VLSI DESIGN OF THE TINY RISC MICROPROCESSOR

A. Abnous, C. Christensen, J. Gray,
J. Lenell, A. Naylor, N. Bagherzadeh

Department of Electrical and Computer Engineering
Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No. 91-74

VLSI Design of the Tiny RISC Microprocessor

Arthur Abnous, Christopher Christensen, Jeffrey Gray,
John Lenell, Andrew Naylor, and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717

Abstract

This report describes the Tiny RISC microprocessor designed at UC Irvine. Tiny RISC is a 16-bit microprocessor and has a RISC-style architecture. The chip was fabricated by MOSIS [1] in a $2\mu m$ n-well CMOS technology. The processor has a cycle time of 70 ns.

1 Introduction

This report presents the VLSI design and implementation of the Tiny RISC microprocessor. Tiny RISC is a 16-bit microprocessor with a RISC-style instruction set. The chip was fabricated by MOSIS in a $2\mu m$ n-well, 2-level metal CMOS technology. The processor is pipelined, and can execute an instruction in each cycle. The instruction set is designed for efficient pipelining and decoding. Tiny RISC has a cycle time of 70 ns (14.3 MHz clock speed). This gives it a peak performance of 14 16-bit MIPS. The processor was laid out using the MAGIC VLSI layout system [2]. IRSIM [3] was used extensively to simulate and verify the operation of the processor. PLA's were created using MPLA [4].

2 Instruction Set

Table 1 outlines the instruction set of Tiny RISC. Instruction formats are shown in Figure 1. All instructions, memory addresses, and operands are 16

bits wide. Most instructions are of the arithmetic/logic/shift type. *src1* and *src2* are source registers and *dest* is the destination register. There are eight general-purpose registers, R0 thru R7. R0 is hardwired to contain zero at all times. Writing to it is allowed but will not affect its content. Shift instructions have two versions, constant and variable. For constant shift instructions, the shift amount is specified by the lower four bits of a 5-bit immediate value (I). For variable shift instructions, the shift amount is specified by the lower four bits of the contents of *src2*. ADDI and SUBI are the immediate versions of ADD and SUB with *src2* replaced by a 5-bit immediate value (I), which is always zero-extended. The LDI instruction loads an 8-bit immediate value (LI) into *dest*. LI is also zero-extended. Set instructions (SEQ, SLT, SLTU, SGE, SGEU) set or clear the least significant bit of *dest* by comparing *src1* and *src2* and checking for the specified condition. The upper 15 bits are always cleared to zero. Set instructions are usually used to compute condition codes for a subsequent branch instruction.

Load/Store instructions use the register indirect addressing mode. There are two instructions in this category: LDW and STW. For both instructions, *src1* contains a data memory address. The LDW instruction loads $M[src1]$ into *dest* (in this context, $M[x]$ refers to the content of the memory location addressed by register x). The STW instruction stores the content of *src2* to $M[src1]$. Tiny RISC has a Harvard architecture. There are two separate memory spaces: one for instructions and one for data. Each memory is accessed via a separate 16-bit multiplexed bus. The DATA bus provides access to the data memory, and the INST bus provides access to the instruction memory. The only difference between the operations of the two buses is that there are no write operations to the instruction memory.

Branch instructions (BRF and BRT) check the least significant bit of *src1* and transfer control to the target of the branch if the specified condition is met. The target address of branch instructions is computed by adding an 8-bit immediate offset (O) to the address of the instruction following the branch. O is always sign-extended. The CALL instruction is used for procedure call operations. The Program Counter is loaded with *src1*, and the return address is saved into *dest*. The JUMP instruction loads the PC with the content of *src1*. It is used to return from procedure calls.

Opcode	*Parameters*	*Description*
ADD	*src1,src2,dest*	add
SUB	*src1,src2,dest*	subtract
AND	*src1,src2,dest*	bitwise AND
OR	*src1,src2,dest*	bitwise OR
XOR	*src1,src2,dest*	bitwise exclusive-OR
XNOR	*src1,src2,dest*	bitwise exclusive-NOR
LSL	*src1,src2,dest*	logical shift left (variable)
LSR	*src1,src2,dest*	logical shift right (variable)
ASR	*src1,src2,dest*	arithmetic shift right (variable)
SEQ	*src1,src2,dest*	set if equal
SLT	*src1,src2,dest*	set if less than
SLTU	*src1,src2,dest*	set if less than unsigned
SGE	*src1,src2,dest*	set if greater than or equal
SGEU	*src1,src2,dest*	set if greater than or equal unsigned
ADDI	*src1,#I,dest*	add immediate
SUBI	*src1,#I,dest*	subtract immediate
LSLI	*src1,#I,dest*	logical shift left (constant)
LSRI	*src1,#I,dest*	logical shift right (constant)
ASRI	*src1,#I,dest*	arithmetic shift right (constant)
LDI	*#LI,dest*	load immediate
LDW	*src1,dest*	load word
STW	*src1,src2*	store word
BRF	*src1,#O*	branch if false
BRT	*src1,#O*	branch if true
JUMP	*src1*	jump
CALL	*src1,dest*	call

