be purged. Also, the instructions in the various stages of the pipeline must be removed and the pipeline has to start fetching instructions from the correct path. Thus incorrect prediction involves a flushing of the pipeline.

This thesis proposes a twin processor architecture for reducing the effects of conditional branches. In such an architecture, both the paths following a conditional branch are executed simultaneously on two processors. When the conditional branch is resolved, the results of the incorrect path are discarded. Such an architecture requires a special

purpose twin register file. It is the purpose of this thesis to design a twin register file consisting of two register files which can be independently accessed by the two processors. Each of the register files also has the capability of being copied into the other, making the design of the twin register file a complicated issue.

The special purpose twin register file is designed using layout tools Lager and Magic. The twin register file consists of two three-port register files which are capable of executing the 'read', 'write' and 'transfer' operations. The transfer of data from one register file to another is accomplished in a single phase of the clock. The functionality of a 32-word-by-16-bit twin register file is verified by simulating it on IRSIM. The timing requirements for the read, write and transfer operations are determined by simulating the twin register file on SPICE.

I would like to express my sincere gratitude to my advisor, Dr. J. J. Lee, for the guidance and encouragement he provided throughout my graduate program. I would also like to express my deep appreciation to Dr. Louis Johnson for the technical assistance he provided during the course of this study. His suggestions were of invaluable help to me, as was the constant support and encouragement of Dr. James Baker, to whom special thanks are due.

I would also like to thank my parents for their understanding, support, and love; and my sisters and my brother for being a source of constant inspiration in my pursuit of excellence.

Finally, I am grateful to the faculty, staff, and students of the department of Electrical Engineering at Oklahoma State University for the friendship, guidance, and help they provided throughout my graduate studies.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. CONDITIONAL BRANCHES	6
Branch Penalty	6
Branch Cost	10
Conventional Branch Schemes	11
Branch Prediction	11
Branch Target Buffer	13
Delayed Branches	14
Forward Semantic	15
Bypassing Conditional Branches	15
III. TWIN PROCESSOR ARCHITECTURE	18
Twin Processor Machine	18
Branch Handling	20
Effect of Branches	23
IV. DESIGN OF A TWIN REGISTER FILE	26
Three-Port Memory Cell	26
Twin Register File	29
System Timing	29
Organization	32
Cell Placement	34
Twin Memory Cell	37
Transfer Operation	40
Read Operation	42
Write Operation	45
Address Decoder	48
Line Drivers	52
Read/Write Circuit	55
Layout of Twin Register File	59
V. SIMULATION RESULTS	61
IRSIM Results	61
SPICE Simulation Results	70
Wordline Delay	70

Chapter	Page
Write Delay	71
Precharge Delay	73
Read Delay	74
Transfer Delay	75
 V1. CONCLUSIONS	 78
 BIBLIOGRAPHY	 80
 APPENDIXES	 83
APPENDIX A - TIMLAGER C FILE FOR TWIN REGISTER FILE	84
APPENDIX B - SPICE FILE	88

LIST OF TABLES

Table	Page
I. Simulation Data for Register File A	62
II. Simulation Data for Register File B	63
III. Transfer Data for Register B	66
IV. Transfer Data for Register A	68

LIST OF FIGURES

Figure	Page
1. Space - Time Diagram for a Pipelined Machine	3
2. Space - Time Diagram for a Non - Pipelined Machine	3
3. Typical Processor Pipeline	7
4. Example of a Code Segment	8
5. Instruction Flow in a Pipeline	9
6. System Configuration of a Twin Processor Machine	19
7. Instruction Flow in a Twin Processor Machine	22
8. Pseudo Three-Port RAM	27
9. System Timing	30
10. Twin Register File Architecture	33
11. Cell Placement in a 32-Word-by-16-Bit Twin Register File	36
12. Twin Memory Cell	38
13. Layout of a Twin Memory Cell	39
14. Equivalent Circuit During Transfer Operation	41
15. Equivalent Circuit During Read Operation.....	43
16. Equivalent Circuit During Write Operation	47
17. A 2-to-4 Line Precharged NOR Decoder	50
18. Layout of a 2-to-4 Line Interleaved Decoder	51
19. Wordline and Transfer Line Driver	53
20. Layout of Wordline and Transfer Line Driver	54
21. Write Circuit	56

Figure	Page
22. Read Circuit	57
23. Layout of Read/Write Circuit	58
24. Layout of a 32-Word-by-16-Bit Twin Register File	60
25. Read/Write Operation of Twin Register File	64
26. Transfer Operation from Register File A to Register File B	67
27. Transfer Operation from Register File B to Register File A	69
28. Wordline Delay	71
29. Write Delay	72
30. Precharge Delay	73
31. Read Delay	75
32. Transferring 'high'	76
33. Transferring 'low'	77

CHAPTER I

INTRODUCTION

Computer performance has been steadily improved over the past two decades by exploiting advances in semiconductor processing technology and by making architectural innovations. This evolution has led to high-performance machines which execute more than one instruction simultaneously by exploiting the parallelism available at the instruction level of the program.

One major approach for incorporating a higher parallelism in computer systems has been the pipelined processing, which is defined as "the technique of decomposing a repeated sequential process into subprocesses, each of which can be executed efficiently on a special dedicated autonomous module that operates concurrently with others " [1]. The advantage of pipelining is that it provides a way of starting a new process before the old one has been completed. Therefore, in a pipelined computer the completion rate of instructions is not a function of the latency of each instruction, but rather of how soon an instruction can be initiated [2].

An example of pipelined processing is the execution of instructions in a typical computer. The execution process consists of five steps: fetching an instruction (IF), decoding it (ID), reading operands (OP RD), performing arithmetic or logical operations (EXE), and finally writing the results back (OP WR). In a non-pipelined machine these five steps must be completed before the next instruction can be issued, while in a pipelined machine the successive instructions can be executed in an overlapped fashion. Since the execution of the instruction is divided into five subprocesses and each subprocess is executed in five different stages of the pipeline, we can have five instructions executing in

parallel. This principle is illustrated in Figures 1 and 2, which show the execution of five consecutive instructions on a pipelined machine and on a non-pipelined machine, respectively. The pipelined architecture executes five instructions in nine cycles, while the non-pipelined architecture executes only two instructions in ten cycles.

A typical pipelined computer executes instructions in a sequential manner so that the address of the next instruction to be executed is obtained by incrementing the current program counter by the size of the current instruction. Thus each instruction does not have to explicitly specify the address of the next instruction to be executed. However, a branch instruction alters the sequential flow of instructions and explicitly specifies the address of the next instruction to be executed. A branch instruction may be a conditional branch or an unconditional branch. In case of an unconditional branch, the sequential flow of instructions is always altered and the processor has to fetch and execute the instructions from an explicitly specified target address.

In case of a conditional branch, the processor has to decide whether the branch will be taken or not. This decision is typically based on the results of a 'COMPARE' or 'TEST' operation and usually takes more than one cycle. So either the pipeline stalls until the condition is resolved, or it makes a prediction as to whether the branch will be taken or not and executes the appropriate instruction stream. If the prediction is incorrect, then the pipeline has to be flushed, the results of the incorrect prediction have to be purged, and a new instruction has to be fetched from the correct path. Thus branch instructions disrupt the sequential flow of instructions through the pipeline, leading to a significant delay in the execution of the programs in the pipelined computer.

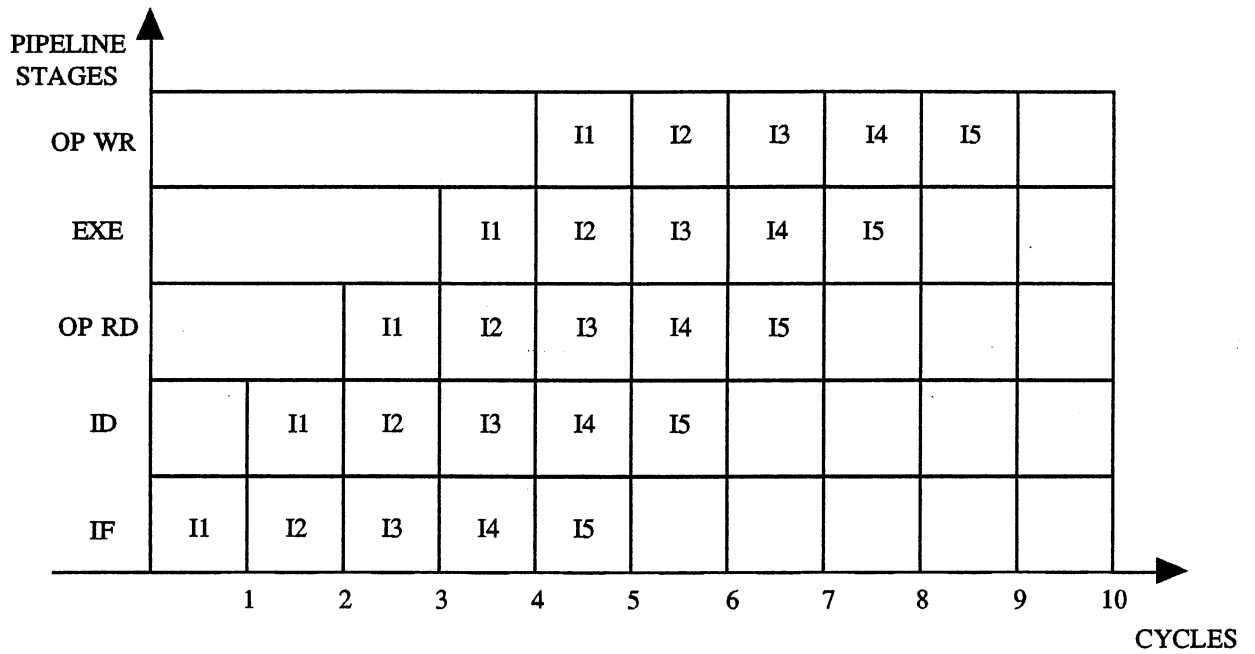


Figure 1. Space - Time Diagram for a Pipelined Machine

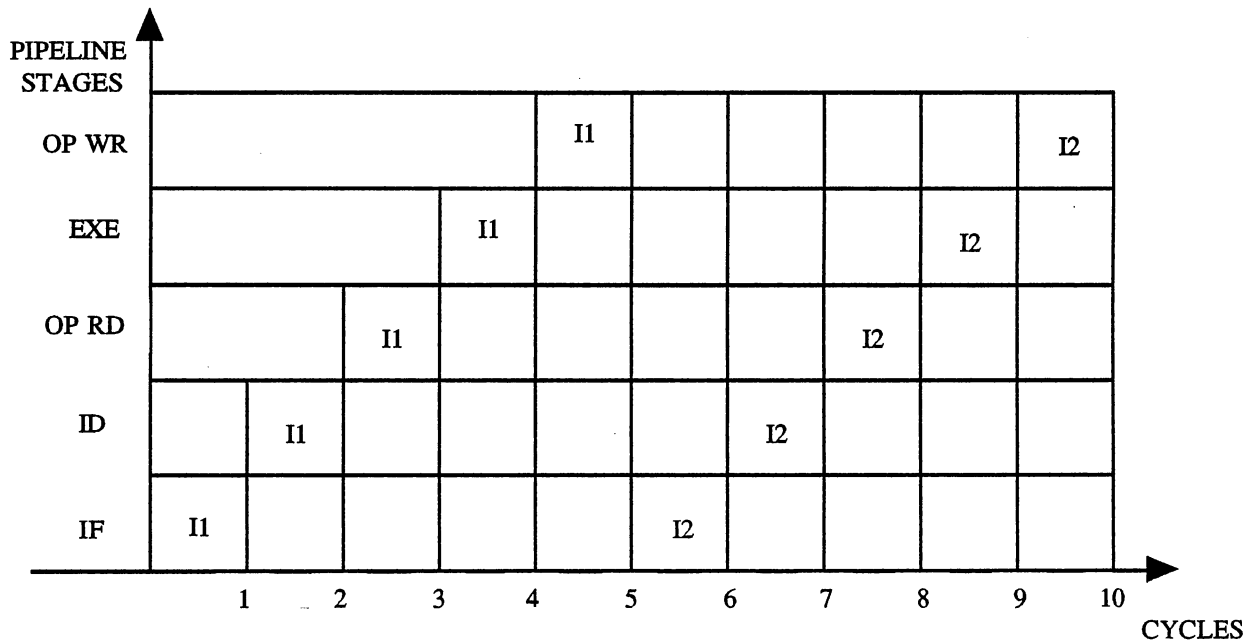


Figure 2. Space - Time Diagram for a Non - Pipelined Machine

This thesis presents the design of a special purpose twin register file intended to reduce the effects of conditional branches. When a computer has two processors connected to the twin register file, two possible instruction streams of a conditional branch can be processed simultaneously. When the condition is resolved, the results of the incorrect path are simply discarded. Some architectural aspects resembling this approach are incorporated in the IBM 3033 [12] and in the Pipeline Pushdown Computer proposed by Stone [13].

The twin register file consists of two register files, each of which can be independently accessed by two processors. When the two processors start executing two instruction streams of a conditional branch, their register files need to have the same copy of data to ensure that both processors use the same data. Hence, each register file has the capability of being copied into the other one. In other words, each register file can create a shadow of its contents in the other register file. This special purpose twin register file was designed using the layout tools Magic and Lager. The design of the register file was parameterized, and a 32-word-by-16-bit twin register file was laid out and verified by SPICE and IRSIM simulations.

The contents of this thesis are presented in six chapters. Chapter II describes the effects of conditional branches on a pipelined computer, analyzes the branch cost in terms of time, and discusses the major schemes for reducing the branch penalties, such as branch prediction, branch target buffer, delayed branching and forward semantic. The final section of Chapter II presents the advantages of bypassing conditional branches. Chapter III describes the novel twin processor architecture, discusses the handling of conditional branches by the twin processor, and derives the branch cost for the twin processor machine. Chapter IV describes the design of the special purpose twin register file which is used in the twin processor machine to process two paths of a conditional branch. The twin register file consists of two independent three-port register files, each of which has the capability of being copied into the other. Chapter V presents the simulation results of the twin register file. The functionality of a 32-word-by-16-bit twin register file is verified by

simulating it on IRSIM. The read, write and transfer latencies are determined by SPICE simulation of a single memory cell with the capacitances of a 32-word-by-16-bit register file placed on the critical control and data lines of the memory cell.

CHAPTER II

CONDITIONAL BRANCHES

Studies have indicated that branch instructions represent around 15 to 30% of the total number of instructions executed in a typical computer [5]. Branches alter the sequential flow of instructions, resulting in discontinuities in program execution. These discontinuities cause the machine to waste an average of 35% of the total execution time [3]. The discontinuities also complicate the design of instruction fetch units [15]. Even in superscalar processors the performance is limited due to branches, not due to hardware constraints [14]. In this chapter, the effects of conditional branches are discussed and the branch cost is analyzed. We study some conventional schemes which have been used for alleviating the detrimental effects of conditional branches, and discuss the technique of bypassing the conditional branches through simultaneous execution of two instruction streams following a conditional branch.

Branch Penalty

To analyze the effects of branch instructions on the performance of a pipelined processor, consider a hypothetical machine with the pipeline shown in Figure 3. Instructions are fetched by the instruction fetch unit (IF) and decoded in the decode unit (ID). Once the data dependencies have been resolved, operands are read from the registers (OP RD) and the instructions are forwarded to the execution unit for execution (EXE). Upon completion of the execution, the results are written back to the registers (OP WR).



Figure 3. Typical Processor Pipeline

In this hypothetical machine, each conditional branch instruction specifies three parameters: operands to be compared, branch condition, and branch destination. The result of the comparison, which is known at the end of the execution cycle, determines the path of subsequent execution. This differs from the scheme employed in some RISC machines, where the conditional branch instruction is split in two parts: a compare instruction and a jump instruction. The compare instruction sets the condition codes, and the branch instruction execution is based on these condition codes. In the hypothetical machine, we combine the compare and jump operations of a conditional branch instruction because this results in reduced hardware and software complexity [10]. Also, such a scheme simplifies the analysis for branch penalty.

The machine is assumed to have a very large branch target buffer so that the instructions from the branch target are available at the decode time of the branch instruction. The branch target buffer also provides the address of the target instruction. The machine is also assumed to have large data and instruction caches, so that for all practical purposes the memory access time goes to zero and a new instruction can be issued every cycle.

Figure 4 shows a segment of program code to be executed in the pipeline. Instructions k , $k+1$, and $k+2$ execute sequentially. Instruction $k+2$ is a branch instruction that can either jump to instruction m or fall through to instruction $k+3$. Figure 5 shows the flow of these instructions through the pipeline. Instruction k enters the pipeline during cycle 1. The processor produces no results for the first four cycles. This unproductive time

is called the start-up latency. The first instruction completes at the end of cycle 5, and then one instruction is expected to complete every cycle in the pipeline. As seen in Figure 5, instructions k and $k+1$ complete during cycle 5 and cycle 6, respectively. The branch instruction $k+2$ enters the pipeline during cycle 3 and is decoded in cycle 4. Once the conditional branch instruction is detected, the processor stalls until the condition is resolved. This stalling introduces three bubbles in the pipeline. Instruction $k+3$, which was fetched during cycle 4, has to be flushed from the pipeline. During cycle 6, the branch instruction $k+2$ enters the execution unit and the branch condition is resolved at the end of this cycle. Hence, the processor fetches the target instruction m in cycle 7. This instruction is completed during cycle 11. No instructions, therefore, are completed during cycles 8 through 10. These wasted cycles constitute the branch penalty.

k
 $k+1$
 $k+2$ (Branch to m)
 $k+3$
 \cdot
 \cdot
 m
 $m+1$
 \cdot
 \cdot

Figure 4. Example of a Code Segment

CYCLE	PIPELINE SEGMENT				
	IF	ID	OP RD	EXE	OP WR
1	k				
2	k+1	k			
3	k+2(branch)	k+1	k		
4	k+3	k+2	k+1	k	
5	○	○	k+2	k+1	k
6	○	○	○	k+2	k+1
7	m	○	○	○	k+2
8	m+1	m	○	○	○
9		m+1	m	○	○
10			m+1	m	○
11				m+1	m
12					m+1

Figure 5. Instruction Flow in a Pipeline

The penalty will be greater in machines which have longer pipelines, such as machines having floating point units. In these machines, the execution stage will take more than one cycle, depending on the number of pipeline stages in the floating point unit. Due to the longer execution latency, the resolution of the condition will be delayed and the pipeline will have to stall for a greater number of cycles. Thus the effect of conditional branches becomes more severe as the length of the pipeline increases.

Branch Cost

The branch penalty described in the previous section affects the performance of the hypothetical processor. In this research, we used a simple model similar to the one considered by Lilja [3], in order to analyze the effects of branches. In the model we considered the following key parameters:

P_b	=	probability that a particular instruction is a branch
s	=	number of stages in the pipeline
b	=	branch penalty in number of cycles wasted
T_{ave}	=	average number of cycles between the completion of two successive instructions.

The branch penalty b , which is the number of cycles the processor has to stall when a branch is encountered, depends on the number of stages in the pipeline. Typically, $b = s - 2$.

If there are no branches in an instruction stream, one instruction is issued every cycle and the time between the completion of two successive instructions is one cycle. But when a branch instruction is encountered, the pipeline has to stall for b cycles and so the time between the completion of two instructions is $1 + b$. Hence, the average time between completion of two instructions is

$$\begin{aligned} T_{ave} &= (1 - P_b) * (1) + P_b * (1 + b) \\ &= 1 + b * P_b. \end{aligned} \tag{2.1}$$

The above equation shows that the maximum performance occurs when the branch penalty is zero, which results in one instruction being completed every cycle. Let F_b be defined as the average number of instructions completed every cycle. Then

$$F_b = \frac{1}{T_{ave}}$$

$$= \frac{1}{1 + bP_b} \quad (2.2)$$

A previous study has shown that P_b ranges from 0.1 to 0.3 [3]. If we consider a five-stage pipeline having a branch penalty b of three cycles and $P_b = 0.3$, then $T_{ave} = 1.9$ and $F_b = 0.53$. On the other hand, if the pipeline had nine stages with a branch penalty of seven cycles, then $T_{ave} = 3.1$ and $F_b = 0.32$.

Conventional Branch Schemes

The above analysis presents a worst case scenario where the pipeline is stalled as soon as the branch is decoded. In most machines, the branch penalties can be significantly reduced by employing some architectural schemes for reducing the penalties. Some of the most widely used schemes are discussed in this section.

Branch Prediction

Branch Prediction is one of the common approaches used to reduce the penalties caused due to conditional branches [8, 9]. Through use of prediction techniques, the probable execution path of a branch is determined before the condition is resolved, and the instructions are executed in a conditional mode. In this mode of execution, all the results produced in the pipeline are marked tentative. If the prediction is correct, there are no penalties involved. On the other hand, if the prediction is incorrect, then a repair mechanism is required in order to undo all the effects on the registers and main memory caused by the instructions executed from the incorrectly predicted path [17]. Once the processor restores all the tentative changes, it can continue fetching and issuing instructions along the correct branch path. The restoration process requires maintaining backups for the registers, which increases the hardware complexity.

There are two types of prediction schemes: static prediction and dynamic prediction. In case of static prediction, the processor makes a presumptive decision as to whether a

branch is likely to be taken or not, based on some static information. One of the simplest forms of static prediction is to predict that all branches are always taken (predict-branch-taken) or never taken (predict-branch-not-taken) [3]. An example of predict-branch-not-taken is a computer which continues fetching the instructions sequentially even after the branch is detected. Studies have indicated that more than 50% of the time branches are taken. Hence, if the cost of prefetching a sequential instruction is the same as that of fetching an instruction from the branch target, then always prefetching from a branch target address should give better performance. If the branch is not taken, then the instructions fetched from the target have to be flushed [5]. Processors suffer an extra penalty in flushing instructions from the pipeline. Studies by McFarling and Hennessy have indicated that predict-branch-taken is slightly slower than predict-branch-not-taken and is also more complex to implement [5].

Another form of static prediction is based on the opcode of the branch instruction [3]. In this case the opcode is used to predict the branch direction. The hardware assumes that some branches will be taken for certain branch opcodes. For example, consider a DO LOOP which has a conditional branch instruction at end of the loop. Since there is high probability of the branch being taken, the conditional branch is encoded so as to indicate to hardware that branch is always taken. This mechanism has resulted in a prediction accuracy of greater than 75% [3].

A more complicated form of static branch prediction involves the use of a single static branch prediction bit, which was used in the AT&T CRISP microprocessor [12]. If the bit is set, then the hardware prefetches the branch target instruction; otherwise, it prefetches the next sequential instruction. The compiler determines the setting of the bit depending on the use of the branch. Ditzer and McLellan [12] have reported accuracies of 74 to 94% using this scheme.

In dynamic prediction technique, the processor uses history information about branch instructions that have been executed in order to predict the outcome of the branch [3]. Such

an idea requires that the processor be able to monitor the execution behavior of a program. A mechanism called the branch history table is used for this purpose. The branch history table stores information about previously executed branches so that fairly accurate predictions regarding the branch outcomes can be made. The branch history table is usually a cache memory and can be accessed associatively. The branch history table is updated each time the execution of a branch is completed. The hardware cost of this mechanism will be influenced by the number of entries in the history table, the number of history bits maintained, and the set associativity of the table.

Another simple implementation of dynamic prediction technique uses a single history bit for prediction. This bit indicates whether a branch is to be taken or not. Such a history table is much smaller and does not provide the branch address. It is assumed that the branch address is available at instruction decode time. Lee and Smith [8] found that such a scheme yields an accuracy ranging from 80 to 96%.

Branch Target Buffer

In a typical pipelined processor, instructions are prefetched in a sequential order and are stored in an instruction buffer. When a branch occurs, the target instruction is typically not present in the instruction buffer. In order to reduce the time required to fetch the target instruction, a special cache memory called the branch target buffer (BTB) has been used [8]. Each entry in the BTB consists of the address of a branch instruction that has already been executed and the target instruction to which it branched. The BTB may also store the next few instructions after the branch target instruction.

When the pipeline decodes a branch instruction, it associatively searches the BTB for the address of that instruction. If it is in the BTB, the target instruction is supplied directly from the BTB to the pipeline and prefetching begins from the new path. If the instruction is not in the BTB, the pipeline is stalled while it fetches the new instruction stream to the instruction buffer. The processor then selects an entry in the BTB for replacement and

stores the new target instruction in the BTB. But it is to be noted that it takes a fairly large buffer to obtain a good hit rate [5]. Using a two-way, set-associative BTB with 16 entries and least-recently-used replacement policy, Lee and Smith found a hit rate of 27% [8]. A BTB with 256 entries resulted in a hit rate of 72%.

Delayed Branches

Another way of optimizing conditional branches is delayed branching, which has been effectively used in Reduced Instruction Set Computers [10]. In delayed branching, one or more instructions following the branch are always executed, regardless of whether the branch is taken or not [3,11]. The success of delayed branching depends on the ability of the compiler to schedule useful instructions in the slots following the branch [5]. The number of slots to be filled--the branch delay--depends on the number of stages in the pipeline.

For example, suppose an instruction i is a conditional branch to instruction j . A processor with branch delay of b will execute the instructions in the sequence $i, i+1, \dots, i+b$. If the branch is not taken, then instruction $i+b$ is followed by $i+b+1$, and the instructions executed are $i, i+1, \dots, i+b, i+b+1, \dots$. On the other hand, if the branch is taken, then instruction $i+b$ is followed by instruction j . The instructions executed are $i, i+1, \dots, i+b, j, \dots$. Thus b instructions after the branch ($i+1$ through $i+b$) are always executed, regardless of whether a branch is taken or not.

Typically, the branch delay in Berkeley RISC processors consists of only one cycle, and therefore the compiler is required to insert only one instruction following the branch. However, the effectiveness of delayed branching is reduced in machines with long pipelines, because the ability of the compiler to schedule useful instructions in the delay slots is limited.

The performance of delayed branching depends on several factors, such as architecture, implementation, compiler, and the application program. According to DeRosa

and Levy [11], delayed branching results in a 7 to 9% performance improvement compared to non-delayed branching. If the compiler is unable to fill the delay slots with useful instructions, then some instructions which do not perform any useful operation (NO-OP instructions) have to be inserted following the conditional branch instruction. Processor time and memory bandwidth are wasted in fetching and executing these NO-OP instructions.

Forward Semantic

This is a software approach for reducing the cost of branches, in which an optimizing compiler is used to predict the direction of all branches in a program [9]. The program is run once or several times for a representative input suit. Then the program is recompiled and the representative input suit is used to predict the branches. This prediction is stored in the opcode as a single bit indicator. When an instruction is determined to be a branch at the decode time, this bit is examined by the hardware. Depending on the value of the bit, the instruction fetch unit fetches the next sequential instruction or it fetches the target instruction. This scheme has lower branch cost compared to other hardware prediction schemes, but the compilation time is significantly increased.

Bypassing Conditional Branches

One way of avoiding the pipeline stall is to fetch both the instruction streams following a conditional branch, and start executing them simultaneously. Once the condition is resolved, the results of the incorrect path are discarded. This technique is called branch bypassing [3].

Riseman and Foster considered a hypothetical machine to study the effects of bypassing conditional branches [4]. The machine has an infinite number of registers and an unlimited memory so that the memory access time can be assumed to be zero. The machine has an unlimited instruction cache so that decode and dispatch times of instructions can be

ignored. In order to allow a large number of instructions to be executed in parallel, the machine is assumed to have an unlimited number of functional units. Furthermore, conditional branch instructions do not impede the flow of the program in this machine because the path to be taken from the branch point is assumed to be known beforehand; thus, the conditional branches are assumed to be bypassed. Under these conditions the programs run at a speed limited only by the execution times of the various instructions and any inherent sequential dependencies between them. Riseman and Foster found that such an ideal machine is able to execute programs 51 times faster than a typical conventional machine considered for their research [4].

Riseman and Foster then considered the same infinite machine with no conditional jumps bypassed. If the processor encounters a conditional branch, it has to stall until the condition is resolved. In that case they found an average speedup of only 1.72 times that of a conventional machine. This result again stresses the importance of handling conditional branches in pipelined machines. Riseman and Foster also considered the case when only one conditional branch is bypassed on the ideal machine. When a conditional branch is encountered, the machine executes both the paths of the branch until the condition is resolved. But if the machine encounters another branch before the previous branch is resolved, then the machine has to stall. Under these circumstances, they found an average speedup of 2.72 times that of a conventional machine. They also found that the relative speedup increases as a function of \sqrt{j} where j is the number of jumps bypassed. Thus the bypassing of a single conditional branch also results in a significant advantage.

In another study, Stone suggested a pipeline pushdown stack computer, in which multiple instruction streams are processed when a conditional branch occurs [13]. A salient feature of this machine is that the register references in the instruction stream are not explicit. An 'idle register queue' maintains the list of unassigned registers. A translator uses this list to perform the task of register allocation at run time. When a conditional branch is encountered, a second translator is brought into operation and both the translators proceed

concurrently along the two paths of the conditional branch. Dynamic allocation of registers ensures that two paths use a separate set of registers. Thus, by pursuing both the streams of a conditional branch until the correct stream is identified, the delays normally associated with a conditional branch are reduced. In another example, the IBM 3033 does prefetching and decoding of the two possible instruction paths of a conditional branch, but no execution beyond the conditional jump [12].

CHAPTER III

TWIN PROCESSOR ARCHITECTURE

This chapter presents the novel processor with a twin register file. The twin processor is used to bypass conditional branches, and we analyze the twin processor in terms of branch penalties involved.

Twin Processor Machine

The twin processor machine processes two instruction streams following a conditional branch. The simultaneous processing of two instruction streams is achieved by operating two identical processors in parallel. This is similar to the scheme proposed by Stone's pipeline push-down stack computer [13]. The two processors fetch and execute two instruction streams following a conditional branch, and, once the condition is resolved, the machine discards the results of the incorrect path. The processor which was executing the correct path continues executing the next sequential instruction.

The twin processor system consists of two identical processing units which share a common main storage but have two independent connected register files as shown in Figure 6. The machine employs a register-register architecture so that all sources and destinations of arithmetic/logic instructions are registers. The two register files are identical and each of them has two read ports and one write port to support execution of one instruction every clock cycle. When the two processors start executing two paths of a conditional branch, their local memories need to be identical. If this is not true, then the two processors will use different data, leading to incorrect results. Therefore, the contents of one register file should be copied into the other register file before the two processors start

executing two paths of a conditional branch. The copying of data from one file to another should be accomplished as quickly as possible, because the processing will be stalled until the two register files are identical. This requires a special purpose register file which is called the twin register file.

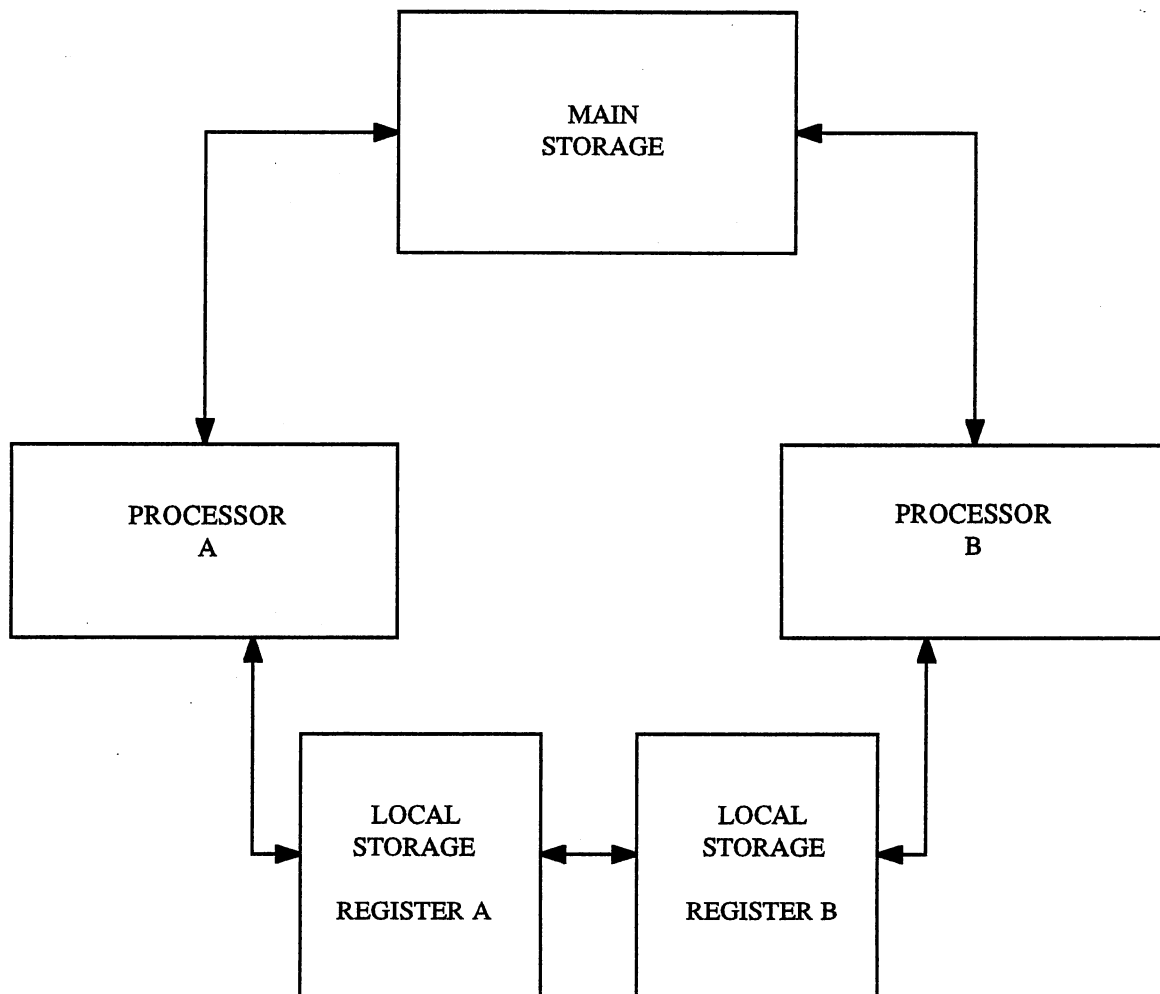


Figure 6. System Configuration of a Twin Processor Machine

The advantage of the twin processor machine is that it eliminates the need to predict the path following the conditional branch. In a machine which employs branch prediction, if an incorrect prediction occurs, then the results of the incorrect path have to be flushed. The instruction prefetch buffer also has to be flushed, and it has to be loaded with the instruction from the correct path. We can avoid these penalties in the twin processor machine because one of the processors is always executing the correct instruction stream.

The disadvantage of the twin processor machine is the extra hardware involved. The execution of multiple instruction streams requires two processors and two register files. Also, the two register files need to have the capability of being copied into each other. This requirement complicates the design of the register file. But it is to be noted that even in single processor machines which employ branch prediction techniques, some mechanism is required to maintain a backup of the local variables before the processor starts executing instructions along the predicted path.

Branch Handling

In the twin processor machine, when no branch is being processed, only one of the processors is active while the other is idle. The active processor executes instructions sequentially. As soon as the branch instruction is decoded, the idle processor starts fetching and executing the instructions from the branch target without waiting for the branch to be resolved. But before the idle processor starts executing the instructions, it is necessary that the local memories (register files) of both the processors be identical. If this is not the case, then both the processors will end up using different values for the same operands.

Once both the register files are identical, the processor which was initially active continues execution along the sequential path, while the other processor executes the instructions from the target address. The target instruction is fetched from the branch target buffer. For simplicity in the analysis, it is assumed that the machine has a very large branch target buffer so that the target instruction is always available at the decode time of the

branch instruction. Once the condition is resolved, the processor which was executing the incorrect path becomes idle, while the other processor continues executing sequential instructions.

Let us consider the execution of the same program segment which was shown in Figure 4 . Figure 7 shows the instruction flow in the pipeline. Initially the instruction stream is being executed in pipeline A, and $k+2$ is the conditional branch instruction. As soon as $k+2$ is decoded, pipeline B starts executing the instructions from the branch address (instruction m). Now two pipelines are processing the two instruction streams, and this situation continues until the condition is resolved.

Instruction $k+2$ is executed in cycle 6 and the result of the condition is available at the end of cycle 6. So from cycle 8, only the pipeline which was executing the correct path will be operational, while the pipeline which was executing the incorrect instructions is made inactive. The results of the incorrect instructions are discarded. Thus, there are no penalties involved in processing the incorrect path since the two possible paths are processed simultaneously.

CYCLE	PIPELINE SEGMENT A				
	IF	ID	OP RD	EXE	OP WR
1	k				
2	k+1	k			
3	k+2	k+1	k		
4	k+3	k+2	k+1	k	
5	k+4	k+3	k+2	k+1	k
6	k+5	k+4	k+3	k+2	k+1
7	k+6	k+5	k+4	k+3	k+2
8		k+6	k+5	k+4	k+3
9			k+6	k+5	k+4
10				k+6	k+5
11					k+6
12					

CYCLE	PIPELINE SEGMENT B				
	IF	ID	OP RD	EXE	OP WR
1					
2					
3					
4					
5	m				
6	m+1	m			
7	m+2	m+1	m		
8		m+2	m+1	m	
9			m+2	m+1	m
10				m+2	m+1
11					m+2
12					

Figure 7. Instruction Flow in a Twin Processor Machine

Effect of Branches

In the twin processor machine, both paths of a conditional branch are executed by two identical processors until the branch is resolved. If any of the processors encounters another conditional branch before the previous conditional branch is resolved, then that processor has to stall. If the probability of an instruction being a branch is high, then the frequent stalling can cause a significant penalty in the twin processor architecture. This degrading effect can be reduced by using an optimizing compiler. The compiler schedules instructions so that two successive conditional branch instructions are separated by a number of cycles greater than that required for branch resolution. This separation will ensure that the previous conditional branch is always resolved when the next conditional branch is encountered.

If the optimizing compiler ensures that once a conditional branch occurs, no other conditional branch is encountered until the first one is resolved, then we need only one cycle to execute a conditional branch instruction. This is due to the fact that the processor does not have to stall until the condition is resolved. We also need one cycle to execute any other instruction (which is not a conditional branch). Hence the average number of cycles between completion of two instructions is given by:

$$\begin{aligned} T_{ave} &= (1 - P_i) * (1) + P_i * (1) \\ &= 1. \end{aligned}$$

Thus $T_{ave} = 1.0$ and $F_b = 1.0$, and they do not depend on the number of stages in the pipeline. Comparing these results with those obtained for a single processor pipeline, we see a significant improvement in performance. In case of a five-stage pipeline, the twin processor architecture has an improvement of $\frac{1.0}{0.52} = 1.92$, and in case of a nine-stage pipeline it has an improvement of $\frac{1.0}{0.322} = 3.11$ times over a single processor pipeline.

These results have been obtained by assuming an optimizing compiler. But as the number of stages in the pipeline increases, the scheduling task of the compiler may become more and more difficult.

Now let us consider the worst case, in which the probability of an instruction i being a branch is statistically independent and there is no optimizing compiler. If instruction i is a branch, then the processor starts executing the two instruction streams. If instruction $i+1$ is also a branch, then the processor has to stall for $b-1$ cycles until the first branch is resolved. If instruction $i+1$ is not a branch, but instruction $i+2$ is, then the processor has to stall the instruction $i+2$ for $b-2$ cycles. The number of stalling cycles is reduced because when instruction $i+2$ is decoded, the instruction i is one step closer to being resolved. When instruction $i+s-3$ is decoded, the instruction i is already resolved and hence $i+s-3$ will not be stalled. Therefore, the average time between the completion of two successive instructions is given by:

$$\begin{aligned}
 T_{ave} = & (1 - P_i) * (1) + P_i [P_{i+1} * (1+b-1) + (1 - P_{i+1}) * [P_{i+2} * (1+b-2) \\
 & + (1 - P_{i+2}) * [P_{i+3} * (1+b-3) \\
 & \dots \\
 & + (1 - P_{i+(s-4)}) * [P_{i+(s-3)} * (1+b-(s-3)) \\
 & + (1 - P_{i+(s-3)}) * (1)] \dots]]. \tag{3.1}
 \end{aligned}$$

For a five stage pipeline, this reduces to:

$$\begin{aligned}
 T_{ave} = & (1 - P_i) * (1) + P_i [P_{i+1} * (3) \\
 & + (1 - P_{i+1}) * [P_{i+2} * (2) + (1 - P_{i+2}) * (1)]]. \tag{3.2}
 \end{aligned}$$

If we assume that $P_i = P_{i+1} = 0.3$, then for a five-stage pipeline we have $T_{ave} = 1.243$ and $F_b = 0.80$, an improvement of 1.53 over the single processor pipeline. In case of a nine-stage pipeline, we have $T_{ave} = 2.18$ and $F_b = 0.46$, an improvement of 1.42 over the single processor pipeline. These results are based on the assumption that the probability

of instruction $i+1$ being a branch is the same as that of i being a branch. But actually the instructions are not statistically independent, and once a branch instruction is encountered, the probability of the very next instruction being a branch is much less. If we assume that the probability of instruction $i+1$ being a branch is zero (given that instruction i is a branch), then for a five-stage pipeline we have $T_{ave} = 1.09$ and $F_b = 0.92$ (an improvement of 1.76). For a nine-stage pipeline we have $T_{ave} = 1.92$ and $F_b = 0.52$ (an improvement of 1.62). Thus it is seen that the twin processor machine results in a significant improvement in performance by processing both the instruction streams following a conditional branch.

CHAPTER IV

DESIGN OF TWIN A REGISTER FILE

This chapter presents the design of a CMOS twin register file which can be used for reducing the effects of conditional branches. A parameterized twin register file was designed using layout tools Magic and Lager. This chapter describes the design of the basic leafcells and the overall architecture of the register file.

Three-Port Memory Cell

In order to meet the goal of executing one instruction every clock cycle, the basic operations which need to be performed during each cycle of a typical processor include: reading two operands from the register file, performing an arithmetic or logic operation, and writing the result back into the register file. So the register file has to support two read operations and one write operation in every cycle, and hence a three-port register file is required in most processors.

Several design choices are available for three-port memory cells. In the RISC I "Gold Chip", a true three-port register file was used with dedicated bitlines for two read ports and one write port [19]. This design resulted in a memory cell with three bitlines. On the other hand, a shared bitline organization uses the same bitlines for reading and writing. There are only two bitlines in each cell, both of which are used for reading the two operands and for writing the result. Such an approach was used in the RISC II, resulting in a significant reduction in the size of the cell. This design also ensured an increase in the speed of the cell because of the reduced loading on wordlines and bitlines. A shared bitline register file cell reads two operands during one phase of the clock and writes one result into the register file

during the other phase. Since it supports three operations in one cycle, such a register file is called 'pseudo three-port register file'.

A basic six-transistor static memory cell for the pseudo three-port CMOS register file is shown in Figure 8. This cell consists of two cross-coupled inverters forming the storage flip flop, and two NMOS pass gates (M1 and M2) as the access transistors for the bitlines.

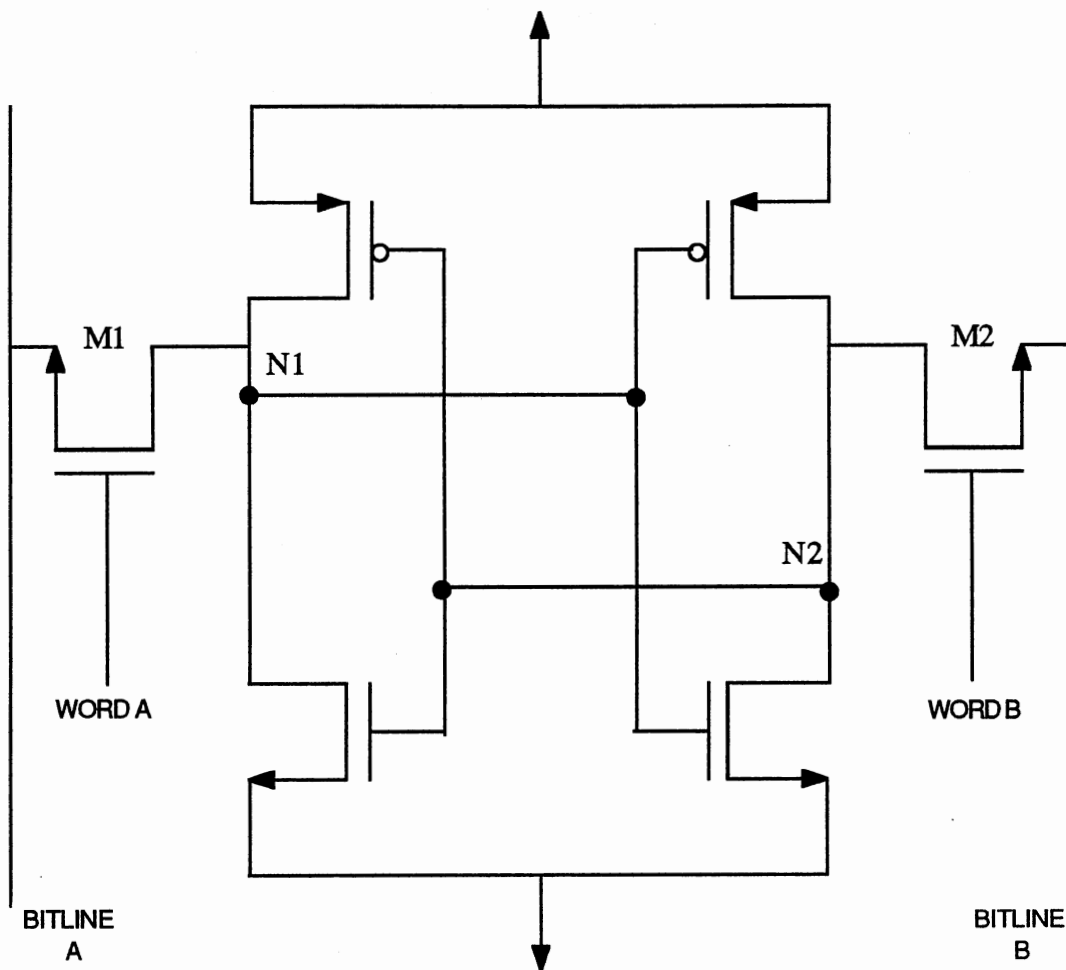


Figure 8. Pseudo Three-Port RAM

The two inverters drive each other, and the circuit can be set either to a state where node N1 is *high* and N2 is *low*, or to a state where N1 is *low* and N2 is *high*. In either case the condition is stable and the cell will not change its state unless it is forced to do so by some external means. The write operation on the cell is accomplished by applying datum and its complement on nodes N1 and N2 through the pass gates. Since the two inverters are driving each other, a state change at any one of the nodes is going to force a change at the other node.

The bitlines A and B access the internal nodes N1 and N2 of the memory cell through NMOS pass transistors. Since the NMOS pass gates are not good at passing *high*, it is difficult to write "1" at the nodes. This problem is solved by designing the inverters with very strong pull-down NMOS transistors and weak pull-up PMOS transistors. The strong pull-down transistor makes it easier to change the outputs of the inverters from "1" to "0"

The read operation is performed with single-ended sensing in which the bitlines access the storage node (N1 or N2) through a single NMOS transistor (M1 or M2). Though this single-ended sensing scheme is slower than the one which utilizes differential sensing, it is preferred because of the simplicity of the approach and the resulting compactness of the cell. A differential sensing scheme would require twice as many bitlines and an extra pair of transistors per bit [19]. Since the NMOS pass gate is not good at passing *high*, the bitlines are dynamically precharged high before reading, ensuring that "1" stored at the nodes is read correctly. If the bitline is connected to a node which has "0" on it, the cell discharges the precharged bitline during the read operation. On the other hand, if "1" is being read, the bitline remains precharged high. The speed of the read operation depends on how fast the cell can discharge the 'precharged' bitline. Since the inverters have strong pull-down NMOS transistors, the cell is able to discharge the bitlines at a high speed.

Twin Register File

This section describes the architecture of the twin register file and the design of its basic blocks. Before going into the organization of the register file, we will discuss the clocking scheme used for the twin register file.

System Timing

The register file needs to perform three operations every clock cycle: precharging of bitlines, read operation, and write operation. None of these operations can overlap because each of them requires an exclusive control of the bitlines. For these basic operations, a three-phase clock similar to the one used in the RISC I could be used. But a three-phase clock is asymmetric and difficult to generate. A four-phase clock similar to the one used in the RISC II is used in the twin register file because it is easy to generate and it also results in a simplified decoder circuit [24].

In the RISC microprocessors, the ALU operations have longer latencies than those of the register file operations. Hence, the ALU unit requires a clock which is slower than the one used in the register file. Therefore, a clocking scheme which has two fast phases and two slow phases would be more efficient. The RISC II uses a four-phase clock system having two slow phases, Phi1 and Phi3 of 80ns each, and two fast phases, Phi2 and Phi4 of 60ns each [24]. A clock with phases of shorter duration can also help in fine tuning the timing of various operations [24].

In the twin register file, a symmetric clock with two fast phases (P1f and P2f) and two slow phases (P1s and P2s) is used. The two slow phases of the clock are used throughout the system as the system clock. The two fast phases, along with the two slow phases, provide the four-phase clocking for the register file. As seen in Figure 9, such a scheme precisely defines the time intervals for the register file operations.

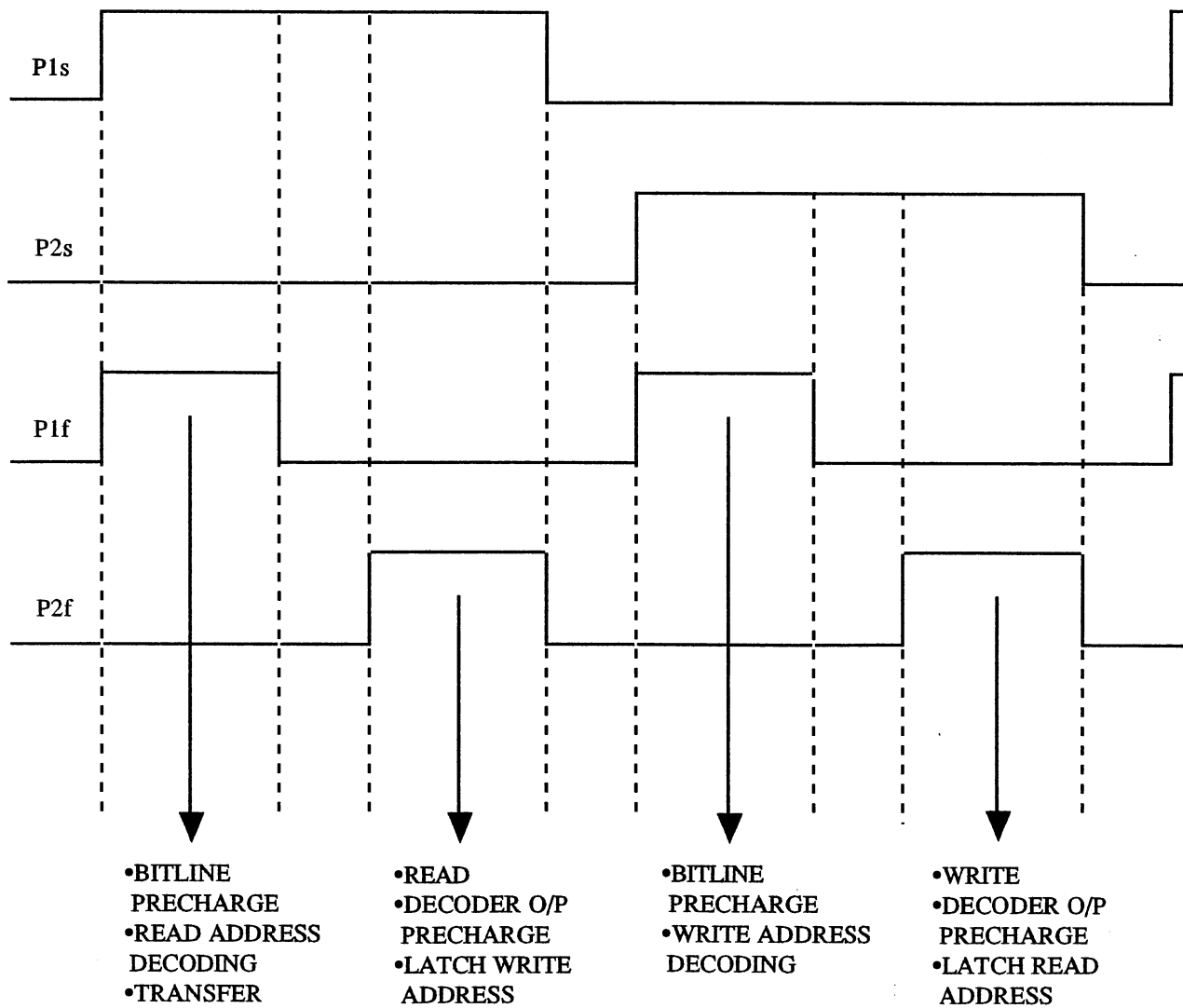


Figure 9. System Timing

The overlapped period of P1s and P1f, which is henceforth referred to as P1s•P1f, is the precharge period of the bitlines. Next, the read operation takes place during P1s•P2f, and the read data becomes valid on the bitlines before the end of this period. For simplicity, the bitlines are also precharged during P2s•P1f (though not necessary). P2s•P2f is the period during which the data is written into the register file cell.