Table 1: Instruction Set

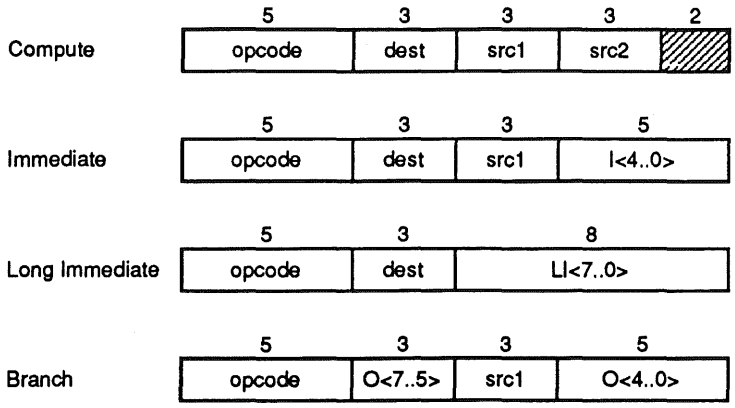


Figure 1: Instruction Formats

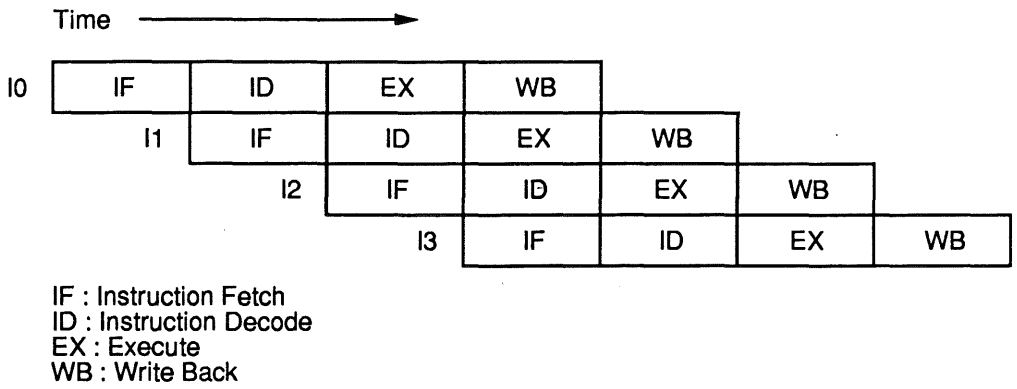


Figure 2: Pipeline Structure of Tiny RISC

3 Pipeline Structure

Figure 2 shows the pipeline structure of Tiny RISC. There are four pipeline stages: IF (Instruction Fetch), ID (Instruction Decode), EX (EXecute), and WB (Write Back). The instruction set was designed with this pipeline structure in mind. Each pipeline stage takes one cycle to complete its function. The timing of each cycle is derived from a non-overlapping four-phase clock. As shown in Figure 3, there are five different clock signals along with their complements: Ph1, Ph2, Ph3, Ph4, and Ph123.

During the IF stage, an instruction is fetched from the program memory.