In the twin register file, a precharged NOR decoder is used to decode the addresses because of its simplicity, regularity, and speed. Such a decoder requires that its outputs be precharged before the addresses are decoded. One way of doing this is to overlap the precharge of the decoder outputs and the precharge of bitlines during P1f, and then decode the address during the read/write period (P2f). This implies that three events occur in the same phase P2f: decoding of the address, charging of the wordlines, and the read or write operation on the selected word. In order to accomplish these three events in a single phase (P2f), we need a slower clock. This will reduce the speed of the register file.

In order to increase the speed of the twin register file, the decoding of the address is not done during the read/write phase (P2f). Instead, decoding is done during P1f, and the decoder outputs are precharged during the previous P2f. In order to prevent this precharge from appearing on the wordlines during the read/write period, a pipeline stage was added between the decoder outputs and the wordline drivers. The pipeline stage, which consists of a latch, isolates the decoder outputs and the wordline drivers. When the read or write phase begins, the address is already decoded and the drivers just need to charge up the wordlines. Such a scheme requires that the addresses for read or write operations be latched into the register file prior to the end of decoder output precharge (P2f). As shown in Figure 9, decoder outputs are precharged and addresses are latched during P1s•P2f and P2s•P2f. The addresses are decoded during P1s•P1f and P2s•P1f, and the read/write operation takes place during the next P1s•P2f and P2s•P2f.

The twin register file has the capability of transferring data from one file to another. In order to avoid any penalties in terms of time, the period of data transfer is overlapped

with the precharge period of the bitlines before the read operation ($P1s \cdot P1f$).

Organization

The basic architecture of the twin register file is shown in Figure 10. The twin register file has two three-port register files (file A and file B) which are laid out in an interleaved manner. Since each of the two files has the capability of being copied into the other one, the corresponding memory cells of the two files are placed next to each other along with the cell interconnections. This cell-to-cell connection between two files ensures data transfer from one file to another without having complicated routing.

The twin register file consists of five basic blocks: an array of memory cells, read/write circuits, address line drivers, address decoders, and control circuits. There are four address buses to provide the addresses for the two ports of each register file. There are two input buses (in A and in B) and four output buses (out A1, out A2, out B1, and out B2) forming one input and two output ports for each of the two register files. There are two transfer lines, TA2B and TB2A, which control the data transfer between register file A and register file B. A control line 'regdis' is provided in order to disable the register file. When this signal is activated, no read/write operations can be performed on the register file. The four clock phases and their complements provide the clocking to synchronize all the events in the register file.

Two groups of read/write circuits provide two output ports and one input port for each of the two register files. There are two groups of address line decoders and drivers for the two ports of each register file. One of the groups provides the wordlines for port 1 of file A and file B, while the other group provides the wordlines for port 2 of register file A and register file B. These groups also provide the two data transfer lines to the memory cell.

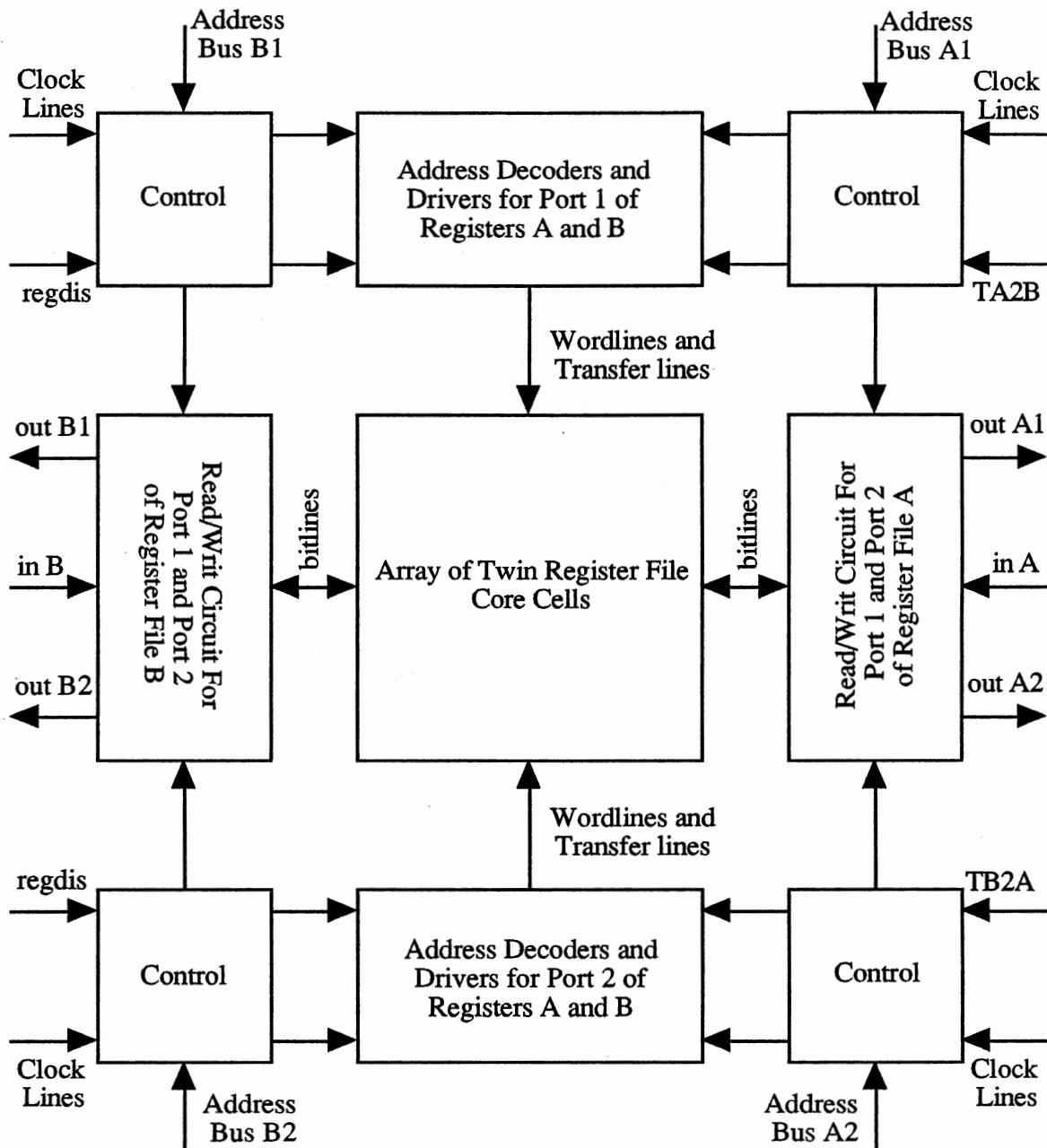


Figure 10. Twin Register File Architecture

The control sections provide the buffered address lines and their complements to the decoders. They also provide the buffered clock signals and transfer signals to the wordline drivers and read/write circuits. The control unit needs to ensure that the delay introduced on each clock line is equal in order to minimize the problem of clock skew.

Cell Placement. The memory cells in a typical register file are placed in a two-dimensional array of rows and columns, with the number of columns equal to the number of words. The address decoder selects one column, and a read/write operation can take place on all the storage cells in the selected column. The number of memory cells connected to each bitline equals the number of words. As the number of words increases, bitline capacitance increases because of the increased length of the bitlines (rows). Also, the drain capacitance on each bitline is increased by the additional memory cells connected to the bitlines, which results in a considerable speed penalty. If the number of words in a register file are much more than the number of bits in each word, the two dimensional array of cells (having number of words equal to number of columns) may result in a long and skinny layout.

The twin register file takes care of the above problems by arranging the cells in such a way that the number of columns is half the number of words. In this arrangement of cells, each column contains two consecutive words and the least significant address bit, $addr_0$, determines which of the words from the selected column is connected to the read/write circuit. Such an arrangement of cells results in a reduced number of columns and shorter bitlines (rows). The number of memory cells connected to each bitline is also reduced to half, decreasing the bitline capacitance.

Another advantage of multiplexing the words is that it reduces the size of the decoder required. For a register file of 128 words, the column decoder decodes the six most significant address bits-- $addr_1$, $addr_2$, ..., $addr_6$ --to select one column, and uses $addr_0$ to select a particular word from the column. Thus, for a 128-word register file, a 6-to-64 line

decoder is required instead of a 7-to-128 line decoder.

It should be noted that the cell placement used in the twin register file increases the number of rows, thereby increasing the wordline length. Each wordline now selects two words, so the gate capacitance on each wordline is doubled. The delay resulting from the increased wordline capacitance is reduced by using more powerful wordline drivers.

The placement of the basic cells of the twin register file is shown in Figure 11. In the twin register file, the storage cells for register files A and B are placed side by side in an interleaved manner. A basic memory cell has two storage cells: one cell for file A and the other for file B. The bitlines from cell A are taken out on one side, while those from cell B are taken out on the other side. Four memory cells form the structure *4cells*. Each *4cells* consists of two rows and two columns of storage cells. The decoders for each port of register A and register B are interleaved. *Decoder 1* decodes the addresses for port 1 of files A and B, while *Decoder 2* decodes the addresses for port 2 of files A and B. The cell *drv1* consists of drivers for port 1 of register A and register B, while *drv2* consists of drivers for port 2 of register A and register B. As seen in Figure 10, the address lines for port 1 of register files A and B are placed at the top while those for port 2 are at the bottom. These address lines are decoded by *Decoder 1* and *Decoder 2*, and are driven by *drv1* and *drv2*.

The array of cells *dst2a* and *dst2b* form the read/write circuits for each bit of file A and file B, respectively. Each of the cells *dst2a* and *dst2b* is multiplexed between two rows of memory cells, and the least significant address bit selects one of the two rows and places its data on the I/O lines of the read/write circuit. The number of read/write cells equals the number of bits in each word.

Twin Memory Cell

The twin processor machine requires two register files (A and B) which can be independently accessed by the two processors. Each register file must be capable of being copied into the other one. In order to achieve this function, two corresponding memory cells of files A and B are placed side by side along with the circuit for transferring data from one cell to another. The basic twin memory cell, shown in Figure 12, consists of two pseudo three-port static RAM cells. This memory cell is capable of executing the "read", "write" and "transfer" operations. The layout of the twin memory cell is shown in Figure 13. The cell consists of two CMOS static memory cells formed by two pairs of inverters. The cross-coupled inverters, which form the flip-flop for storing data, have a strong NMOS pull-down transistor and a weak PMOS pull-up transistor. The switching point of these inverters is set at about 1.4V. The inverters I1 and I2 form storage cell A, while I3 and I4 form storage cell B. M1 and M2 are NMOS access transistors which connect cell A to its bitlines (BITLINE A and BITLINE A-), while M3 and M4 connect cell B to its bitlines (BITLINE B AND BITLINE B-). WORD A1 and WORD A2 are the two wordlines for cell A, while WORD B1 and WORD B2 are the two wordlines for cell B. M5 and M7 are NMOS transistors which provide the direct connection between the two cells for data transfer. M8 and M6 are feedback transistors which open the back-to-back connection of the two inverters of a storage cell during data transfer.

During normal operation of the twin register file, the NMOS transistors M8 and M6 are closed, and nodes SA_BAR and SB_BAR drive I2_IN and I3_IN through these transistors. Due to the threshold voltage of NMOS pass transistors (V_{tnpass}), the nodes I2_IN and I3_IN can be charged only up to $V_{dd} - V_{tnpass}$. This may result in the weak PMOS pull-up transistor of the inverters I2 and I3 being slightly ON when *high* is stored at nodes SA_BAR and SB_BAR. Hence, *high* stored at SA_BAR and SB_BAR causes continuous current flow through I2 and I3, increasing the power consumption. This

consumption can be avoided by using CMOS transmission gates in place of NMOS pass gates M6 and M8.

During the transfer of data from cell A to cell B, the NMOS transistor M5 is turned on and M6 is turned off. *High* at SA_BAR charges I3_IN to $V_{dd} - V_{tnpass}$ through M5. This voltage has to turn on the NMOS pull-down transistor of I3 and set node SB to *low*. Since I3_IN does not reach up to V_{dd} , the noise margin is reduced. Again, CMOS transmission gates in the place of NMOS pass gates M5 and M7 can resolve this problem.

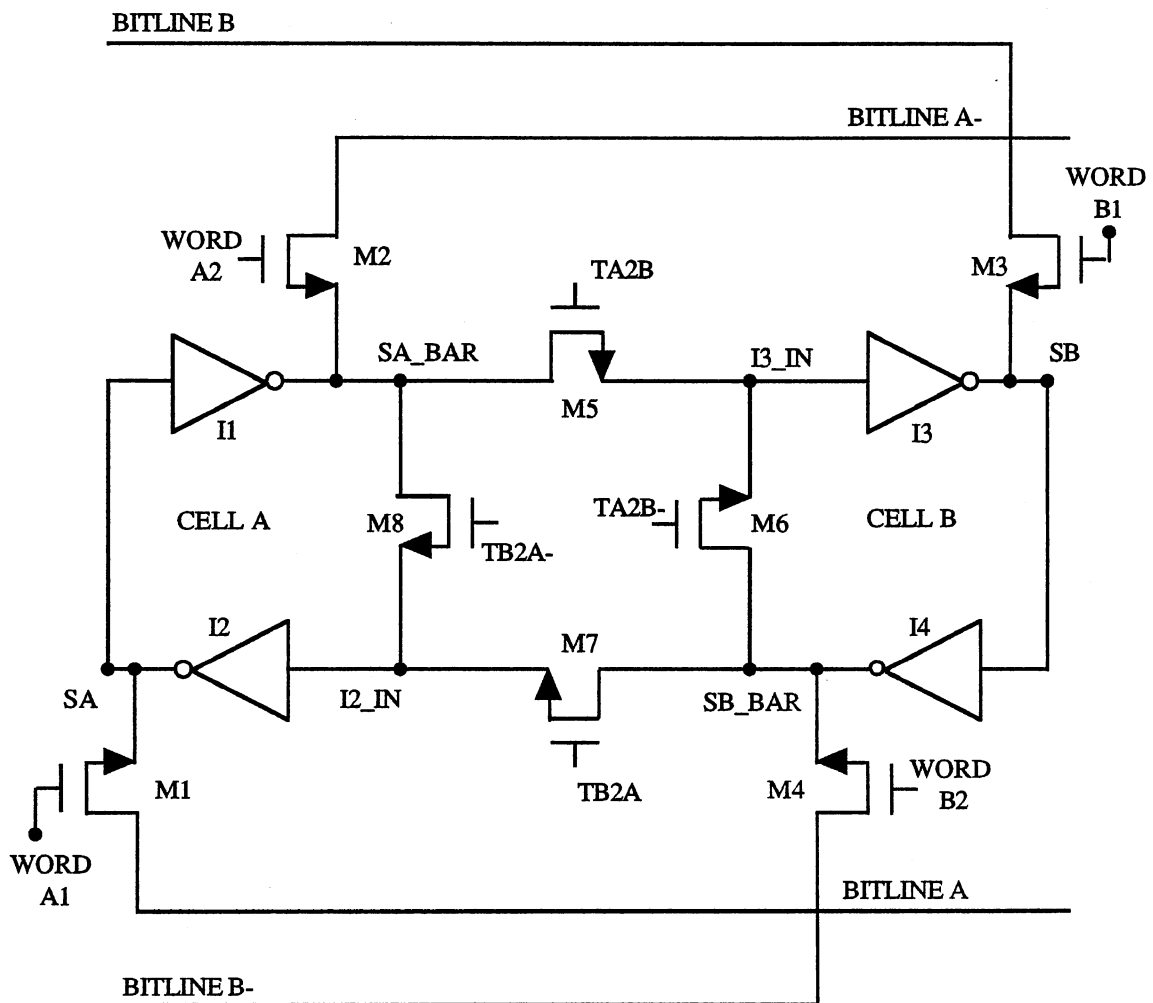


Figure 12. Twin Memory Cell

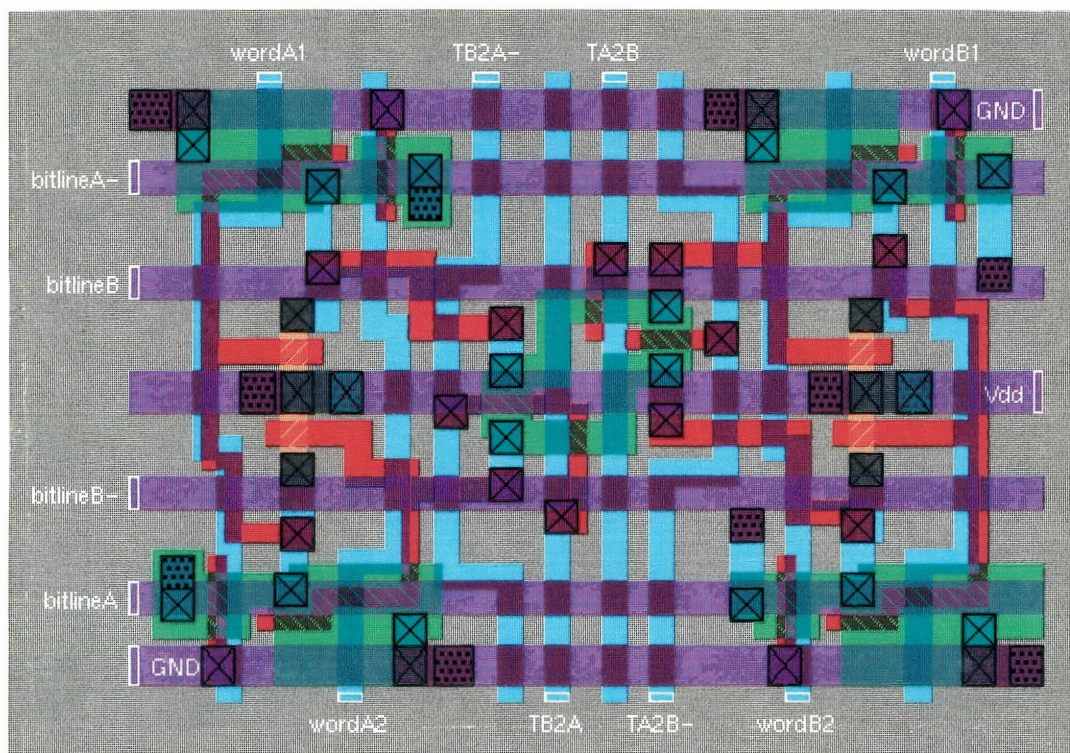


Figure 13. Layout of a Twin Memory Cell

Transfer Operation. In the twin register file, *high* on TA2B enables data transfer from cell A to cell B, while *high* on TB2A enables data transfer from cell B to cell A. Consider the transfer of data from cell A to cell B in Figure 12. Initially, TA2B is not activated, and so transistors M5 and M7 are open while M6 and M8 are closed. During the transfer operation, the data at I3_IN is to be overwritten by the data at SA_BAR. But the cross-coupled inverters which form the storage cell B resist any attempts made at changing the state of I3_IN.

In order to facilitate the transfer of data, node I3_IN is isolated from the output of inverter I4 during the transfer operation. This is achieved by activating TA2B, which turns on M5 and turns off the feedback transistor M6. The feedback transistor opens the closed loop formed by the cross-coupled inverters, making the transfer easy. The resulting equivalent circuit is shown in Figure 14. Since the node I3_IN is isolated from the output of I4, the node SA_BAR can easily overwrite the data stored at I3_IN. Such a scheme of data transfer was suggested in the 32-Bit CPU designed by Beyers, et al. [21].

Let us consider the transfer of "1" from cell A to cell B. In this case node SA is initially "1" while node SB is "0". When the transfer line TA2B is activated, the *low* at SA_BAR is passed through M5 and is applied at I3_IN. This *low* on I3_IN will turn on the weak PMOS pull-up transistor of inverter I3 and the output of I3 (node SB) will start changing its state from *low* to *high*. When SB reaches a voltage of 1.4V (the switching voltage of I4), inverter I4 will change its state and the strong pull-down transistor will immediately set its output (node SB_BAR) to 0V. The time for transferring "1" from cell A to cell B depends on the time required to activate the transfer lines and the time required to switch the output of the inverter I3 from *low* to *high*. Since the inverter I3 has a weak pull-up transistor, there is some delay involved in changing its output from "0" to "1".

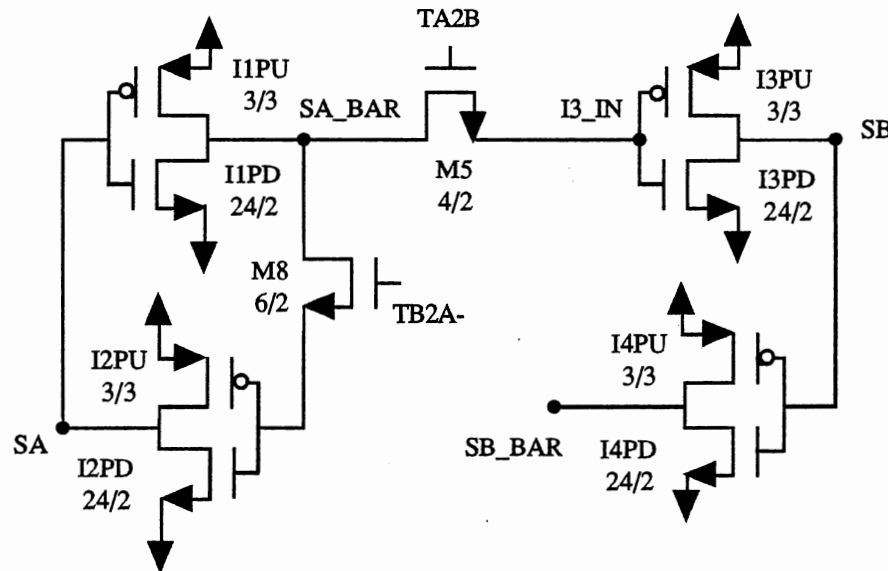


Figure 14. Equivalent Circuit During Transfer Operation

Now consider the transfer of "0" from cell A to cell B. In this case, node SA is initially "0", and node SB is "1". When the transfer line is activated, the transistor M5 is closed and it passes *high* stored at SA_BAR to node I3_IN. A NMOS pass gate is not good at passing *high* because it has a threshold voltage of about 1.5 volts. Since the inverters have been designed using strong pull-down transistors, the voltage of 3.5V applied at I3_IN will easily switch I3 and set the node SB to a *low*. This *low* will then start changing the state of I4, and the output of I4 (node SB_BAR) slowly rises to *high*. The time for transferring "0" from cell A to cell B depends on the time required to activate the transfer lines and the time required to switch the output of inverter I3 from "1" to "0". Since the inverter I3 has a strong pull-down transistor, its output rapidly changes from "1" to "0".

Under the worst case, the latency of transfer operation depends on the delay involved in activating TA2B and TA2B_BAR lines, and the time involved in changing the output of inverter I3 from "0" to "1". The delay of transfer lines depends on the RC time constant of

the transfer lines. Since each transfer line has to drive a number of cells equal to twice the number of bits in a word (because of the multiplexed wordlines), the latency increases as the number of bits increases. The latency is approximately given by:

$$T_{tr} \propto (R_{trd} + R_{trl}) * (C_{trl} + 2 * b * C_{gm5}) + (R_{pu}) * (C_{ginv}) \quad (4.1)$$

where

T_{tr}	=	Data transfer delay
R_{trd}	=	Resistance of transfer line driver
R_{trl}	=	Resistance of transfer line
R_{pu}	=	ON Resistance of the PMOS pull-up transistor of the inverters
C_{trl}	=	Line capacitance of transfer line
C_{gm5}	=	Gate capacitance of the NMOS transistor M5
C_{ginv}	=	Gate capacitance at the input of the cross-coupled inverters
b	=	Number of bits in a word.

In order to avoid any time penalties, the time for the transfer of data from one file to another is overlapped with the precharge period of the bitlines before the read operation. This is shown in Figure 9.

Read Operation. During the read operation, the transfer lines are inactive and so transistors M5 and M7 are open while M6 and M8 are closed. Two addresses are placed on the two address buses of each register file, and two words are read out on the two output ports of each register file. The decoder decodes the addresses and activates two wordlines of each register file to read two words on the BITLINE and BITLINE_BAR, respectively. If same address is placed on both the address buses of a register file, BITLINE and BITLINE_BAR will access the same word. The equivalent circuit during read operation is shown in Figure 15.

For the read operation, the bitlines are precharged *high*, and then during the read phase they are selectively discharged, depending on the contents of the memory cell. If the node to which the bitlines are connected is *low*, then the bitlines will be discharged; otherwise they will remain *high*. Consider the case when "1" stored in cell A is being read out on BITLINE_BAR. When the wordline is asserted, the precharged bitline BITLINE_BAR is connected to node SA_BAR through transistor M2. The node SA_BAR is *low* and the strong pull-down transistor I1PD is ON. The bitline discharge occurs through the two NMOS transistors: access transistor M2 and pull-down transistor I1PD of the cross-coupled inverters forming the memory cell. These two transistors will determine the pull-down speed of the bitline. The discharge rate of the bitline will depend on the bitline capacitance and the combined series resistance of the two transistors.

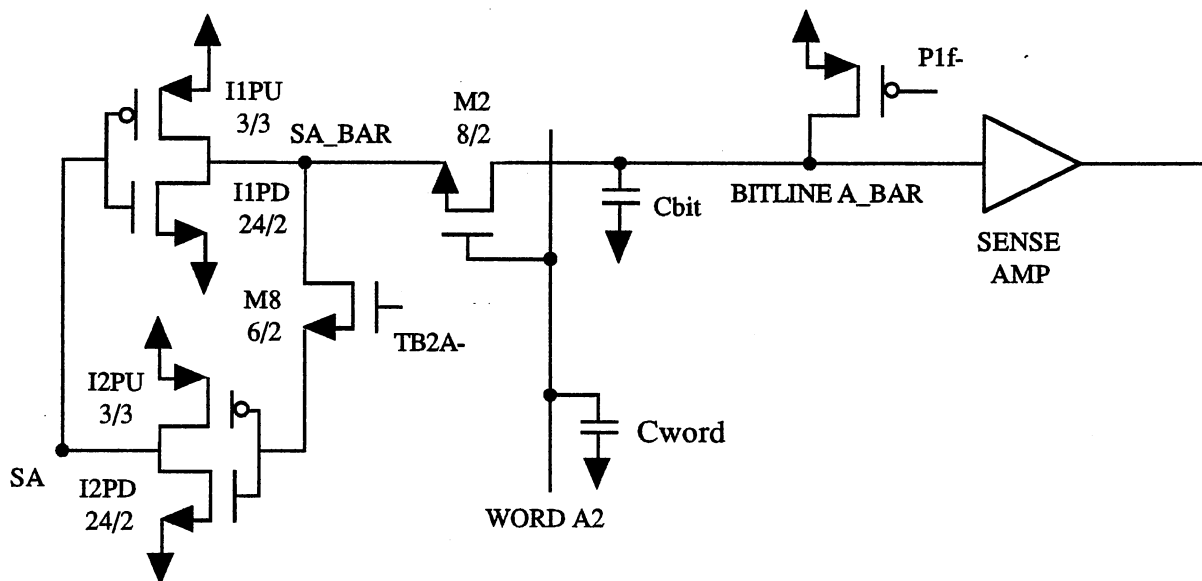


Figure 15. Equivalent Circuit During Read Operation

The voltage level of the bitlines (BITLINE and BITLINE_BAR) is sensed by an inverter. The output of this inverter is passed through another inverter to get the true level stored at the node. This second inverter also acts as a driver. The BITLINE_BAR is connected to a node which has the complementary data stored on it. Hence, BITLINE_BAR is passed through an additional third inverter to get the correct output.

The discharge rate will increase as the width of the access transistor and the pull-down transistor is increased. However, increasing the width of the access transistor will increase the gate capacitance on the wordlines, resulting in increased wordline delay. Another problem resulting from the increased width of the access transistor is 'Read Disturb', which is an undesired side effect during read operation.

The 'Read Disturb' occurs when a *low* node is read out on a precharged bitline. When the precharged line is connected to a node which is *low* (0V), it deposits some voltage at the node. A narrow pass transistor will have a significant voltage drop across it, and so the voltage deposited on the node will be below the switching point of the inverter; therefore, cell state will be maintained. On the other hand, if a wide pass transistor is used, then the voltage deposited on the inverter input may be sufficient ($> 1.4V$) to switch the output of the inverter. This will result in the cell being spuriously written into during the read operation. This undesired side effect is called 'Read Disturb'.

In order to avoid the 'Read Disturb', the ratio of the access transistor and the pull-down transistor need to ensure that the storage node is not pulled above the switching point of the following inverter during read operation. This 'Read Disturb' problem constrains the size of the access transistor, limiting the speed of the bitline discharge [19]. In the twin register file, the width of the pull-down transistor I1PD is three times that of the access transistor M2. Therefore, a *high* on the bitlines (5V) deposits a voltage of about 1.25V on the storage node which is below the switching point of the inverters (1.4V). This prevents the precharged line from writing into the cell.

The Read delay consists of two components: wordline delay and bitline discharge

delay. The wordline delay is proportional to the RC constant of the wordlines, and it increases as the number of bits in the word increases. The bitline discharge delay depends on the combined series resistance of the access transistor M2 and the pull-down transistor, and on the bitline capacitance. This delay increases as the number of words increases. In general,

$$\begin{aligned} \text{Trd} \propto & (R_{wdr} + R_{wl}) * (2 * b * C_{gm2} + C_{wl}) \\ & + (R_{m2} + R_{i1pd} + R_{bt}) * \left(\frac{n}{2} * C_{dm2} + C_{bt}\right) \end{aligned} \quad (4.3)$$

where

Trd	=	Read delay
Rwdr	=	Resistance of wordline driver
Rwl	=	Resistance of the wordline
Rm2	=	Resistance of the access transistor M2
Ri1pd	=	ON resistance of the pull down transistor of the inverter
Rbt	=	Resistance of the bitlines
Cgm2	=	Gate capacitance of access transistor M2
Cwl	=	Line capacitance of the wordlines
Cdm2	=	Drain capacitance of the access transistor M2
Cbt	=	Line capacitance of the bitlines
b	=	Number of bits in a word
n	=	Number of words in the register file.

Write Operation. During the write operation, the transfer lines are inactive, so transistors M5 and M7 are open while M6 and M8 are closed. In this mode, the twin register file behaves as two independent three-port register files. Data is written into the register files in the same way as in the pseudo three-port cell.

The write operation on cell A starts by placing the input datum and its complement on

BITLINE A and BITLINE A_BAR, respectively. The wordlines WORD A1 and WORD A2 are asserted simultaneously by placing the same address on address bus A1 and address bus A2 of the twin register file. As mentioned previously, the cross-coupled inverters which form the storage node have their switching point set at 1.4V, making it easier to change the output of the inverters from *high* to *low*, rather than from *low* to *high*.

Consider the case where SA is initially "1" and we want to write "0" on it. To achieve this, BITLINE A_BAR is driven *high* and BITLINE A is driven *low*, and the wordlines are activated, turning on the NMOS pass transistors. The equivalent circuit is shown in Figure 16. The *high* on the BITLINE A_BAR tries to drive SA_BAR *high* through the pass gate M2. Initially, I1PD is ON, and so M2 and I1PD form a voltage divider. Since the resistance of M2 is three times that of I1PD, a voltage of about 1.25V is applied on the node SA_BAR. This voltage passes through M8 and is applied on node I2_IN. There is no voltage drop across M8 because its source is connected to the gate of I2, which presents a very high impedance. This voltage of 1.25V on I2_IN is below the switching point of I2, and hence it does not change the state of I2. This helps in avoiding the problem of 'read disturb' which was discussed in the previous section.

At the same time, *low* on the BITLINE A tries to pull node SA to *low* through M1. In this case, I2PU and M1 form a voltage divider. Since the resistance of I2PU is twelve times that of M1, a voltage of about 0.4V is deposited on node SA. This voltage is below the switching point of I1, and therefore, I1 starts changing its output from "0" to "1". Thus, *low* applied on one of the bitlines is primarily responsible for changing the state of the cell. As the weak pull-up PMOS transistor of I1 starts turning on and the pull-down transistor (I1PD) starts turning off, the resistance offered by I1PD starts increasing and the voltage at SA_BAR starts rising. As soon as this voltage reaches 1.4V, I2 changes its state and pulls the node SA to *low*. Also, the weak pull-up I1PU which has turned on tries to pull node SA_BAR to a higher voltage. Once SA_BAR reaches 3.5V, M2 turns off. Eventually, the weak pull-up I1PU is fully turned on and the node SA_BAR will reach a

voltage of 5V. Driving the bitlines with data and its complement ensures that one of the bitlines is always *low*, and it is this bitline which initiates the change of state of the memory cell.

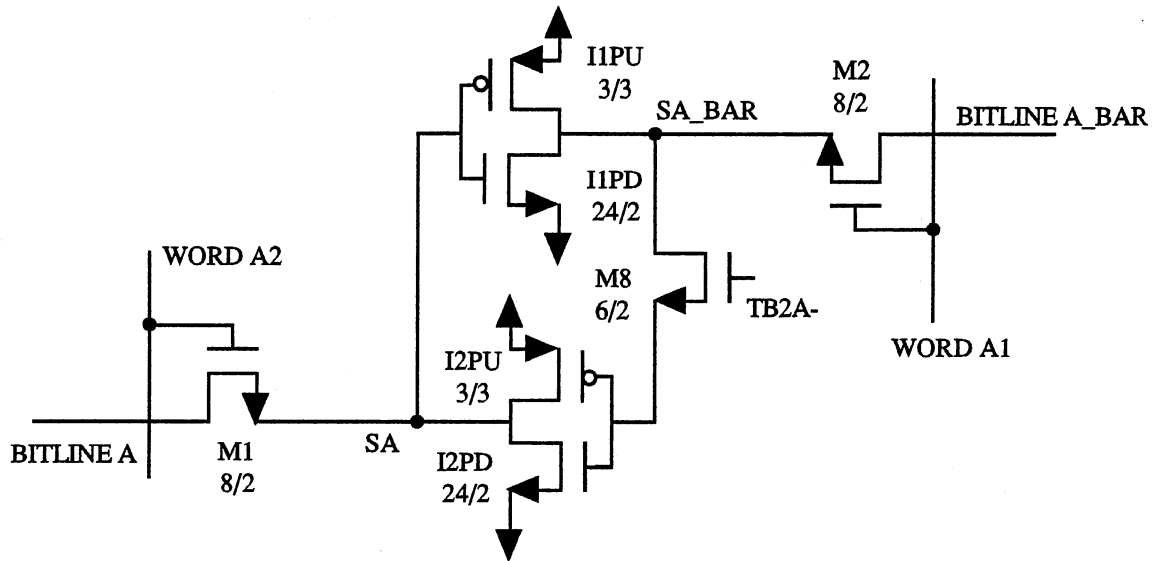


Figure 16. Equivalent Circuit During Write Operation

The write operation occurs when the wordlines are asserted and the bitlines are charged with data. The write delay depends on the time required to charge up the bitlines or the wordlines, whichever is larger. The wordline delay is proportional to the RC constant of the wordlines and it increases as the number of bits in a word increases. The bitline delay is proportional to the RC constant of bitlines, and it increases as the number of words increases. In general,

$$T_{wr} \propto \text{MAX} \left[\left\{ (R_{wdr} + R_{wl}) * (2 * b * C_{gm1} + C_{wl}) \right\}, \left\{ (R_{btdr} + R_{bt}) * \left(\frac{n}{2} * C_{dm1} + C_{bt} \right) \right\} \right] \quad (4.2)$$

where

T_{wr}	=	Write delay
R_{wdr}	=	Resistance of wordline driver
R_{wl}	=	Resistance of the wordline
R_{btdr}	=	Resistance of the bitline driver
R_{bt}	=	Resistance of the bitlines
C_{gm1}	=	Gate capacitance of access transistor M1
C_{wl}	=	Line capacitance of the wordlines
C_{dm1}	=	Drain capacitance of the access transistor M1
C_{bt}	=	Line capacitance of the bitlines
b	=	Number of bits in a word
n	=	Number of words in the register file

Address Decoder

The twin register file uses a precharged NOR address decoder for each of its four address buses. This NOR decoder is designed with only NMOS transistors for a fast implementation of the decoder logic. The regular arrangement of the transistors eases the design of a parameterized decoder. A 2-to-4 line decoder is shown in Figure 17. The input lines a_1 and a_2 are the address lines, while a_1^- and a_2^- are their complements. The output lines of the decoder Wa_0 , Wa_1 , Wa_2 , and Wa_3 are precharged high during $P2f$ through PMOS transistors M1, M2, M3 and M4. When $P2f$ goes *low*, the NMOS transistors M5, M6, M7 and M8 turn on. All the decoder output lines, except one, are discharged. The output line which remains *high* is determined by the input address lines. For example, consider the case when an address 00 is placed on the address bus. For this address, the word line Wa_0 should remain *high* while all the other wordlines need to be pulled *low*. The input address places *low* on a_1 and a_2 while it places *high* on a_1^- and a_2^- . As seen in the figure, *high* on a_1^- and a_2^- will pull the lines Wa_1 and Wa_3 *low* through the transistors M9

and M10. The *high* on a_2 also pulls Wa_2 *low* through the transistor M12. Only the selected wordline Wa_0 remains *high*, which is then fed to the address line driver.

In the twin register file there are four decoders for the two ports of the two register files. Since the cells for files A and B are placed next to each other, their wordlines also run next to each other. In order to simplify the routing of the output of the decoders to the wordlines, two decoders for the same port of the two register files are laid out in an interleaved manner. A column of one decoder is placed next to the corresponding column of the other decoder. The layout of such an interleaved 2-to-4 line decoder is shown in Figure 18. It should be noted that the height of the decoder remains the same, irrespective of the decoder size. As the decoder size increases, the width of the decoder goes on increasing. The width of the decoder is determined by the number of columns (words) in the register file, and the outputs of the decoder are aligned with the inputs to the wordline drivers. The interleaved decoder provides two word-select lines for one port of register file A and one port of register file B, respectively. The inputs a_1, a_2, \dots, a_7 are the address lines for port 1 of register file A, while b_1, b_2, \dots, b_7 are the address lines for file B. Two such interleaved decoders are used to select two ports of each register file.

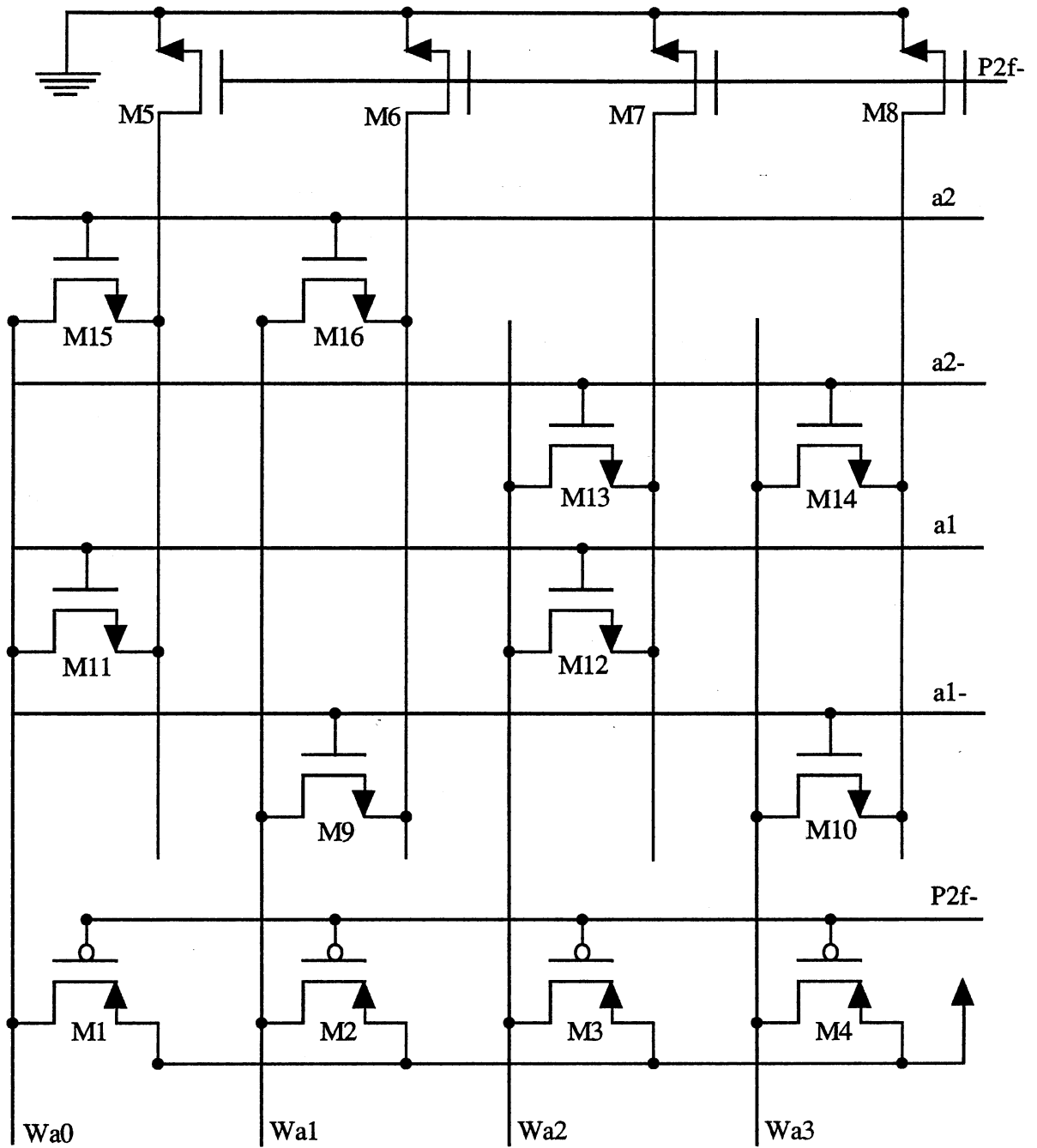


Figure 17. A 2-to-4 Line Precharged NOR Decoder

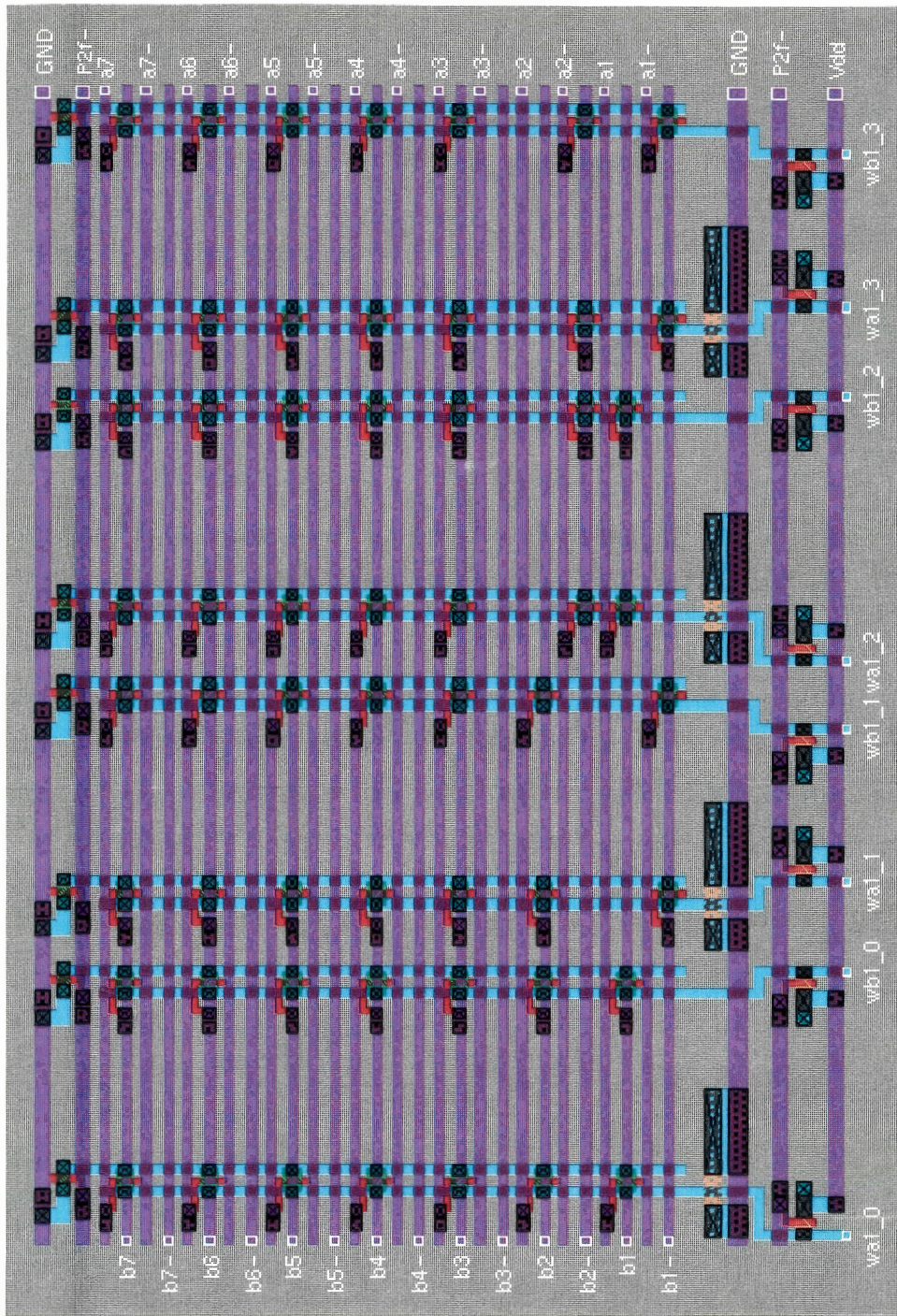


Figure 18. Layout of a 2-to-4 Line Interleaved Decoder

Line Drivers

The line drivers drive the wordlines and the transfer lines. Figure 19 shows the circuit diagram of the line driver. One such driver is provided for each wordline and transfer line connected to a column of memory cells. The output of the address decoder is input into the wordline driver. The output of the address decoder is valid during P1f, and we need to select the wordline during the next P2f. So, a pipeline stage is placed between the decoder output and the wordline. The pipeline latch is composed of a transmission gate T1 and transistors M1, M2, and M3. The latch ensures that the wordline is pulled *low* all the time except during the read/write phase (P2f). The *low* on the wordline turns off the access transistors of the memory cells. This ensures that the memory cell is isolated from the bitlines, except during the read/write period. The control signal 'regdis' is provided to disable the register file. The wordline is pulled *low* if 'regdis' is *high*, and thus the read and write operation of the cell is disabled. Strong inverters I2 and I3 speed up the charging of wordlines.

The transfer line TA2B is normally *low* and TA2B_BAR is normally *high*. The control circuit provides the transfer signal (tra2b-P1s) to the transfer line driver. If this line is *high*, then during P1f, TA2B goes *high* and TA2B_BAR goes *low*. Thus the transfer line is activated only during P1s•P1f, which is the precharge period before the read operation.

Four wordline drivers (two for each register file) are provided for each word in the twin register file. Since the wordlines for the two files run next to each other, the wordline drivers for the two files are also placed next to each other. The layout for such a wordline driver and transfer line driver is shown in Figure 20.