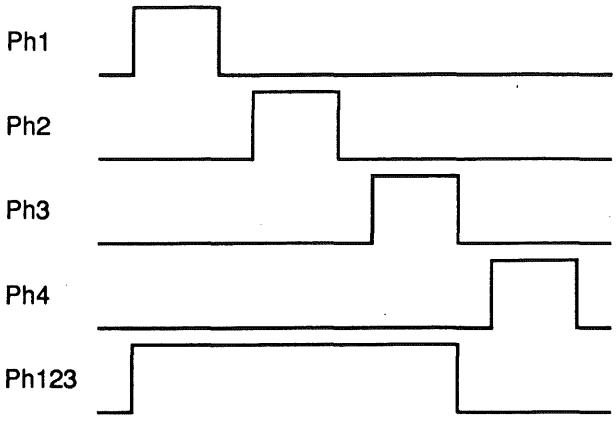


Figure 3: Four-Phase Clocking Scheme

The instruction is decoded, and the source operands are read from the register file during the ID stage. Most control signals are generated in this stage. The control signals that become active during later pipeline stages are delayed by the proper amount. Control transfer instructions update the PC at the end of ID stage. This means that all control transfer instructions have a delay of one cycle. The instruction following a control transfer instruction is always fetched and executed. It is the programmer's responsibility to schedule a useful instruction in the delay slot. If such an instruction is not found a NOP should be scheduled in the delay slot. During the EX stage, the actual operation specified by the instruction is performed. LDW and STW instructions access the data memory during this stage. In the WB stage, the result of the instruction is written to the register file.

To avoid data hazards caused by the delay between RD and WB stages, Tiny RISC uses one level of *bypassing* [5]. The bypassing hardware compares the source register addresses of the instruction about to enter the EX stage to the destination register address of the previous instruction. If there is a match, the operand read from the register file is discarded, and instead, the result of the previous EX stage is used.

4 Data Path Design

Figure 4 shows the structure of the data path of the processor. The dimensions of various hardware blocks are shown in microns. The data path of the pro-

cessor consists of the Register File, the Bypassing Unit, the Load/Store Unit, the Shifter, the ALU, and the PC Unit. The control circuitry, the instruction register, and the clock generator are all located on the right hand side of the data path.

Each bit-slice of the data path is 56 μm wide and can accommodate seven metal2 (second-level metal) tracks running in the vertical direction. One of these tracks is used to distribute power or ground. This track is shared between two adjacent bit-slices. This is done by flipping every other bit-slice (see Figure 5). One track is allocated to the S1 (*src1*) bus, one to the S2 (*src2*) bus, and one to the D (*dest*) bus. The other three tracks are used for local routing within the data path.

Static flow-thru latches were used throughout the data path. Figure 6 shows the schematic of a latch cell. The layout of a latch cell is shown in Figure 7.

5 Operand Unit

The Operand Unit (OU) consists of the following hardware blocks:

1. Register File
2. Bypassing Unit
3. Load/Store Unit

The block diagram of the OU is shown in Figure 8.

5.1 Register File

The register file is dual-ported and contains eight registers, R0 thru R7. Register R0 is hardwired to contain zero at all times. Figure 9 shows the floorplan of the register file. The dimensions of various hardware blocks are shown in microns. The floorplan of the register file is centered around an 8 by 16 array of register cells (eight 16-bit registers). The first row from the bottom corresponds to register R0. Two sets of decoders (one for each port) are positioned on the right hand side of the register cell array. The read/write circuitry is located at the bottom of the register cell array and facilitates access to the register file.