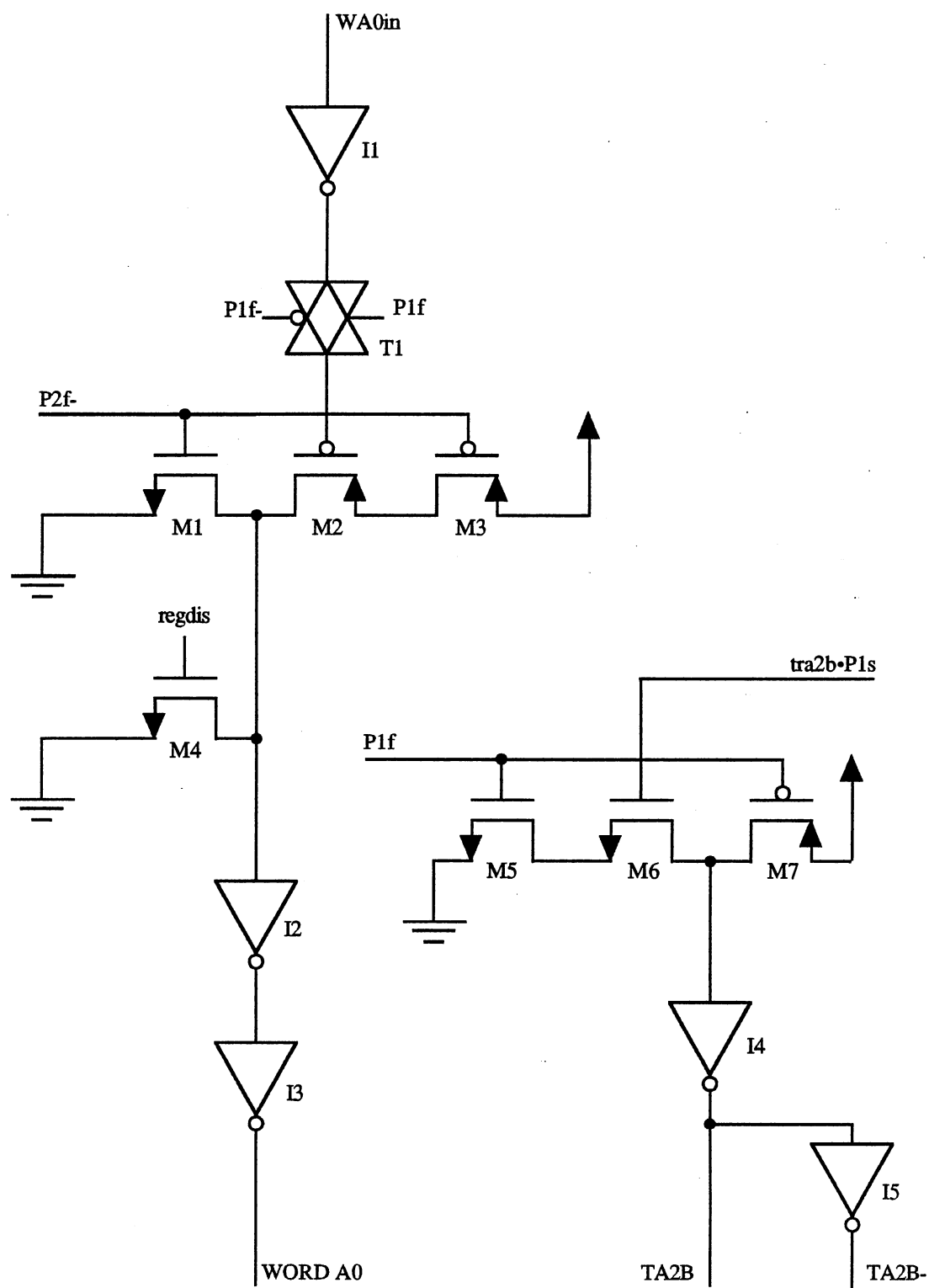


Figure 19. Wordline and Transfer Line Driver

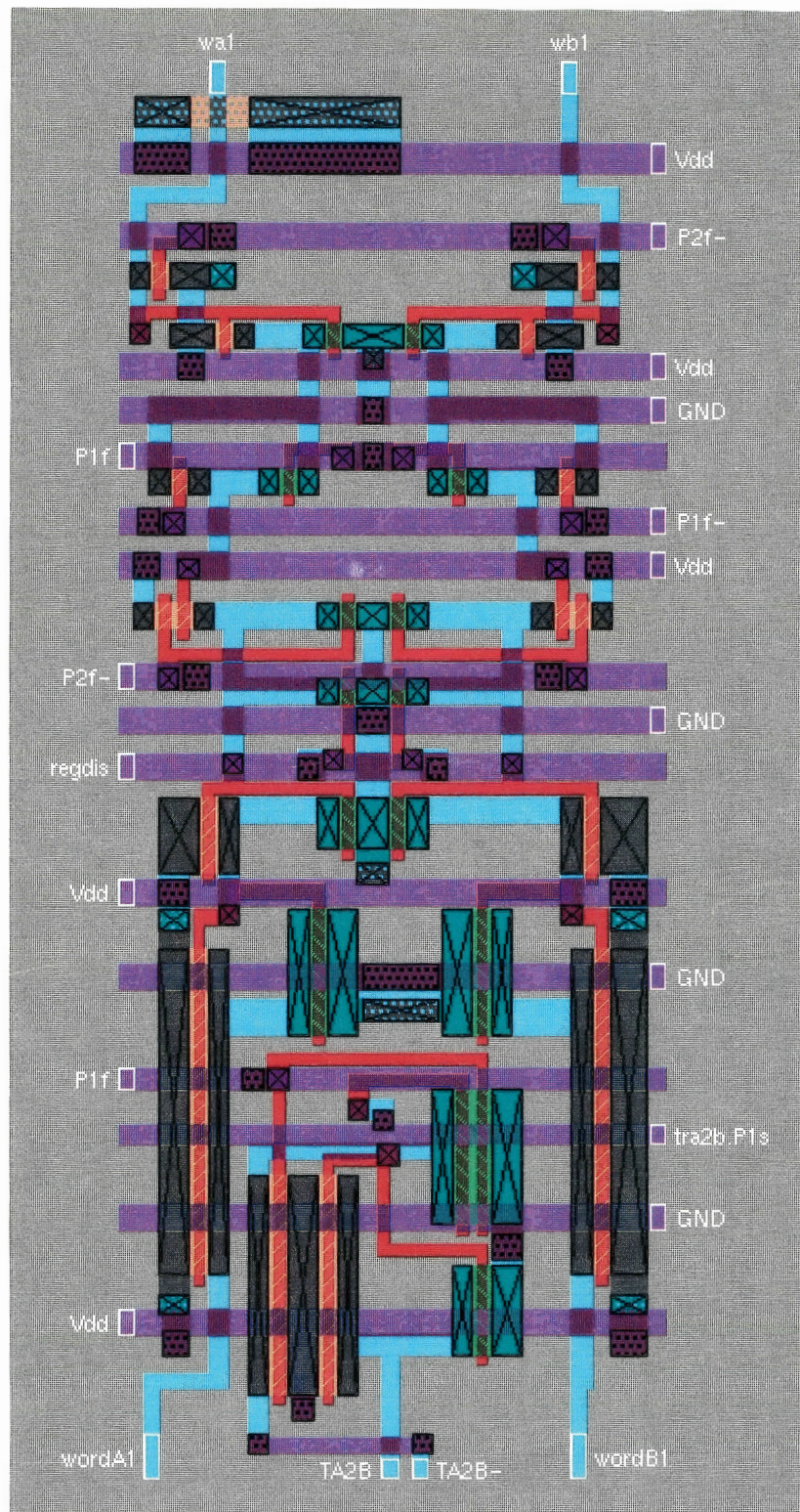


Figure 20. Layout of Wordline and Transfer Line Driver

Read/Write Circuit

The write circuit is shown in Figure 21. The input and its complement are fed into the bitline and bitline-, respectively. The two latches formed by transistors M5, M6, ..., M14, ensure that the input is placed on the bitlines only during the write phase ($P2s \cdot P2f$). The transmission gates T1, T2, T3, and T4 select the bitlines on which the input data is to be placed. The least significant address, bit a_0 , is used to demultiplex the input line between two successive words. If a_0 is *low*, then the data and its complement are placed on BITLINE A0 and BITLINE A0-, respectively. If a_0 is *high*, then the input data and its complement are placed on BITLINE A1 and BITLINE A1-, respectively.

The read circuit is shown in Figure 22. The bitlines are precharged during P1f through transistors M1, M2, M3, and M4. The inverters I1, I2, I3, and I4 act as the sense amplifiers. The output of the sense amplifiers is multiplexed through the transmission gates T1, T2, T3, and T4. The transmission gates T1 and T2 are controlled by the least significant address bit of port 1, while the gates T3 and T4 are controlled by the least significant address bit of port 2. If $a1_0$ is *low*, then the data on BITLINE A0 is placed on port 1 of the register file (output $a1$). If $a1_0$ is *high*, then the data on BITLINE A1 is placed on port 1 of the register file. If $a2_0$ is *low*, then the data on BITLINE A0- is placed on port 2 of the register file (output $a2$). If $a2_0$ is *high*, then the data on BITLINE A1- is placed on port 2 of the register file. Since the output $a2$ is reading the complementary bitlines, an extra inverter I5 is placed in its path. The latch which is placed before the output bus ensures that the output data changes during the read phase ($P1s \cdot P2f$) and remains valid for the entire clock cycle.

The layout for the read/write circuit is shown in Figure 23. Two such read/write circuits (one for register file A and one for register file B) are required for each bit in the word.

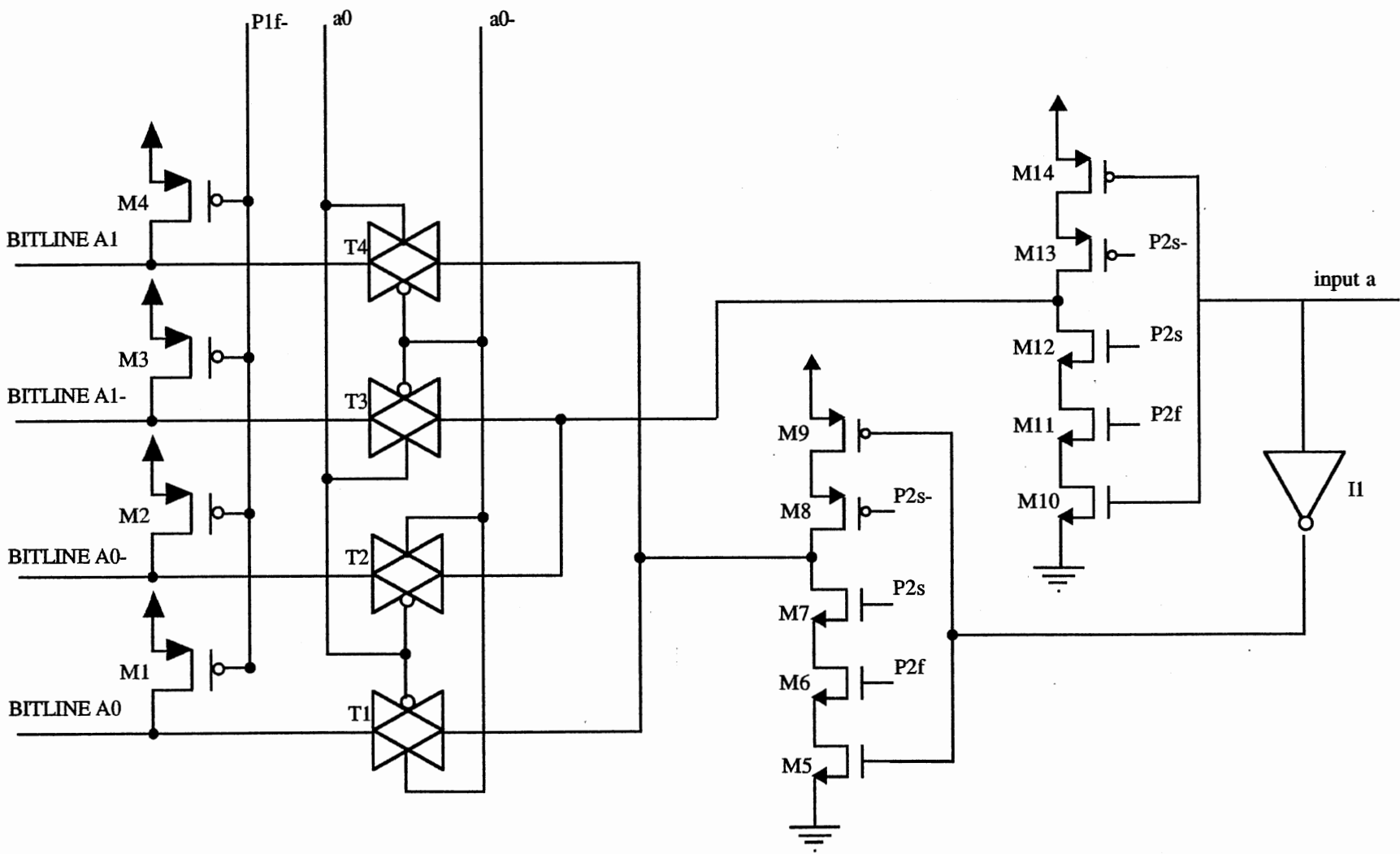


Figure 21. Write Circuit

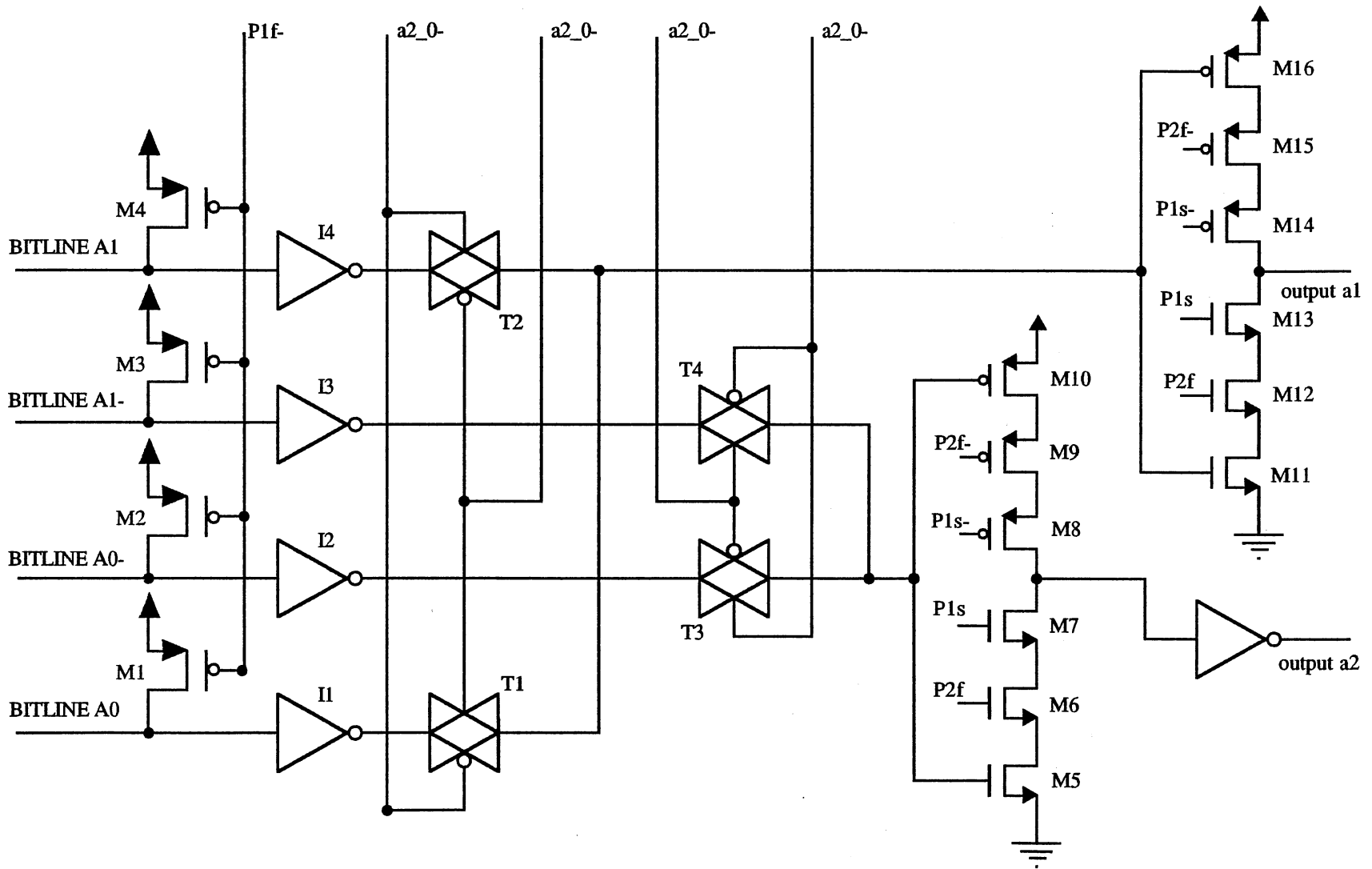


Figure 22. Read Circuit

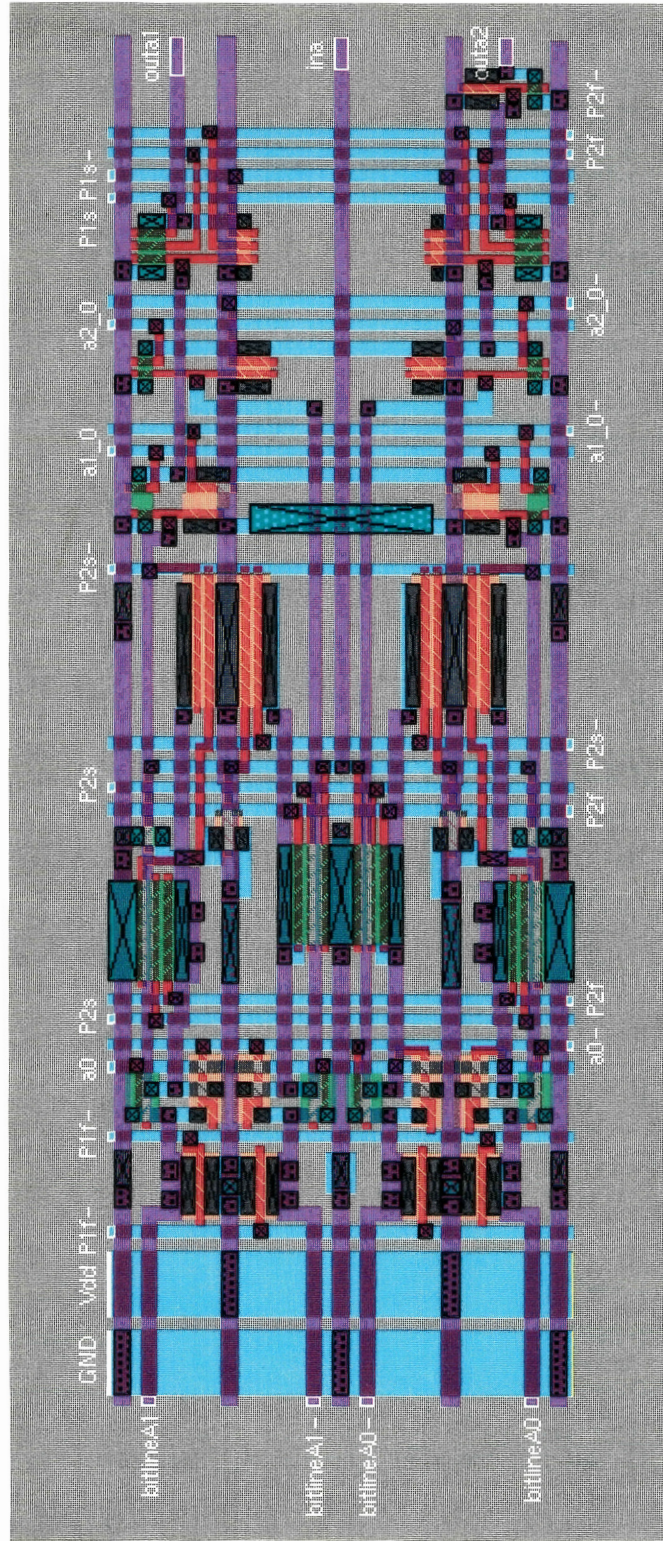


Figure 23. Layout of Read/Write Circuit

Layout of Twin Register File

The layout of a 32-word-by-16-bit twin register file is shown in Figure 24. The basic leafcells were designed using the layout tool Magic, which is an interactive editor for VLSI layouts. These basic leafcells were then put together using Lager, which is a collection of tools and cell libraries used to design custom CMOS digital integrated circuits. TimLager is one of the many tools available in the Lager System. TimLager provides a library of C functions used to write a parameterized macrocell layout generator. We used these functions to write a C program, which gives the tiling procedure for a parameterized twin register file. The C program written to tile the leafcells is given in Appendix A.

The twin register file has a total of 12,158 transistors (8368 NMOS transistors and 3790 PMOS transistors) and an area of $3250 \times 2520 \mu\text{m}^2$. A typical three-port register file which uses the same size transistors for its memory cell and other associated circuits as the twin register file has an area of $3250 \times 1007 \mu\text{m}^2$. The twin register file is about 2.5 times larger than a typical register file due to the fact that it has twice the number of memory cells, twice the number of bitlines, two extra decoders, and an extra read/write circuitry. The twin register file also has four extra transfer lines passing through each memory cell.

The increased width of the twin register file results in longer bitlines which results in increased line capacitance of the bitlines. The capacitance associated with the bitlines in the twin register file is about 450 fF, which is about 50fF more than the bitline capacitance of an ordinary register file. According to equation 4.3, the read delay is proportional to the bitline capacitance and to the ON resistance of the NMOS pull-down transistor of the cross-coupled inverters. The effect of increased capacitance can be reduced by lowering the resistance of the pull-down transistor and by designing better sense amplifiers.

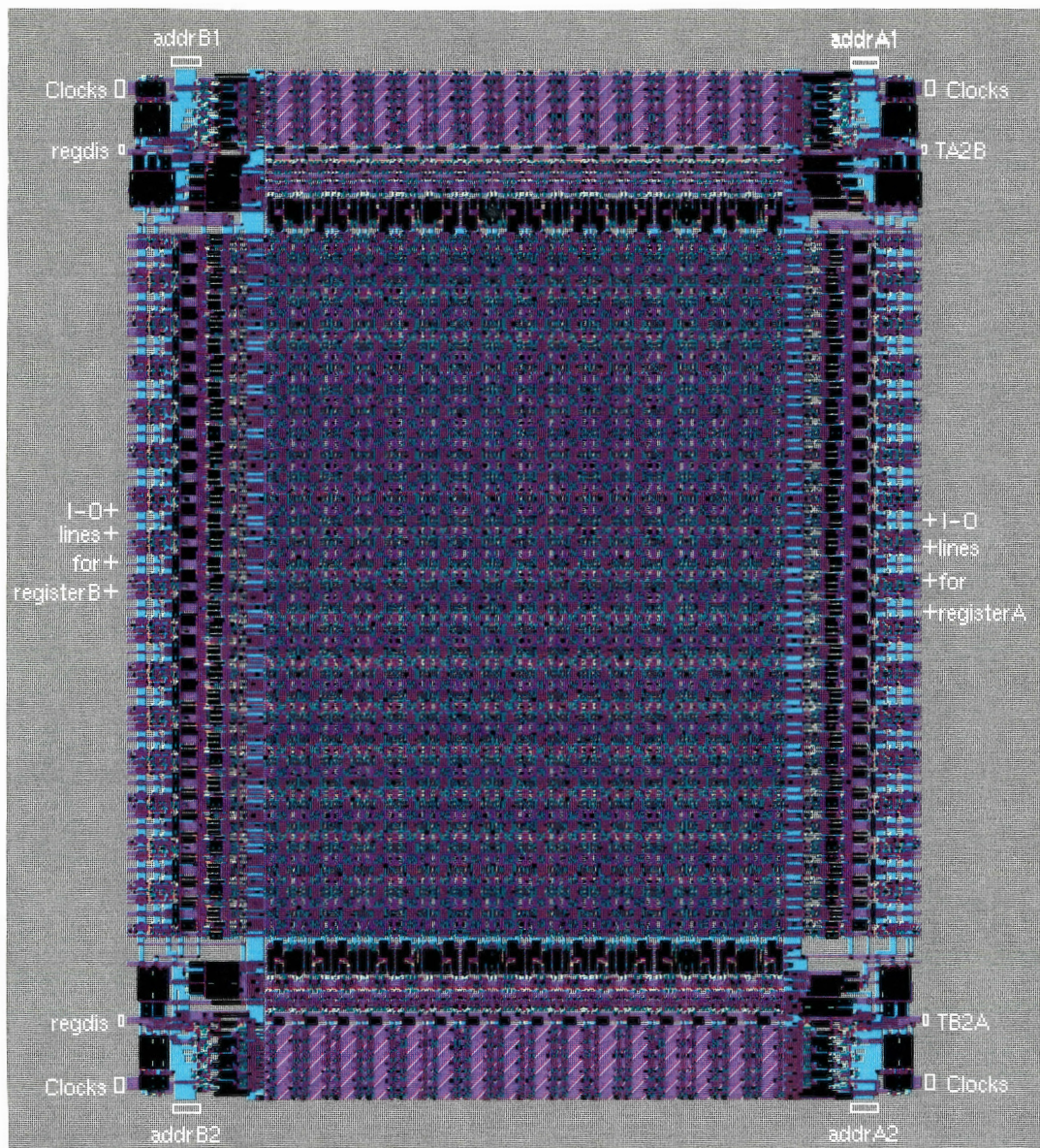


Figure 24. Layout of a 32-Word-by-16-Bit Twin Register File

CHAPTER V

SIMULATION RESULTS

This chapter presents the results of simulations performed on the 32-word-by-16-bit twin register file. The functionality of the design is verified by simulating it on IRSIM. The read, write, and transfer delays of the register file are estimated by simulating the twin register file on SPICE.

IRSIM Results

IRSIM is an event-driven logic-level simulator for MOS transistor circuits in which each transistor is modeled as a resistor in series with a voltage controlled switch. Also, the capacitance associated with each node is considered in IRSIM. The simulator evaluates the voltage levels and transition times at the nodes from the resulting RC network. In our study, we used IRSIM to verify the functionality of the circuit.

The twin register file was exhaustively tested for read, write, and transfer operations on IRSIM. At every write cycle ($P2s \cdot P2f$), an input datum is written into a selected location of the register file. Also, at every read cycle ($P1s \cdot P2f$), data stored in two locations are read out on the two ports of each register file. Table I shows the data which were written into different locations of the register file A. The addresses, which were read out on the two ports at every read cycle, are shown in the table along with the expected data on the output ports.

TABLE I
SIMULATION DATA FOR REGISTER FILE A

Clock	Write Address	Write Data	Read Address for Port 1	Expected Data on Port 1	Read Address for Port 2	Expected Data on Port 2
1	00	55aa	00	55aa	00	55aa
2	05	8925	00	55aa	00	55aa
3	13	04be	05	8925	00	55aa
4	11	32f0	13	04be	05	8925
5	0f	0cd2	11	32f0	05	8925
6	1f	1076	0f	0cd2	13	04be
7	08	4f4c	05	8925	1f	1076
8	16	f0db	08	4f4c	16	f0db
9	09	9009	13	04be	00	55aa
10	1a	23fa	11	32f0	09	9009
11	01	aaaa	1a	23fa	01	aaaa

Table II shows the data which were written into different locations of register file B. It also shows different locations which are read out on the two ports at every read cycle and the expected data on each port.

TABLE II
SIMULATION DATA FOR REGISTER FILE B

Clock	Write address	Write Data	Read Address for Port 1	Expected Data on Port 1	Read Address for Port 2	Expected Data on Port 2
1	00	acab	00	acab	00	acab
2	12	ed65	00	acab	00	acab
3	13	3c30	00	acab	12	ed65
4	01	4e29	12	ed65	13	3c30
5	1f	f90e	01	4e29	13	3c30
6	09	c332	00	acab	1f	f90e
7	0e	1afb	13	3c30	01	4e29
8	15	5435	0e	a1fb	12	ed65
9	08	9b81	15	5435	1f	f90e
10	17	635d	08	9b81	0e	1afb
11	09	fd03	13	3c30	17	635d

The results of IRSIM simulation with the data of Table I and Table II are shown in Figure 25. The figure shows the four access addresses (addra1, addra2, addrb1, and addrb2) placed on the two ports of each register file at every clock cycle. It also shows the two input data (ina and inb) placed on the input bus of the register files, and the four output data (outa1, outa2, outb1, and outb2) read out on the two ports of the two register files. To write data, the same address is placed on the two address buses of a register file during P1s•P2f, and the write datum is placed on the input bus during P2s•P1f. The write operation on the register file takes place during P2s•P2f.

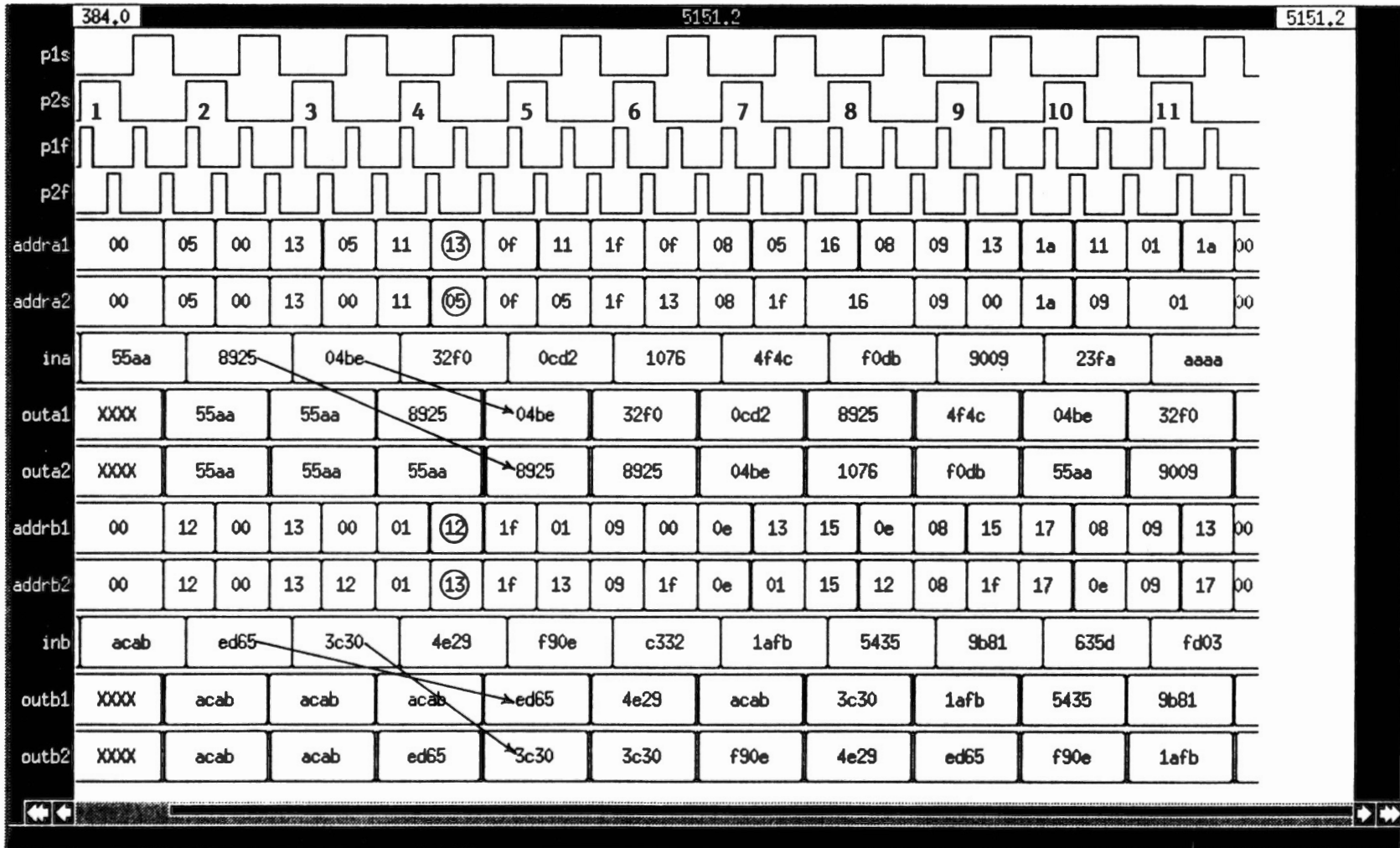


Figure 25. Read/Write Operation of Twin Register File

Addresses for the two output ports are placed on the address buses during P2s•P2f, and the data appear on the output ports on P1s•P2f. As seen in the figure, we can write into the two register files independently, and two locations in each file can be read out on the two output ports. For example, 8925 is written into location 05 of register file A during cycle 2, and 04be is written into location 13 of file A during cycle 3. Both of these locations are read out on the two ports during cycle 4. As seen in Figure 25, the output data matches the input data. Figure 25 also shows the read/write operation of file B. The simulation demonstrates the basic functions of the twin register file: writing one word in each register file and reading out two words from each file every clock cycle.

Figure 26 shows the transfer operation from file A to file B. File A is written with the same data as in Table I, while file B is written with the data in Table III. Data in file A is copied into file B by activating the transfer signal 'ta2b' during cycle 5. The data transfer takes place during P1s•P1f. In order to verify the transfer operation, the data written into file A during the first five cycles are read out from file B during the subsequent cycles. In Figure 26, 04be is written into location 13 of file A during cycle 3. On the other hand, 3c30 was written into location 13 of file B during cycle 3. During cycle 7, location 13 of file B is read out on port 2. The output data is 04be, which was the data written into file A during cycle 3 and transferred to file B during cycle 5. This verifies the transfer operation from file A to file B.

TABLE III
TRANSFER DATA FOR REGISTER B

Clock	Write addresses	Write Data	Address for port 1	Expected Data on Port 1	Address for port 2	Expected Data on Port 2
1	00	acab	00	acab	00	acab
2	12	ed65	00	acab	00	acab
3	13	3c30	12	ed65	13	3c30
4	01	4e29	01	4e29	12	ed65
5	1f	f90e	05	8925	13	04be
6	09	c332	00	55aa	05	8925
7	0e	1afb	11	32f0	13	04be
8	15	5435	05	8925	11	32f0

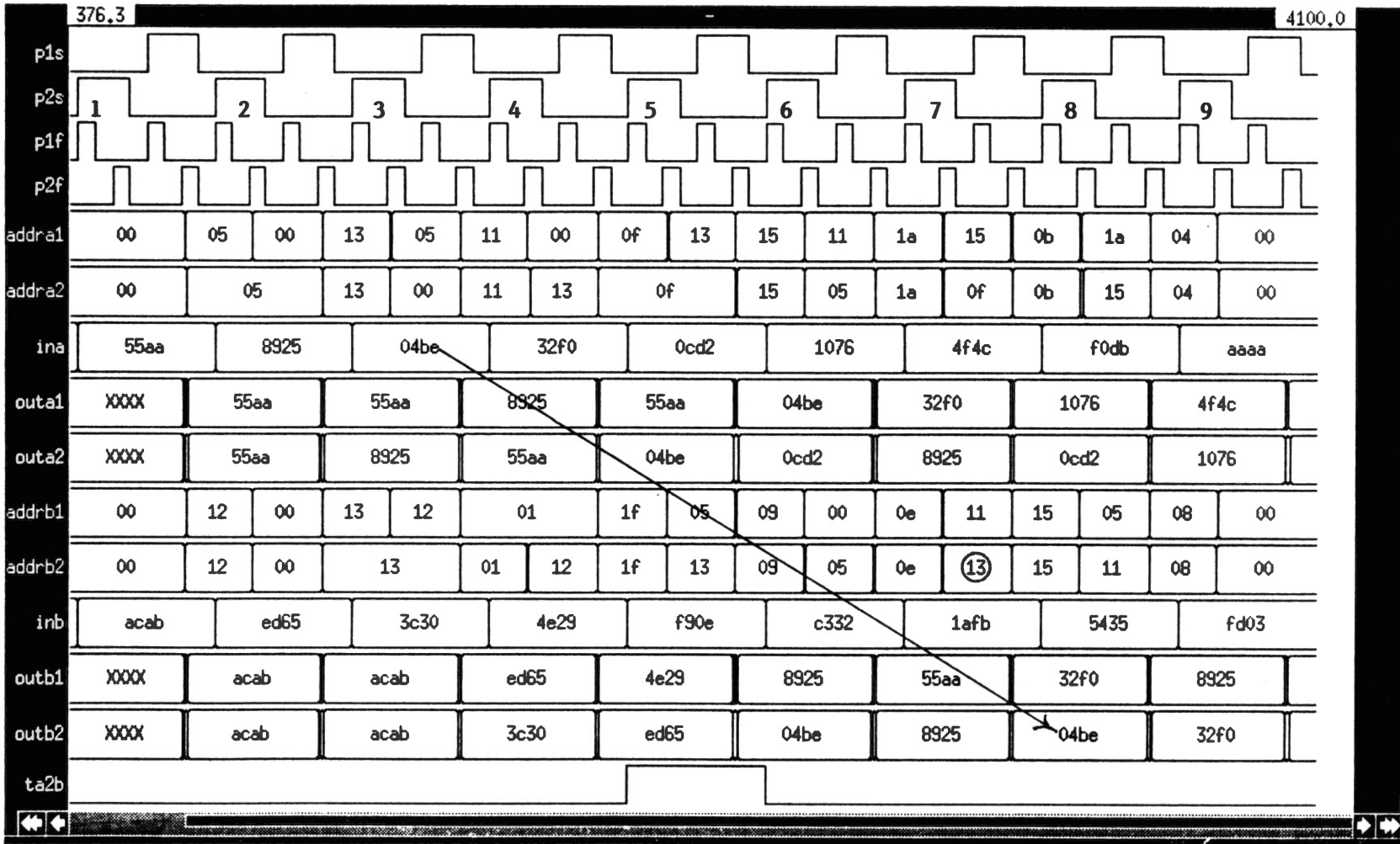


Figure 26. Transfer Operation from Register File A to Register File B

Figure 27 shows the transfer operation from register file B to register file A. In this case, file B is simulated with the data in Table II, while file A is simulated with the data in Table IV. In order to transfer data from file B to file A, the transfer signal 'tb2a' is activated during cycle 5. This simulation verifies the transfer operation from file B to file A.

TABLE IV
TRANSFER DATA FOR REGISTER A

Clock	Write Addresses	Write Data	Address for Port 1	Expected Data on Port 1	Address for Port 2	Expected Data on Port 2
1	00	55aa	00	55aa	00	55aa
2	05	8925	00	55aa	05	8925
3	13	04be	05	8925	00	55aa
4	11	32f0	00	55aa	13	04be
5	0f	0cd2	01	4e29	00	9cab
6	11	1076	12	ed65	01	4e29
7	10	4f4c	13	3c30	12	ed65
8	02	f0db	00	acab	13	3c30

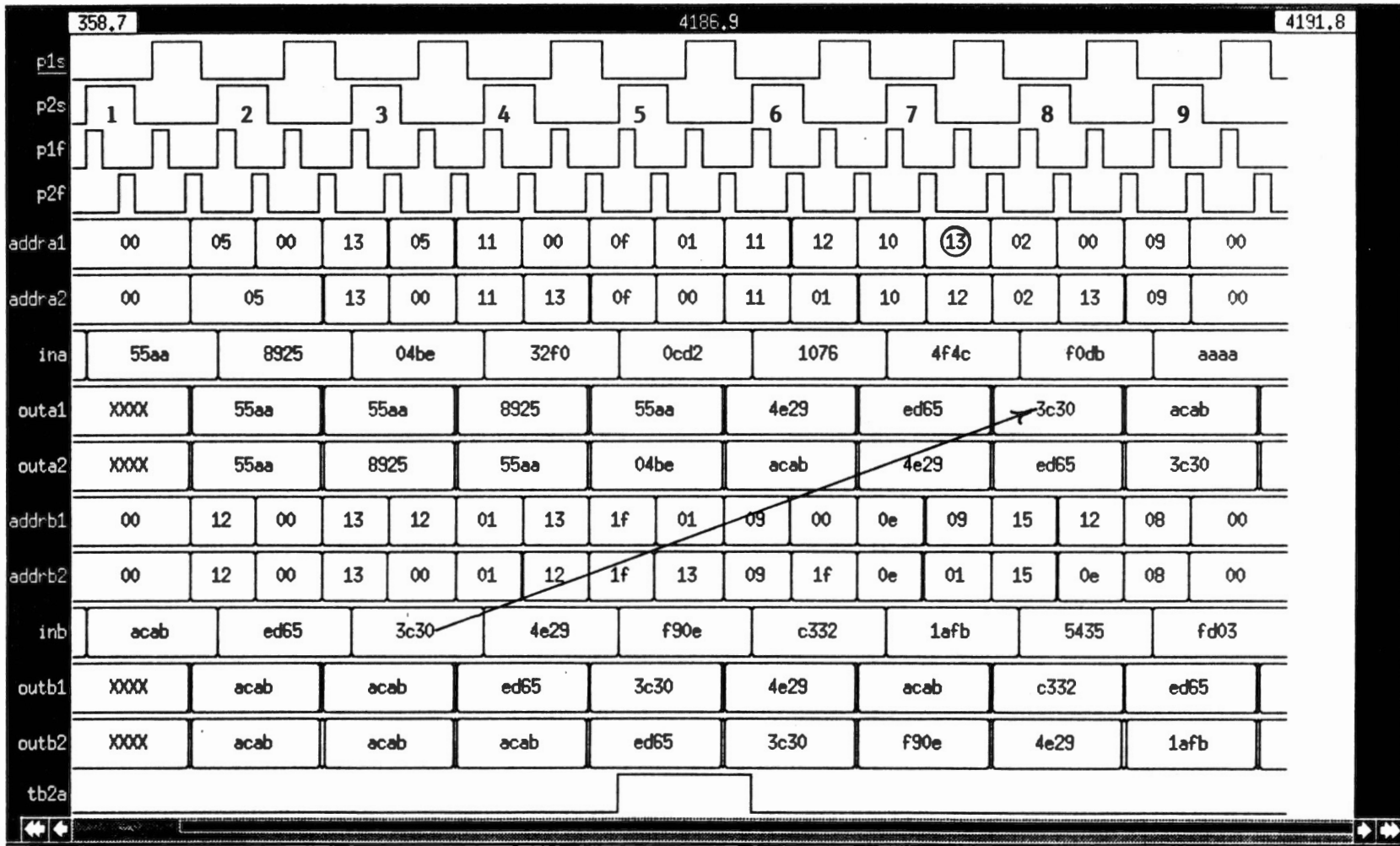


Figure 27. Transfer Operation from Register File B to Register File A

SPICE Simulation Results

In order to get an estimate of the read, write, and transfer delays of the register file, we simulated the register file on SPICE. A register file with a single memory cell and associated read/write circuit, address decoders, and drivers was designed and extracted to get the SPICE simulation file. In order to obtain accurate results, the capacitances associated with the bitlines, wordlines, and transfer lines in a 32-word-by-16-bit register file were determined (by extracting the entire twin register file), and these capacitances were included in the SPICE simulation file of the test circuit. The clock signals were fed in with realistic rise and fall times of 0.9ns, while the address signals were fed in with rise and fall times of 1.5ns. These rise and fall times were determined by calculating the RC loading on each of the clock lines and address lines. The SPICE file which was used in the simulation is given in Appendix B.

Wordline Delay

The wordline delay is dependent on the RC constant of wordlines. Each wordline is driving thirty-two access transistors in a column of memory cells. The gate capacitance of these access transistors is the major source of capacitance in the wordlines. As the number of bits in a word increases, the number of transistors connected to the wordline increases, leading to an increased wordline delay. The wordline delay can be reduced to some extent by using more powerful drivers.

Figure 28 shows the wordline delay of the twin register file. It takes around 6ns for the word line to go *high*, and it takes around 3ns to pull the wordline *low*. Since the read or write operation cannot be started until the wordline goes *high*, the wordline delay limits the read/write time. Each read/write period (P2f) is followed by a precharge period (P1f). During the precharge period, the wordlines should be pulled *low* so that the memory cell is not spuriously written into. Therefore, the precharge period cannot begin until the wordline

is pulled *low*. Thus the time required to pull down the wordline determines the non-overlap period between P2f and P1f.

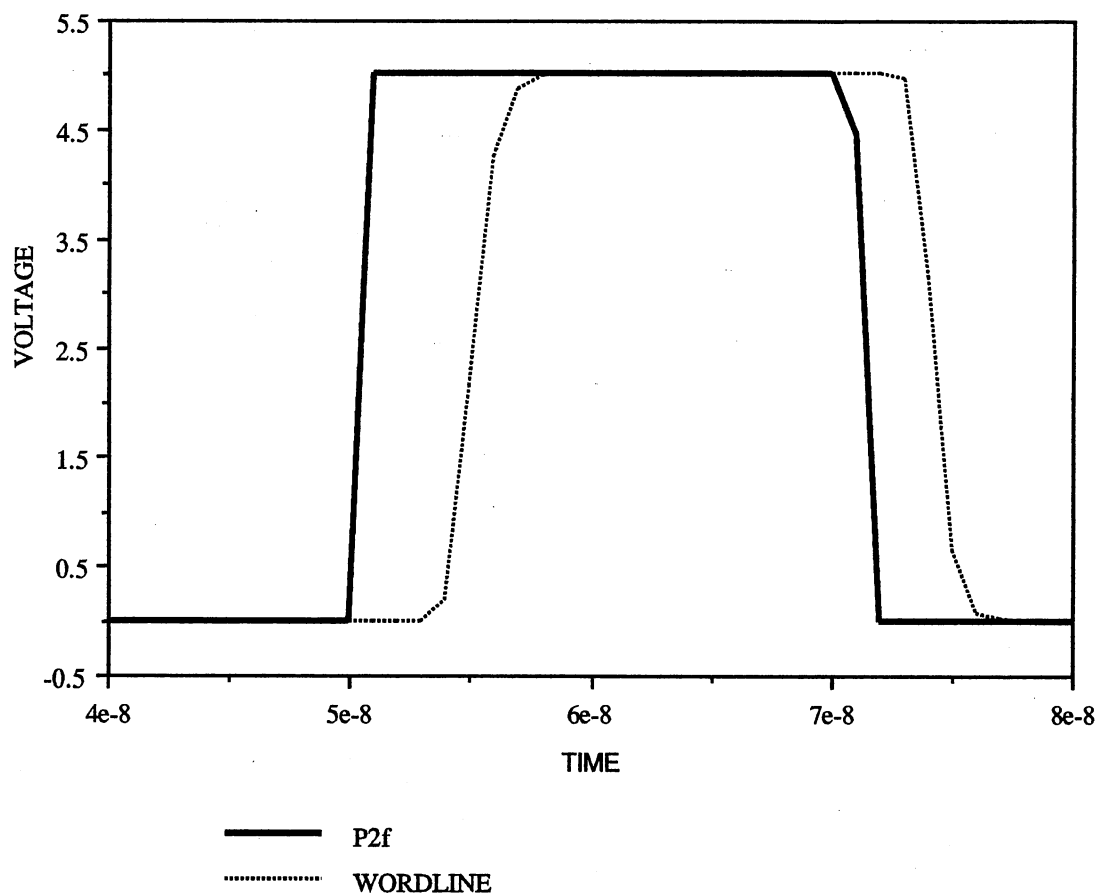


Figure 28. Wordline Delay

Write Delay

Figure 29 shows the write delay in the twin register file. The write operation takes place during P2f. The figure shows the wordline and the internal nodes of the memory cell.

High is being written on node SA, while *low* is being written on node SA_BAR. Since the inverters forming the storage nodes have a strong pull-down transistor and a weak pull-up transistor, it is easier to write *low* at the storage nodes. The node at which *low* is being written changes its state rapidly and settles to *low* within 8ns. This is a stable state, and we could terminate the write operation as soon as one of the two nodes reaches *low*. This node is going to pull the other node *high*, thus, the write operation takes around 8ns in the twin register file.

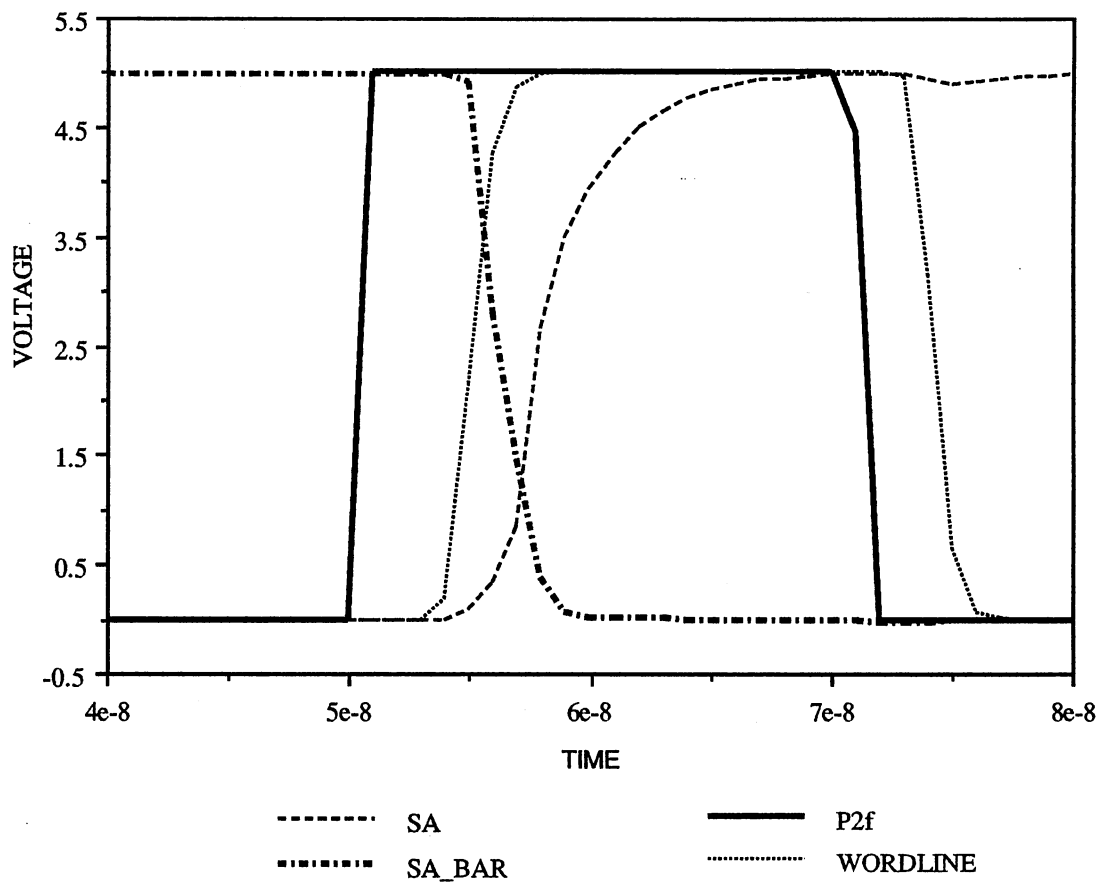


Figure 29. Write Delay

Precharge Delay

The bitlines in the twin register file are precharged before every read/write cycle. Figure 30 shows the bitline being precharged on P1f. The precharge delay of the bitline is proportional to the capacitance of the bitline. This delay increases as the number of words in the register file increases. As seen in the figure, the twin register file has a precharge delay of 8ns. The precharge delay can be reduced to some extent by increasing the width of the PMOS pull-up transistors connected to each bitline.

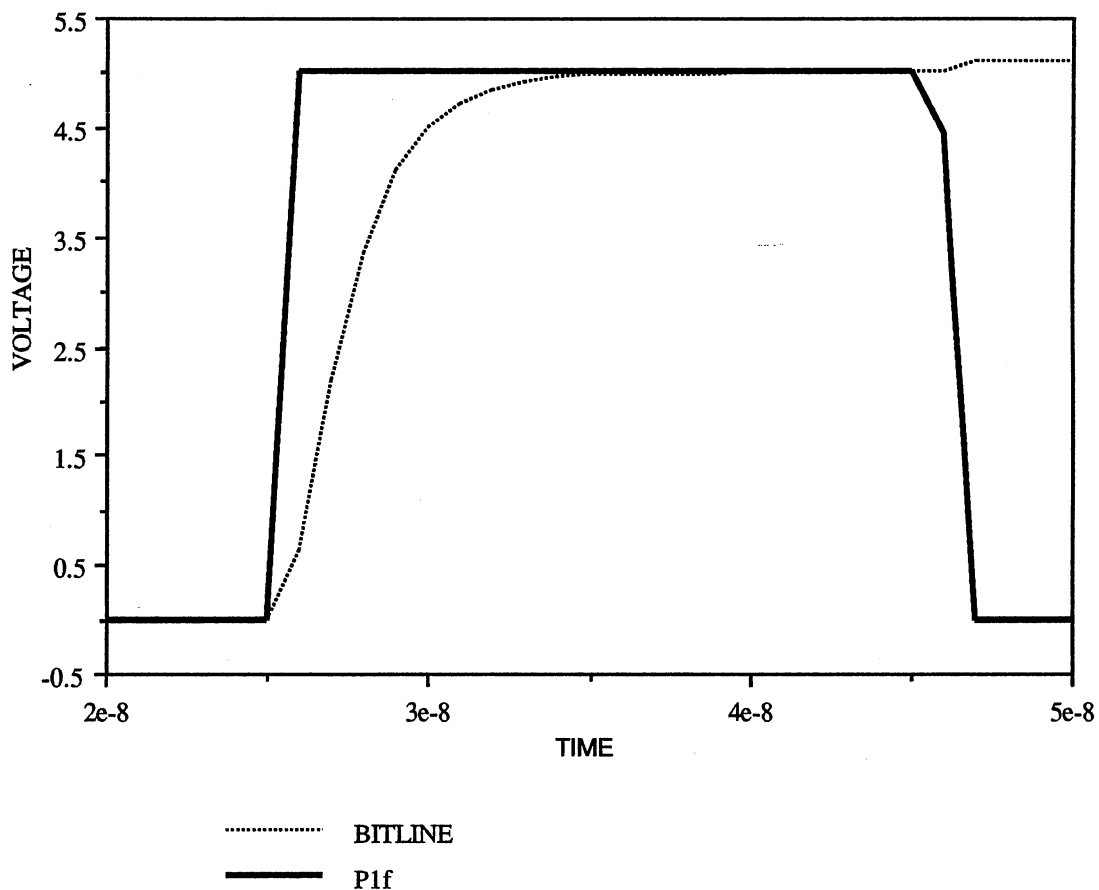


Figure 30. Precharge Delay

Read Delay

The read delay consists of two parts: wordline delay and bitline discharge delay. The bitline discharge delay depends on the bitline capacitance and the combined series resistance of the access transistor and the pull-down transistor. The wordline delay was shown in Figure 28. As soon as the wordline goes *high*, the selected bitline starts discharging (if it is connected to a node which is *low*). If the bitline is connected to a node which is *high*, then the bitline remains precharged high. The bitlines are connected to inverters which sense the voltage level of the discharging bitlines. The bitline connected to port 2 senses complementary data, and therefore it is passed through an extra inverter. Due to the precharge, port 1 is initially *high*, while port 2 is *low*. If *low* is stored in the cell, then the output of port 1 will go from *high* to *low* (if the cell is read out on port 1), while port 2 will remain *low* (if the cell is read out on port 2). On the other hand, if *high* is stored at the cell, then port 1 stays *high*, while port 2 goes from *low* to *high*.

Figure 31 shows the read delay when *low* is stored in one of the cells of register file A and *high* is stored in one of the cells of register file B. Cell A is read out on port 1 of file A, while cell B is read out on port 2 of file B. Hence, the output of port 1 of register A is going *low*, while the output of port 2 of register B is going *high*. The figure also shows the bitline discharging. As seen in the figure, the bitline discharges to 0.5V in 10ns and at this time the outputs rapidly change their states. Both the outputs are stabilized in 11ns.

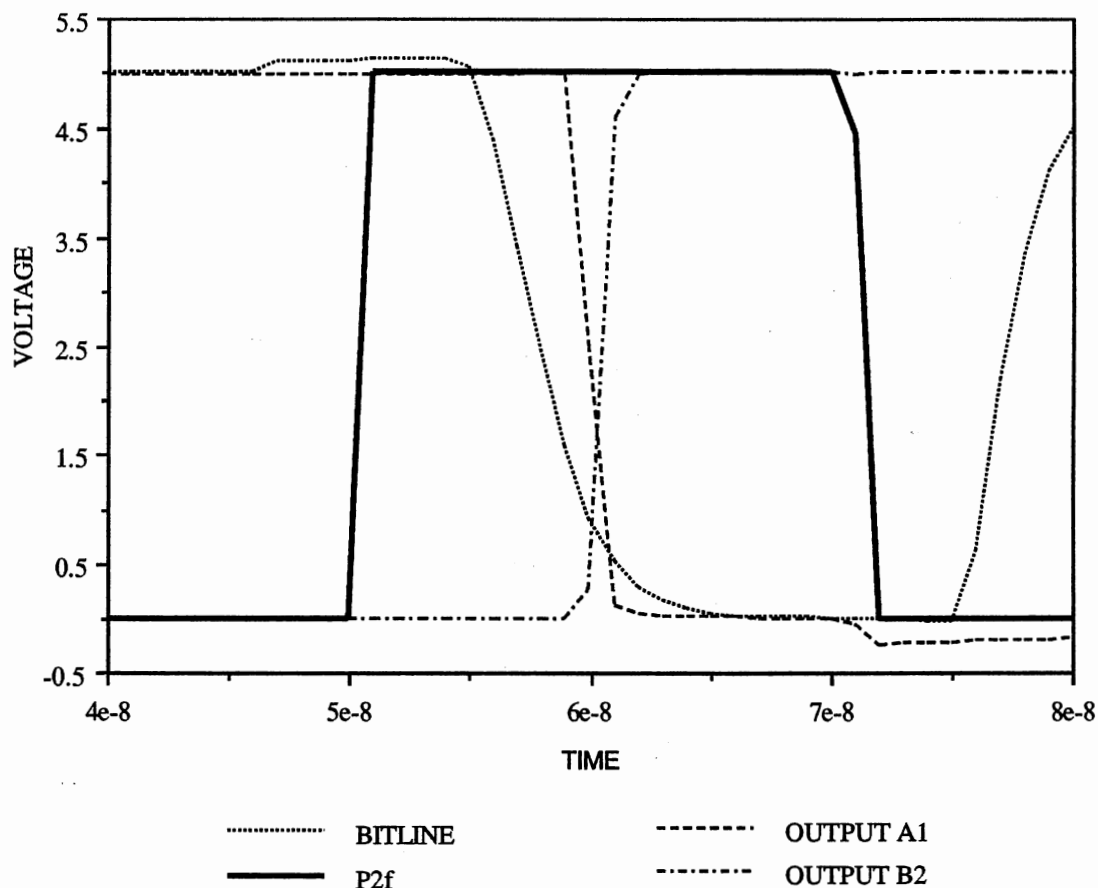


Figure 31. Read Delay

Transfer Delay

The twin register file has the capability of transferring data from one register file to another. The transfer operation takes place during the phase when the bitlines are being precharged (P1s•P1f). The transfer delay depends on the time required to activate the transfer lines and the time required to change one of the storage nodes from *low* to *high*.

Figure 32 shows the delay involved in transferring *high* from Register A to Register B. The transfer line TA2B takes around 4ns to be charged up. Once the transfer line is activated, the *high* stored at node SA is transferred to node SB. Since the inverters have a weak pull-up transistor, the node SB does not change its state rapidly. As soon as the node

SB reaches a voltage level of around 1.4V, the inverter I4 starts changing its state. Since the inverters have strong pull-down transistors, the node SB_BAR rapidly changes its state to *low*. As seen in Figure 32, node SB settles to *low* within 8ns. This is a stable state and we could turn off the transfer lines at this time because the *low* stored at SB_BAR is going to pull node SB to *high*.

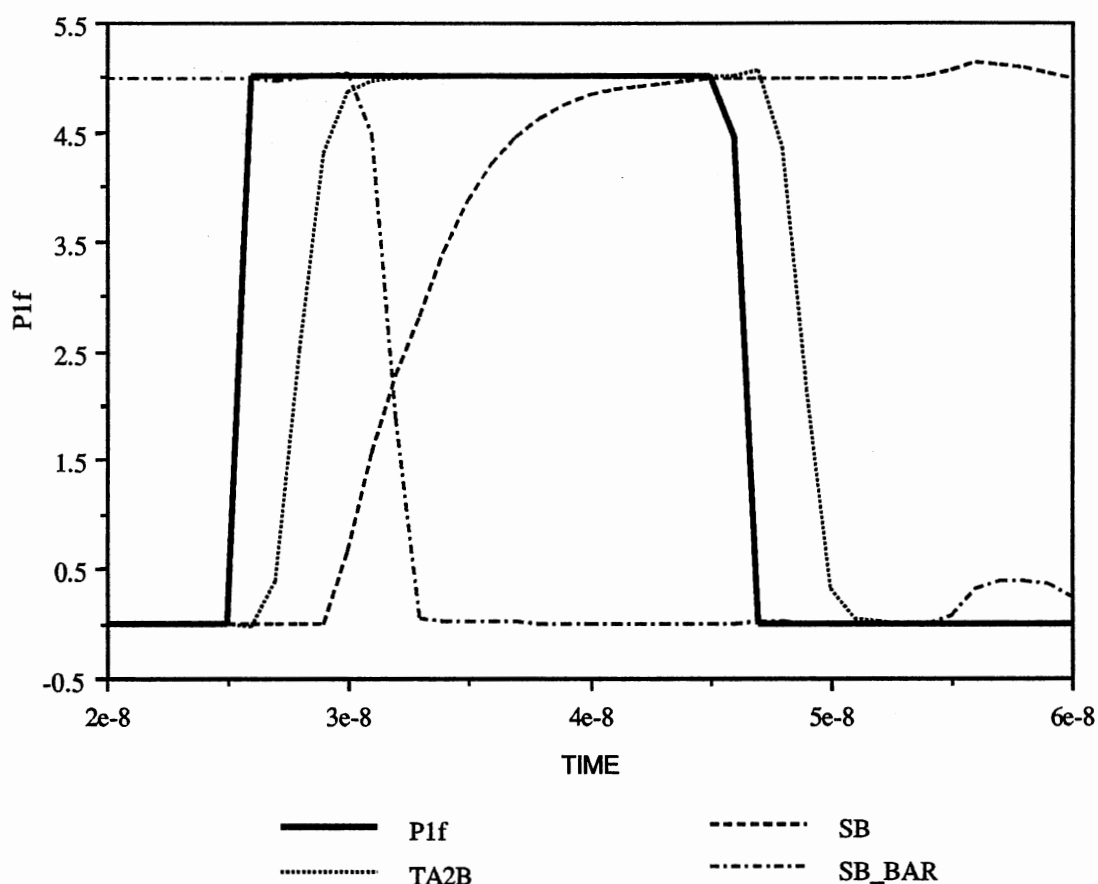


Figure 32. Transferring 'high'

Figure 33 shows the delay involved in transferring *low* from register A to register B. In this case the *low* stored at node SA is transferred to SB. Since the inverters have strong

pull-down transistors, node SB rapidly settles to *low* in 6ns. The *low* at SB starts pulling up the node SB_BAR, and we could turn off the transfer lines at around 10ns. The *low* at node SB is eventually going to pull the node SB_BAR to *high*.

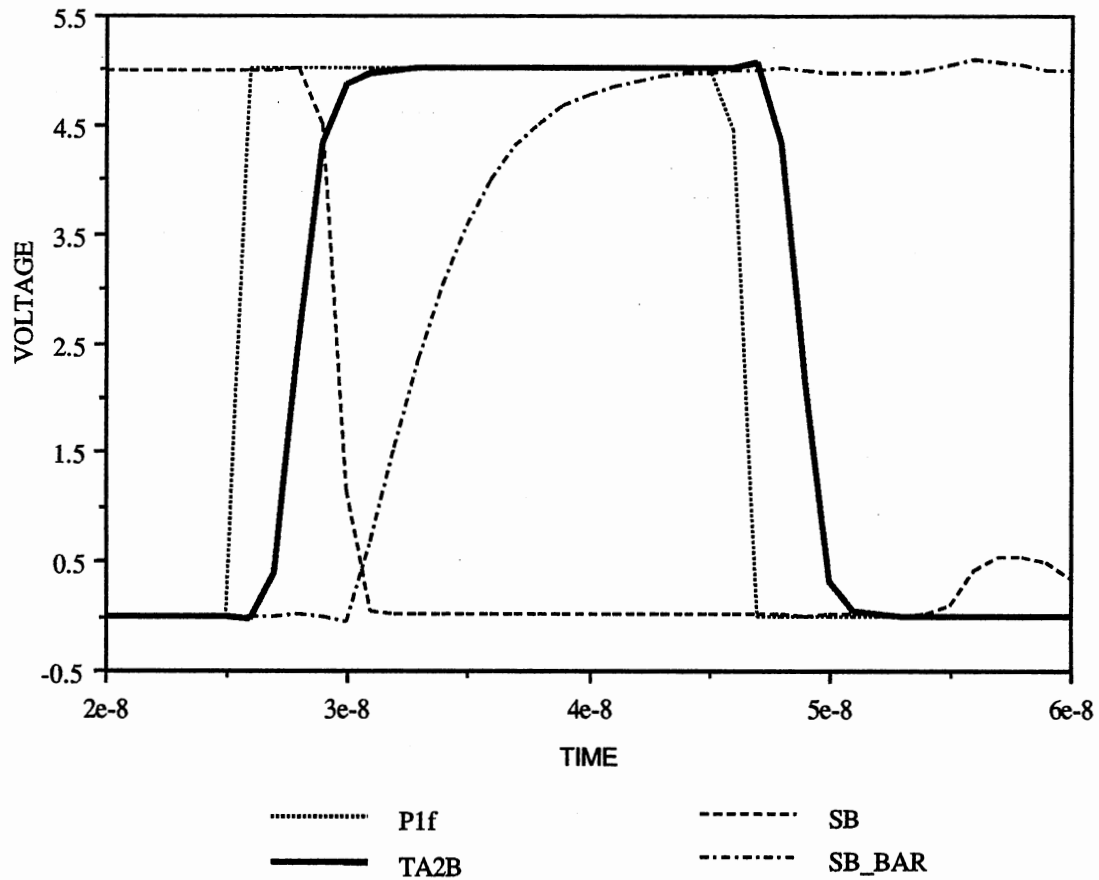


Figure 33. Transferring 'low'

CHAPTER VI

CONCLUSIONS

This thesis presents the design of a novel twin register file which can be used for reducing the effects of conditional branches in a pipelined architecture. The twin register file consists of two files capable of executing the 'read', 'write', and 'transfer' operation. Each of the two register files has two read ports and one write port. We can write one word and read two words from each file every clock cycle. Each register file has the capability of copying all of its contents into the other register file. This transfer operation can be accomplished in a single phase of the clock, and does not involve any extra penalties in terms of time because it is overlapped with the precharge period of the bitlines.

A parameterized twin register file is designed using layout tools Magic and Lager. The functionality of a 32-word-by-16-bit twin register file is verified by simulating it on IRSIM. The delays associated with the 'read', 'write', and 'transfer' operations are estimated by simulating the register file on SPICE. The speed of the register files is limited by the delay associated with the read operation, which is about 11ns.

This thesis discusses the effects of conditional branches and reviews the conventional schemes which have been used for reducing the branch penalties. The thesis proposes a twin processor architecture which processes both paths of a conditional branch simultaneously. The advantage of such a machine is that it does not have to stall, nor does it have to make any predictions about the path to be taken. Hence, there is no need to flush the results of an incorrect prediction. In the twin processor machine, the twin register file is required for executing two instruction streams of a branch instruction. The contents of one register file have to be copied into the other on detection of a conditional branch instruction.

This ensures that the two processors use the same data when they begin executing the two instruction streams. The special purpose twin register file copies data from one file to another without incurring any extra penalties in terms of time.

The twin processor architecture along with the twin register file provides an alternative to the conventional schemes employed for reducing the branch penalties. The twin register file can also be used in other applications where a backup of the local memory is required. Subroutine calls/returns are an example of possible applications which could use the twin register file for a backup of local registers. Further research into the application of twin register files is recommended.

BIBLIOGRAPHY

- [1] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *Computing Surveys*, Vol.9, pp. 61-102, 1977.
- [2] H. S. Stone, *High Performance Computer Architectures*, Addison-Wesley Publishing Company, 1987.
- [3] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," *Computer*, Vol.21, pt.2, pp. 47-54, July 1988.
- [4] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, Vol.21, pt.2, pp. 1404-1411, Dec. 1972.
- [5] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," *Computer Architecture News*, Vol.14, no. 2, pp. 396-403, June 1986.
- [6] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, Vol.19, no.10, pp. 889-895, Oct. 1970.
- [7] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Vol.17, pp. 6-22, Jan. 1984.
- [8] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. of the 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.
- [9] W. W. Hwu, T. M. Conte and P. P. Cheny, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches," *Proc. of 16th Intl. Symposium on Computer Architecture*, pp. 224-231, June 1989.
- [10] J. A. DeRosa and M. M. Levy, "An Evaluation of Branch Architectures," *Proc. of 14th Intl. Symposium on Computer Architecture*, pp. 10-16, June 1987.

- [11] D. R. Ditzel and H. R. McLellan, "Branch Folding in CRISP Microprocessor: Reducing Branch Delay to Zero," *Proc. of 14th Intl Symposium on Computer Architecture*, pp. 2-9, June 1987.
- [12] W. D. Connors, J. H. Florkowski and S. K. Patton, "The IBM 3033: An Inside Look," *Datamation*, pp. 198-218, May 1979.
- [13] H. S. Stone, "A Pipeline Pushdown-Stack Computer," *Parallel Processor Systems, Technologies and Applications*, L.C.Hobbs, Ed. Washington D.C.: Spartan, pp. 235-249, 1970.
- [14] M. D. Smith, M. Johnson and M. A. Horowitz, "Limits on Multiple Instruction Issue," *Computer Architecture News*, pp. 290-302, April 1989.
- [15] D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development*, Vol.11, pp. 8-24, Jan. 1967.
- [16] G. S. Sohi and S. Vajapeyam, "Instruction Issue Logic For High Performance, Interruptable Pipelined Processors," *Proc. of 14th Intl Symposium on Computer Architecture*, pp. 27-34, June 1987.
- [17] W. W. Hwu and Y. N. Patt, "CheckPoint Repair For Out-Of-Order Execution Machines," *Proc. of 14th Intl Symposium on Computer Architecture*, pp. 18-26, June 1987.
- [18] R. W. Sherburne, M. G. H. Katenevis, D. A. Patterson and C. H. Sequin, "A 32-Bit NMOS Microprocessor with a Large Register File," *IEEE Journal of Solid State Ckts.*, Vol.SC-19, no. 5, pp. 682-689, October 1984.
- [19] R. W. Sherburne, "Processor Design Tradeoffs In VLSI," *Doctoral Dissertation*, University of California, Berkeley, 1984.
- [20] R. D. Jolly, "A 9-ns, 1.4 Gigabyte/S, 17-Ported CMOS Register File," *IEEE Journal of Solid State Ckts.*, Vol. 26, no. 10, pp. 1407-1412, October 1991.

- [21] J. Beyers, L. Dohse, J. Fucetola, R. Kochis, C. Lob, G. Taylor and E. Zeller, "A 32 Bit VLSI CPU Chip," *IEEE Journal of Solid State Ckts.*, Vol. SC-16, no. 5, pp. 537-541, October 1981.
- [22] R. W. Sherburne, M. G. H. Katevenis, D. A. Patterson and C. H. Sequin, "Datapath Design for RISC," *Proceedings of the Conference on Advanced Research in VLSI*, Massachusetts Institute of Technology, pp. 53-62, January 1982.
- [23] J. L. Rosenfeld and R. D. Villani, "Micromultiprocessing : An Approach to Multiprocessing at the Level of Very Small Tasks," *IEEE Transactions on Computers*, Vol. 22, no. 2, pp. 149-153, Feb. 1973.
- [24] M. G. H. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI," *Doctoral Dissertation*, Computer Science Division, University of California, Berkeley, 1983.
- [25] N. Weste and K. Eshragian, *Principles of CMOS VLSI Design: A System Perspective*, Addison Wesley Publishing Company, 1985.
- [26] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison Wesley Publishing Company, 1985.
- [27] L. G. Johnson, *Private Communication*, Oct. 1992.

APPENDIXES

APPENDIX A

TIMLAGER C FILE FOR TWIN REGISTER FILE

This appendix contains the tiling procedure used for laying out the Twin Register File. TimLager functions were used to write this tiling procedure.

```
/* TimLager file for scalable twin register file
Author: Manish Shah */

#include "TimLager.h"
    int width, words;
    int i, j, k;

/* th_sram is the parameterized twin register file */

th_sram()
{
    width = Getparam("width");
    if (width>=32) width=32;           /* ensures that maximum width is 32 */
    words = Getparam("words");
    if (words >=256) words=256;      /*ensures that maximum number of
words is 256 */

/* Generate the Basic Blocks */

    th_array();
    th_rwA();
    th_rwB();
    th_bufferA();
    th_bufferB();
```

```
/* Tile Blocks */
```

```

Open_newcell(Read("name"));
Addup("th_controlB2", LEFT|BOTTOM, MX, MY, END);
Addright("th_bufferB", BOTTOM, MY, END);
Addright("th_controlA2", RIGHT|BOTTOM, MY, END);
Addup("th_rwB", LEFT, MX, END);
Addright("th_array", NONE, END);
Addright("th_rwA", RIGHT, END);
Addup("th_controlB1", LEFT|TOP, MX, END);
Addright("th_bufferA", TOP, END);
Addright("th_controlA1", RIGHT|TOP, END);
Close_newcell();
}

```

/* th_array() generates the array of memory cells. th_4cells are the basic memory cells which have four Twin Register File memory cells.*/

```

th_array()
{
    int i, j;
    Open_newcell("th_array");
    Addup("th_4cells", NONE, END);
    for(i=1; i<words/4; i++)
        Addright("th_4cells", NONE, OFFSETX, -12, END);
    for(j=1; j<width; j++)
    {
        Addup("th_4cells", NONE, OFFSETY, -9, END);
        for(i=1; i<words/4; i++)
            Addright("th_4cells", NONE, OFFSETX, -12, END);
    }
    Close_newcell();
}

```

/* th_rwA() adds the read/write circuits for register file A */

```

th_rwA()
{
    Open_newcell("th_rwA");
    Addup("th_dst2a", RIGHT, RIGHTINDEX, 0, END);
    for(i=1; i<width; i++)
        Addup("th_dst2a", RIGHT, RIGHTINDEX, i, OFFSETY, -9, END);
    Close_newcell();
}

```

/* th_rwB() adds the read/write circuits for register file B*/

```
th_rwB()
{
  Open_newcell("th_rwB");
  Addup("th_dst2b", RIGHT, RIGHTINDEX, 0, END);
  for(i=1; i<width; i++)
    Addup("th_dst2b", RIGHT, RIGHTINDEX, i, OFFSETY, -9, END);
  Close_newcell();
}
```

/* th_bufferA() adds the driver cells for port 1 of register file A and register file B. It also places the parameterized decoder for port 1 of register file A and register file B at the top of the drivers. */

```
th_bufferA()
{
  int i,j;
  Open_newcell("th_bufferA");
  Addup("th_drvcellA", NONE, END);
  for(i=1; i<words/4; i++)
    Addright("th_drvcellA", NONE, END);
```

/* the following procedure places the decoder cells */

```
for(j=0; j<7; j++)
{
  Addup("th_dec.0", NONE, END);
  for(i=1; i<words/2; i++)
  {
    if(i/2*2==i)
      Addright((i & (1<<j)) ? "th_decA.1" : "th_dec.0", NONE, END);
    else
      Addright((i & (1<<j)) ? "th_decB.1" : "th_dec.0", NONE, END);
  }
  Addup("th_decA.1", NONE, END);
  for(i=1, i<words/2; i++)
  {
    if(i/2*2==i)
      Addright((i & (1<<j)) ? "th_dec.0" : "th_decA.1", NONE, END);
    else
      Addright((i & (1<<j)) ? "th_dec.0" : "th_decB.1", NONE, END);
  }
}
Addup("th_dec.top", TOP, END);
for(i=1; i<words/4; i++)
  Addright("th_dec.top", TOP, END);
Close_newcell();
}
```


/* th_bufferB() adds the driver cells for port 2 of register file A and register file B. It also places the parameterized decoder for port 2 of register file A and register file B at the top of the drivers. */

```

th_bufferB()
{
  int i, j;
  Open_newcell("th_bufferB");
  Addup("th_drvcellB", NONE, END);
  for(i=1; i<words/4; i++)
    Addright("th_drvcellB", NONE, END);

  /* the following procedure places the decoder cells */

  for(j=0; j<7; j++)
  {
    Addup("th_dec.0", NONE, END);
    for(i=1; i<words/2; i++)
    {
      if(i/2*2==i)
        Addright((i & (1<<j)) ? "th_decA.1" : "th_dec.0", NONE, END);
      else
        Addright((i & (1<<j)) ? "th_decB.1" : "th_dec.0", NONE, END);
    }
    Addup("th_decA.1", NONE, END);
    for(i=1; i<words/2; i++)
    {
      if(i/2*2==i)
        Addright((i & (1<<j)) ? "th_dec.0" : "th_decA.1", NONE, END);
      else
        Addright((i & (1<<j)) ? "th_dec.0" : "th_decB.1", NONE, END);
    }
  }
  Addup("th_dec.top", TOP, END);
  for(i=1; i<words/4; i++)
    Addright("th_dec.top", TOP, END);
  Close_newcell();
}

```

APPENDIX B

SPICE FILE

This appendix contains an example spice circuit file which is used for simulating the transfer operation. The spice deck obtained by extracting the test circuit and the spice model for the transistors are also included in this file.

SIMULATION OF TRANSFER TIME FOR TWIN REGISTER FILE

```
*
V1 1 0 dc 5v
V4 4 0 dc 0v
v24 24 0 dc 5v
*
Vp1f 29 0 pulse(0 5 25.0ns 0.9ns 0.9ns 20ns 50ns)
Vp1fb 27 0 pulse(5 0 25.0ns 0.9ns 0.9ns 20ns 50ns)
Vp2f 63 0 pulse(0 5 0.0ns 0.9ns 0.9ns 20ns 50ns)
Vp2fb 5 0 pulse(5 0 0.0ns 0.9ns 0.9ns 20ns 50ns)
Vp2s 69 0 pwl(0 5 20.9ns 5v 21.8ns 0)
Vp2sb 57 0 pwl(0 0 20.9ns 0 21.8ns 5)
Vp1s 39 0 pulse(0 5 25.0ns 0.9ns 0.9ns 45ns 100ns)
Vp1sb 45 0 pulse(5 0 25.0ns 0.9ns 0.9ns 45ns 100ns)
*
Vada10 83 0 pwl(0 5 25ns 5v 26.5ns 0v)
Vada11 22 0 pwl(0 5 .5ns 0v 50ns 0v 51.5ns 5v)
Vada12 20 0 dc 0v
Vada13 18 0 dc 0v
Vada14 16 0 dc 0v
Vada15 14 0 dc 0v
Vada16 12 0 dc 0v
Vada17 9 0 dc 0v
Vada20 77 0 pwl(0 5 25ns 5v 26.5ns 0v)
Vada21 131 0 pwl(0 5 1.5ns 0v 50ns 0v 51.5ns 5v)
Vada22 135 0 dc 0v
Vada23 137 0 dc 0v
Vada24 139 0 dc 0v
Vada25 141 0 dc 0v
Vada26 143 0 dc 0v
Vada27 145 0 dc 0v
Vadb10 108 0 pwl(0 5 25ns 5v 26.5ns 0v)
Vadb11 23 0 pwl(0 5 1.5ns 0v 50ns 0v 51.5ns 5v)
Vadb12 21 0 dc 0v
Vadb13 19 0 dc 0v
```

```

Vadb14 17 0 dc 0v
Vadb15 15 0 dc 0v
Vadb16 13 0 dc 0v
Vadb17 11 0 dc 0v
Vadb20 74 0 pwl(0 5 25ns 5v 26.5ns 0v)
Vadb21 133 0 pwl(0 5 1.5ns 0v 50ns 0v 51.5ns 5v)
Vadb22 136 0 dc 0v
Vadb23 138 0 dc 0v
Vadb24 140 0 dc 0v
Vadb25 142 0 dc 0v
Vadb26 144 0 dc 0v
Vadb27 146 0 dc 0v
*
Vada1-0 60 0 pwl(0 0 25ns 0v 26.5ns 5v)
Vada1-1 162 0 pwl(0 0 1.5ns 5v 50ns 5v 51.5ns 0v)
Vada1-2 164 0 dc 5v
Vada1-3 166 0 dc 5v
Vada1-4 168 0 dc 5v
Vada1-5 170 0 dc 5v
Vada1-6 172 0 dc 5v
Vada1-7 174 0 dc 5v
Vada2-0 93 0 pwl(0 0 25ns 0v 26.5ns 5v)
Vada2-1 159 0 pwl(0 0 1.5ns 5v 50ns 5v 51.5ns 0v)
Vada2-2 157 0 dc 5v
Vada2-3 155 0 dc 5v
Vada2-4 153 0 dc 5v
Vada2-5 151 0 dc 5v
Vada2-6 149 0 dc 5v
Vada2-7 147 0 dc 5v
Vadb1-0 115 0 pwl(0 0 25ns 0v 26.5ns 5v)
Vadb1-1 161 0 pwl(0 0 1.5ns 5v 50ns 5v 51.5ns 0v)
Vadb1-2 163 0 dc 5v
Vadb1-3 165 0 dc 5v
Vadb1-4 167 0 dc 5v
Vadb1-5 169 0 dc 5v
Vadb1-6 171 0 dc 5v
Vadb1-7 173 0 dc 5v
Vadb2-0 88 0 pwl(0 0 25ns 0v 26.5ns 5v)
Vadb2-1 160 0 pwl(0 0 1.5ns 5v 50ns 5v 51.5ns 0v)
Vadb2-2 158 0 dc 5v
Vadb2-3 156 0 dc 5v
Vadb2-4 154 0 dc 5v
Vadb2-5 152 0 dc 5v
Vadb2-6 150 0 dc 5v
Vadb2-7 148 0 dc 5v
*
Vreg1 35 0 dc 0v
Vta2b 48 0 pwl(0 0 25ns 0 25.9ns 5 70ns 5 70.9ns 0)
Vtb2a 118 0 dc 0v
*
Vina 85 0 pwl(0 0 25ns 0v 25.5ns 5v)
Vinb 79 0 pwl(0 5 25ns 5v 25.5ns 0v)
*
.ic V(61)=4.5V v(62)=0.0v v(68)=0.0 v(90)=4.5v

```

```
.ic v(43)=0 v(65)=0 v(45)=0 v(104)=0
.ic v(46)=5v v(92)=5v v(47)=0v v(97)=0v
.ic v(48)=0v v(118)=0v
.ic v(28)=5v v(30)=5v v(124)=5v v(125)=5v
```

*

*** SPICE DECK created from test2.sim, tech=scmos

*

```
M1 6 5 0 4 CMOSN L=2.0U W=4.0U
M2 7 5 0 4 CMOSN L=2.0U W=4.0U
M3 6 9 8 4 CMOSN L=2.0U W=6.0U
M4 7 11 10 4 CMOSN L=2.0U W=6.0U
M5 6 12 8 4 CMOSN L=2.0U W=6.0U
M6 7 13 10 4 CMOSN L=2.0U W=6.0U
M7 6 14 8 4 CMOSN L=2.0U W=6.0U
M8 7 15 10 4 CMOSN L=2.0U W=6.0U
M9 6 16 8 4 CMOSN L=2.0U W=6.0U
M10 7 17 10 4 CMOSN L=2.0U W=6.0U
M11 6 18 8 4 CMOSN L=2.0U W=6.0U
M12 7 19 10 4 CMOSN L=2.0U W=6.0U
M13 6 20 8 4 CMOSN L=2.0U W=6.0U
M14 7 21 10 4 CMOSN L=2.0U W=6.0U
M15 6 22 8 4 CMOSN L=2.0U W=6.0U
M16 7 23 10 4 CMOSN L=2.0U W=6.0U
M17 1 5 8 24 CMOSN L=2.0U W=5.0U
M18 10 5 1 24 CMOSN L=2.0U W=5.0U
M19 25 8 1 24 CMOSN L=2.0U W=5.0U
M20 0 8 25 4 CMOSN L=2.0U W=5.0U
M21 26 10 0 4 CMOSN L=2.0U W=5.0U
M22 1 10 26 24 CMOSN L=2.0U W=5.0U
M23 28 27 25 24 CMOSN L=2.0U W=5.0U
M24 25 29 28 4 CMOSN L=2.0U W=5.0U
M25 30 29 26 4 CMOSN L=2.0U W=5.0U
M26 26 27 30 24 CMOSN L=2.0U W=5.0U
M27 31 5 1 24 CMOSN L=2.0U W=5.0U
M28 32 28 31 24 CMOSN L=2.0U W=5.0U
M29 0 5 32 4 CMOSN L=2.0U W=5.0U
M30 33 5 0 4 CMOSN L=2.0U W=5.0U
M31 34 30 33 24 CMOSN L=2.0U W=5.0U
M32 1 5 34 24 CMOSN L=2.0U W=5.0U
M33 0 35 32 4 CMOSN L=2.0U W=5.0U
M34 33 35 0 4 CMOSN L=2.0U W=5.0U
M35 36 32 1 24 CMOSN L=2.0U W=15.0U
M36 0 32 36 4 CMOSN L=2.0U W=10.0U
M37 37 33 0 4 CMOSN L=2.0U W=10.0U
M38 1 33 37 24 CMOSN L=2.0U W=15.0U
M39 40 39 38 4 CMOSN L=2.0U W=6.0U
M40 0 41 40 4 CMOSN L=2.0U W=7.0U
M41 1 41 42 24 CMOSN L=2.0U W=4.0U
M42 43 36 1 24 CMOSN L=2.0U W=65.0U
M43 0 36 43 4 CMOSN L=2.0U W=25.0U
M44 44 37 0 4 CMOSN L=2.0U W=25.0U
M45 42 45 38 24 CMOSN L=2.0U W=4.0U
M46 1 29 46 24 CMOSN L=2.0U W=44.0U
```

M47 47 46 1 24 CMOSP L=2.0U W=44.0U
M48 49 48 46 4 CMOSN L=2.0U W=27.0U
M49 0 29 49 4 CMOSN L=2.0U W=27.0U
M50 1 37 44 24 CMOSP L=2.0U W=65.0U
M51 51 50 0 4 CMOSN L=2.0U W=7.0U
M52 52 39 51 4 CMOSN L=2.0U W=6.0U
M53 53 50 1 24 CMOSP L=2.0U W=4.0U
M54 52 45 53 24 CMOSP L=2.0U W=4.0U
M55 0 46 47 4 CMOSN L=2.0U W=18.0U
M56 55 54 0 4 CMOSN L=2.0U W=31.0U
M57 58 57 56 24 CMOSP L=2.0U W=42.0U
M58 56 60 59 4 CMOSN L=2.0U W=18.0U
M59 62 61 0 4 CMOSN L=2.0U W=22.0U
M60 64 63 55 4 CMOSN L=2.0U W=31.0U
M61 66 65 62 4 CMOSN L=2.0U W=8.0U
M62 68 67 0 4 CMOSN L=2.0U W=22.0U
M63 56 69 64 4 CMOSN L=2.0U W=31.0U
M64 1 54 58 24 CMOSP L=2.0U W=42.0U
M65 1 71 70 24 CMOSP L=2.0U W=10.0U
M66 72 45 71 24 CMOSP L=2.0U W=4.0U
M67 1 73 72 24 CMOSP L=2.0U W=4.0U
M68 75 74 41 24 CMOSP L=2.0U W=10.0U
M69 1 59 75 24 CMOSP L=2.0U W=10.0U
M70 0 71 70 4 CMOSN L=2.0U W=3.0U
M71 76 39 71 4 CMOSN L=2.0U W=6.0U
M72 0 73 76 4 CMOSN L=2.0U W=7.0U
M73 78 77 73 24 CMOSP L=2.0U W=10.0U
M74 54 79 1 24 CMOSP L=2.0U W=12.0U
M75 1 80 78 24 CMOSP L=2.0U W=10.0U
M76 81 79 1 24 CMOSP L=2.0U W=42.0U
M77 82 57 81 24 CMOSP L=2.0U W=42.0U
M78 56 83 59 24 CMOSP L=2.0U W=24.0U
M79 1 27 59 24 CMOSP L=2.0U W=20.0U
M80 59 44 68 4 CMOSN L=2.0U W=8.0U
M81 84 60 66 4 CMOSN L=2.0U W=18.0U
M82 86 85 0 4 CMOSN L=2.0U W=31.0U
M83 87 63 86 4 CMOSN L=2.0U W=31.0U
M84 84 69 87 4 CMOSN L=2.0U W=31.0U
M85 67 47 62 4 CMOSN L=2.0U W=4.0U
M86 1 61 62 24 CMOSP L=3.0U W=3.0U
M87 80 83 82 24 CMOSP L=2.0U W=24.0U
M88 89 88 41 4 CMOSN L=2.0U W=3.0U
M89 0 59 89 4 CMOSN L=2.0U W=3.0U
M90 90 46 67 4 CMOSN L=2.0U W=6.0U
M91 61 91 1 24 CMOSP L=3.0U W=3.0U
M92 1 67 68 24 CMOSP L=3.0U W=3.0U
M93 91 92 62 4 CMOSN L=2.0U W=6.0U
M94 80 27 1 24 CMOSP L=2.0U W=20.0U
M95 94 93 73 4 CMOSN L=2.0U W=3.0U
M96 0 80 94 4 CMOSN L=2.0U W=3.0U
M97 95 69 82 4 CMOSN L=2.0U W=31.0U
M98 1 27 66 24 CMOSP L=2.0U W=20.0U
M99 84 83 66 24 CMOSP L=2.0U W=24.0U
M100 96 57 84 24 CMOSP L=2.0U W=42.0U

M101 90 68 1 24 CMOS L=3.0U W=3.0U
M102 90 97 91 4 CMOSN L=2.0U W=4.0U
M103 98 63 95 4 CMOSN L=2.0U W=31.0U
M104 0 79 54 4 CMOSN L=2.0U W=6.0U
M105 80 60 82 4 CMOSN L=2.0U W=18.0U
M106 0 79 98 4 CMOSN L=2.0U W=31.0U
M107 61 43 99 4 CMOSN L=2.0U W=8.0U
M108 0 91 61 4 CMOSN L=2.0U W=24.0U
M109 99 83 100 24 CMOS L=2.0U W=24.0U
M110 1 85 96 24 CMOS L=2.0U W=42.0U
M111 101 85 1 24 CMOS L=2.0U W=12.0U
M112 99 27 1 24 CMOS L=2.0U W=20.0U
M113 102 101 1 24 CMOS L=2.0U W=42.0U
M114 100 57 102 24 CMOS L=2.0U W=42.0U
M115 103 69 100 4 CMOSN L=2.0U W=31.0U
M116 90 104 80 4 CMOSN L=2.0U W=8.0U
M117 0 68 90 4 CMOSN L=2.0U W=24.0U
M118 99 60 100 4 CMOSN L=2.0U W=18.0U
M119 105 63 103 4 CMOSN L=2.0U W=31.0U
M120 0 85 101 4 CMOSN L=2.0U W=6.0U
M121 106 99 1 24 CMOS L=2.0U W=10.0U
M122 50 83 106 24 CMOS L=2.0U W=10.0U
M123 107 66 1 24 CMOS L=2.0U W=10.0U
M124 109 108 107 24 CMOS L=2.0U W=10.0U
M125 111 45 110 24 CMOS L=2.0U W=4.0U
M126 110 109 1 24 CMOS L=2.0U W=4.0U
M127 112 111 1 24 CMOS L=2.0U W=10.0U
M128 0 101 105 4 CMOSN L=2.0U W=31.0U
M129 113 99 0 4 CMOSN L=2.0U W=3.0U
M130 50 60 113 4 CMOSN L=2.0U W=3.0U
M131 114 66 0 4 CMOSN L=2.0U W=3.0U
M132 109 115 114 4 CMOSN L=2.0U W=3.0U
M133 116 109 0 4 CMOSN L=2.0U W=7.0U
M134 111 39 116 4 CMOSN L=2.0U W=6.0U
M135 112 111 0 4 CMOSN L=2.0U W=3.0U
M136 65 117 1 24 CMOS L=2.0U W=65.0U
M137 1 29 92 24 CMOS L=2.0U W=44.0U
M138 97 92 1 24 CMOS L=2.0U W=44.0U
M139 0 92 97 4 CMOSN L=2.0U W=18.0U
M140 119 118 92 4 CMOSN L=2.0U W=27.0U
M141 0 29 119 4 CMOSN L=2.0U W=27.0U
M142 0 117 65 4 CMOSN L=2.0U W=25.0U
M143 104 120 0 4 CMOSN L=2.0U W=25.0U
M144 1 120 104 24 CMOS L=2.0U W=65.0U
M145 117 121 1 24 CMOS L=2.0U W=15.0U
M146 0 121 117 4 CMOSN L=2.0U W=10.0U
M147 120 122 0 4 CMOSN L=2.0U W=10.0U
M148 1 122 120 24 CMOS L=2.0U W=15.0U
M149 0 35 121 4 CMOSN L=2.0U W=5.0U
M150 122 35 0 4 CMOSN L=2.0U W=5.0U
M151 123 5 1 24 CMOS L=2.0U W=5.0U
M152 121 124 123 24 CMOS L=2.0U W=5.0U
M153 0 5 121 4 CMOSN L=2.0U W=5.0U
M154 122 5 0 4 CMOSN L=2.0U W=5.0U

M155 126 125 122 24 CMOS L=2.0U W=5.0U
M156 1 5 126 24 CMOS L=2.0U W=5.0U
M157 124 27 127 24 CMOS L=2.0U W=5.0U
M158 127 29 124 4 CMOSN L=2.0U W=5.0U
M159 125 29 128 4 CMOSN L=2.0U W=5.0U
M160 128 27 125 24 CMOS L=2.0U W=5.0U
M161 127 129 1 24 CMOS L=2.0U W=5.0U
M162 0 129 127 4 CMOSN L=2.0U W=5.0U
M163 128 130 0 4 CMOSN L=2.0U W=5.0U
M164 1 130 128 24 CMOS L=2.0U W=5.0U
M165 1 63 129 24 CMOS L=2.0U W=5.0U
M166 130 63 1 24 CMOS L=2.0U W=5.0U
M167 132 131 129 4 CMOSN L=2.0U W=6.0U
M168 134 133 130 4 CMOSN L=2.0U W=6.0U
M169 132 135 129 4 CMOSN L=2.0U W=6.0U
M170 134 136 130 4 CMOSN L=2.0U W=6.0U
M171 132 137 129 4 CMOSN L=2.0U W=6.0U
M172 134 138 130 4 CMOSN L=2.0U W=6.0U
M173 132 139 129 4 CMOSN L=2.0U W=6.0U
M174 134 140 130 4 CMOSN L=2.0U W=6.0U
M175 132 141 129 4 CMOSN L=2.0U W=6.0U
M176 134 142 130 4 CMOSN L=2.0U W=6.0U
M177 132 143 129 4 CMOSN L=2.0U W=6.0U
M178 134 144 130 4 CMOSN L=2.0U W=6.0U
M179 132 145 129 4 CMOSN L=2.0U W=6.0U
M180 134 146 130 4 CMOSN L=2.0U W=6.0U
M181 132 5 0 4 CMOSN L=2.0U W=4.0U
M182 134 5 0 4 CMOSN L=2.0U W=4.0U

*

C183 57 1 4.0F
C184 44 0 2.0F
C185 99 0 5.0F
C186 65 1 2.0F
C187 45 0 4.0F
C188 57 0 4.0F
C189 65 0 2.0F
C190 1 60 3.0F
C191 1 120 2.0F
C192 25 29 2.0F
C193 60 0 4.0F
C194 127 29 2.0F
C195 1 124 2.0F
C196 8 0 2.0F
C197 30 1 2.0F
C198 122 5 3.0F
C199 82 63 2.0F
C200 37 1 3.0F
C201 26 0 6.0F
C202 128 0 6.0F
C203 43 1 2.0F
C204 1 0 31.0F
C205 57 79 2.0F
C206 39 1 2.0F
C207 47 90 2.0F

C208 54 79 2.0F
C209 57 85 2.0F
C210 43 0 2.0F
C211 59 1 5.0F
C212 56 63 2.0F
C213 27 66 2.0F
C214 39 0 4.0F
C215 108 0 2.0F
C216 59 0 4.0F
C217 62 97 2.0F
C218 1 117 2.0F
C219 74 0 2.0F
C220 27 80 2.0F
C221 115 0 2.0F
C222 77 0 2.0F
C223 83 60 2.0F
C224 28 1 2.0F
C225 84 63 2.0F
C226 121 5 3.0F
C227 35 0 4.0F
C228 88 0 2.0F
C229 36 1 3.0F
C230 25 0 6.0F
C231 127 0 6.0F
C232 27 99 2.0F
C233 1 5 4.0F
C234 33 5 3.0F
C235 104 1 2.0F
C236 83 1 3.0F
C237 69 1 4.0F
C238 0 5 6.0F
C239 104 0 2.0F
C240 83 0 4.0F
C241 66 1 5.0F
C242 69 0 4.0F
C243 61 43 3.0F
C244 66 0 5.0F
C245 68 44 2.0F
C246 26 29 2.0F
C247 128 29 2.0F
C248 1 125 2.0F
C249 80 1 4.0F
C250 27 1 4.0F
C251 63 1 4.0F
C252 80 0 4.0F
C253 79 5 2.0F
C254 41 1 2.0F
C255 27 0 4.0F
C256 85 5 2.0F
C257 129 0 2.0F
C258 44 1 2.0F
C259 32 5 3.0F
C260 99 1 4.0F
C261 63 0 4.0F

C262 93 0 2.0F
C263 100 63 2.0F
C264 104 90 2.0F
C265 27 59 2.0F
C266 45 1 4.0F
C267 147 0 5.0F
C268 148 0 5.0F
C269 149 0 5.0F
C270 150 0 5.0F
C271 151 0 5.0F
C272 152 0 5.0F
C273 153 0 5.0F
C274 154 0 5.0F
C275 155 0 5.0F
C276 156 0 5.0F
C277 157 0 5.0F
C278 158 0 5.0F
C279 159 0 5.0F
C280 160 0 5.0F
C281 161 0 5.0F
C282 162 0 5.0F
C283 163 0 5.0F
C284 164 0 5.0F
C285 165 0 5.0F
C286 166 0 5.0F
C287 167 0 5.0F
C288 168 0 5.0F
C289 169 0 5.0F
C290 170 0 5.0F
C291 171 0 5.0F
C292 172 0 5.0F
C293 173 0 5.0F
C294 174 0 5.0F
C295 0 0 1848.0F
C296 5 0 164.0F
C297 146 0 12.0F
C298 145 0 11.0F
C299 144 0 12.0F
C300 143 0 11.0F
C301 142 0 12.0F
C302 141 0 11.0F
C303 140 0 12.0F
C304 139 0 11.0F
C305 138 0 12.0F
C306 137 0 11.0F
C307 136 0 12.0F
C308 135 0 11.0F
C309 134 0 129.0F
C310 132 0 129.0F
C311 133 0 12.0F
C312 131 0 11.0F
C313 63 0 62.0F
C314 1 0 2304.0F
C315 130 0 145.0F

C316 129 0 145.0F
C317 128 0 61.0F
C318 127 0 60.0F
C319 29 0 92.0F
C320 126 0 4.0F
C321 123 0 4.0F
C322 125 0 33.0F
C323 124 0 33.0F
C324 35 0 40.0F
C325 122 0 60.0F
C326 121 0 58.0F
C327 119 0 14.0F
C328 118 0 16.0F
C329 120 0 68.0F
C330 117 0 67.0F
C331 116 0 20.0F
C332 114 0 15.0F
C333 115 0 18.0F
C334 113 0 15.0F
C335 60 0 42.0F
C336 112 0 36.0F
C337 111 0 34.0F
C338 110 0 19.0F
C339 109 0 41.0F
C340 107 0 29.0F
C341 105 0 16.0F
C342 103 0 16.0F
C343 57 0 42.0F
C344 102 0 24.0F
C345 69 0 36.0F
C346 106 0 29.0F
C347 108 0 16.0F
C348 83 0 44.0F
C349 101 0 61.0F
C350 100 0 211.0F
C351 104 0 178.0F
C352 99 0 154.0F
C353 98 0 16.0F
C354 95 0 16.0F
C355 96 0 24.0F
C356 97 0 130.0F
C357 94 0 15.0F
C358 92 0 163.0F
C359 91 0 35.0F
C360 90 0 118.0F
C361 93 0 18.0F
C362 89 0 15.0F
C363 82 0 210.0F
C364 87 0 16.0F
C365 86 0 16.0F
C366 85 0 46.0F
C367 84 0 210.0F
C368 81 0 24.0F
C369 79 0 42.0F

C370 80 0 156.0F
C371 78 0 25.0F
C372 88 0 18.0F
C373 76 0 20.0F
C374 77 0 15.0F
C375 75 0 25.0F
C376 73 0 43.0F
C377 72 0 19.0F
C378 70 0 40.0F
C379 71 0 35.0F
C380 74 0 15.0F
C381 58 0 24.0F
C382 68 0 67.0F
C383 67 0 34.0F
C384 66 0 162.0F
C385 64 0 16.0F
C386 62 0 114.0F
C387 61 0 69.0F
C388 65 0 177.0F
C389 55 0 16.0F
C390 56 0 210.0F
C391 54 0 73.0F
C392 59 0 163.0F
C393 53 0 19.0F
C394 45 0 50.0F
C395 52 0 31.0F
C396 51 0 20.0F
C397 50 0 53.0F
C398 39 0 40.0F
C399 49 0 14.0F
C400 47 0 129.0F
C401 46 0 163.0F
C402 48 0 17.0F
C403 44 0 177.0F
C404 43 0 177.0F
C405 42 0 19.0F
C406 41 0 51.0F
C407 40 0 17.0F
C408 38 0 31.0F
C409 37 0 69.0F
C410 36 0 67.0F
C411 34 0 4.0F
C412 33 0 60.0F
C413 32 0 58.0F
C414 31 0 4.0F
C415 30 0 33.0F
C416 28 0 33.0F
C417 27 0 66.0F
C418 26 0 61.0F
C419 25 0 60.0F
C420 23 0 12.0F
C421 22 0 11.0F
C422 21 0 12.0F
C423 20 0 11.0F

C424 19 0 12.0F
 C425 18 0 11.0F
 C426 17 0 12.0F
 C427 16 0 11.0F
 C428 15 0 12.0F
 C429 14 0 11.0F
 C430 13 0 12.0F
 C431 12 0 11.0F
 C432 10 0 145.0F
 C433 8 0 145.0F
 C434 11 0 12.0F
 C435 9 0 11.0F
 C436 7 0 129.0F
 C437 6 0 129.0F

*

C1505 43 0 337.0F
 C1506 43 0 248.0F
 C1507 65 0 337.0F
 C1508 65 0 248.0F
 C1509 44 0 337.0F
 C1510 44 0 248.0F
 C1511 104 0 337.0F
 C1512 104 0 248.0F
 C1521 46 0 337.0F
 C1522 46 0 186.0F
 C1523 92 0 337.0F
 C1524 92 0 186.0F
 C1525 47 0 337.0F
 C1526 47 0 124.0F
 C1527 97 0 337.0F
 C1528 97 0 124.0F
 C1529 99 0 275.0F
 C1530 66 0 275.0F
 C1531 80 0 275.0F
 C1532 59 0 275.0F

*

*M9BH SPICE LEVEL 2 PARAMETERS

*

.MODEL CMOSN NMOS LEVEL=2 LD=0.181362U TOX=402.000000E-10
 + NSUB=6.567000E+15 VTO=0.805287 KP=4.757000E-05 GAMMA=0.5435
 + PHI=0.6 UO=553.83 UEXP=0.151038 UCRIT=48309.6
 + DELTA=0.823727 VMAX=50459.8 XJ=0.250000U LAMBDA=3.437039E-02
 + NFS=4.094390E+12 NEFF=1 NSS=1.000000E+12 TPG=1.000000
 + RSH=19.340000 CGDO=2.336825E-10 CGSO=2.336825E-10 CGBO=7.582249E-10
 + CJ=1.011600E-04 MJ=0.633000 CJSW=5.320000E-10 MJSW=0.266000
 PB=0.800000

* Weff = Wdrawn - Delta_W

* The suggested Delta_W is 0.40 um

*

.MODEL CMOSP PMOS LEVEL=2 LD=0.250000U TOX=402.000000E-10
 + NSUB=6.786000E+15 VTO=-0.758994 KP=1.843000E-05 GAMMA=0.5525
 + PHI=0.6 UO=214.5 UEXP=0.253978 UCRIT=40136.1

```
+ DELTA=0.135535 VMAX=78961.6 XJ=0.050000U LAMBDA=4.876526E-02
+ NFS=4.352678E+11 NEFF=1.001 NSS=1.000000E+12 TPG=-1.000000
+ RSH=107.700000 CGDO=3.221216E-10 CGSO=3.221216E-10 CGBO=6.309201E-
10
+ CJ=2.474000E-04 MJ=0.548900 CJSW=3.155000E-10 MJSW=0.327000
PB=0.800000
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is -0.12 um

*
.probe
.print tran v(29) v(46) v(47) v(68) v(90)
.tran 1ns 80ns
.end
```

VITA

Manish K. Shah

Candidate for the Degree of

Master of Science

Thesis: VLSI DESIGN OF A TWIN REGISTER FILE FOR REDUCING THE EFFECTS OF CONDITIONAL BRANCHES IN A PIPELINED ARCHITECTURE

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Baroda, India, February 25, 1967, the son of Khantilal Shah and Indumati Shah.

Education: Graduated from Rosary High School, Baroda, India, in June 1984; received Bachelor of Engineering Degree in Electronics Engineering from Maharaja Sayajirao University at Baroda, India in December, 1988; completed requirements for the Master of Science degree at Oklahoma State University in December, 1992.

Professional Experience: Teaching Assistant, Department of Electrical and Computer Engineering, Oklahoma State University, August, 1991, to March 1992. Member of PHI KAPPA PHI, TAU BETA PI, and IEEE.