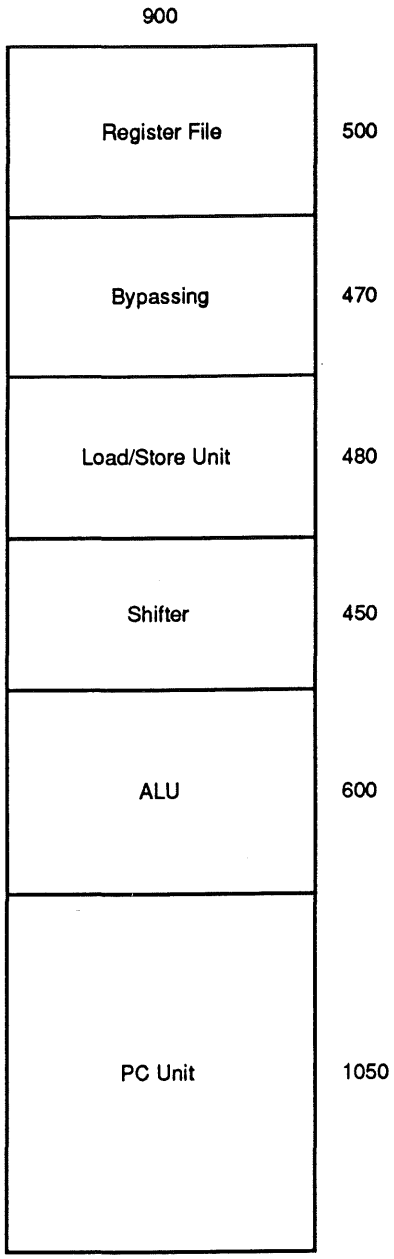


Figure 4: Data Path Structure

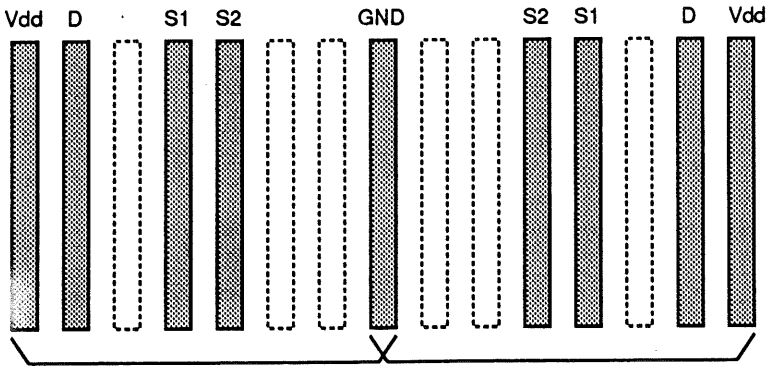


Figure 5: Data Path Track Assignment

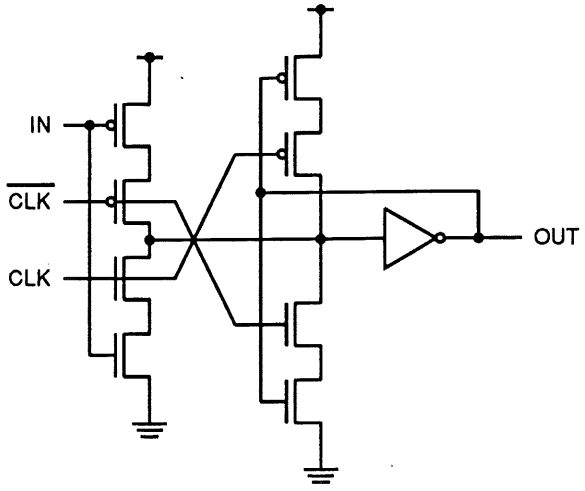


Figure 6: Data Path Latch Cell

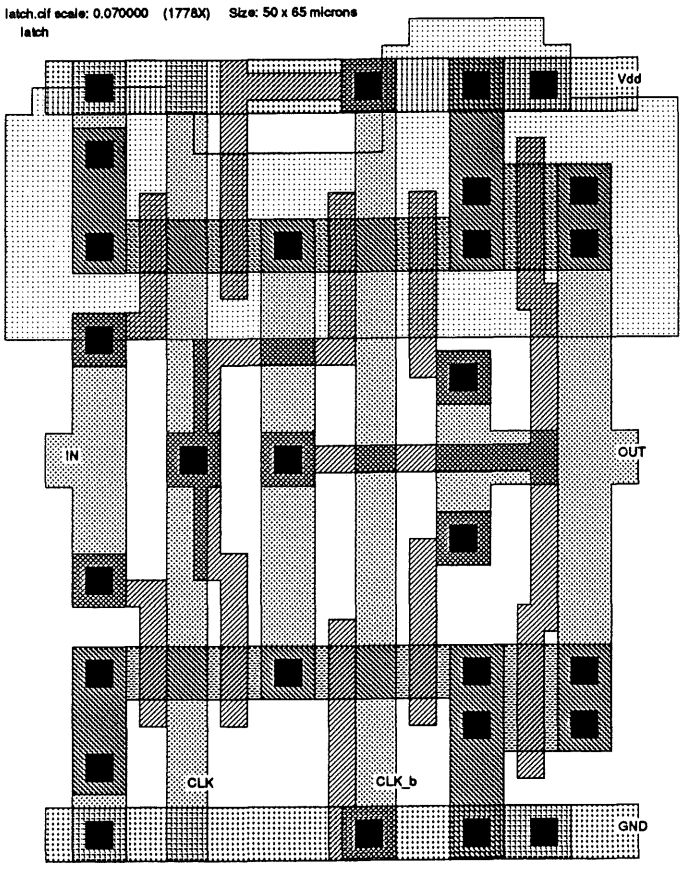


Figure 7: Layout of the Latch Cell

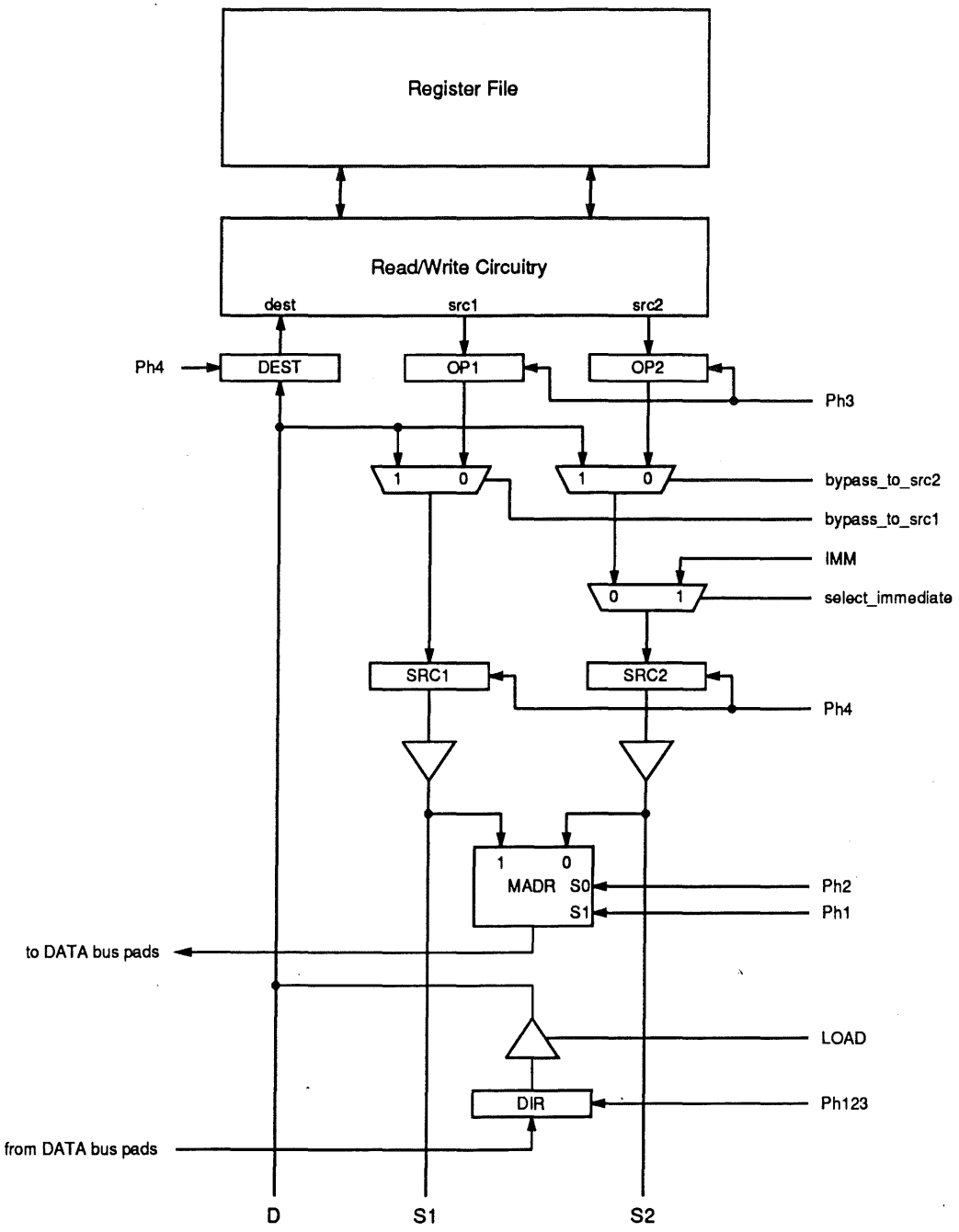


Figure 8: Block Diagram of the Operand Unit

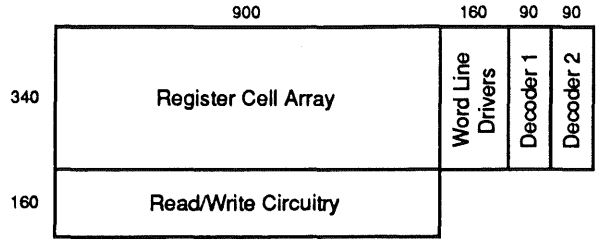


Figure 9: Floorplan of the Register File

The register cell design was based on the CMOS version of the 6-transistor dual-port static RAM cell with split word lines [6], which is shown in Figure 10. The layout of the register cell is shown in Figure 11. A group of four register cells is shown in Figure 12. The register cell consists of a pair of cross-coupled inverters (providing the bistability needed for static storage) and two access transistors. Each access transistor is controlled by one of the word lines (WORD1 or WORD2), which are driven by the decoders.

To perform a read operation, the bit lines (BIT1 and BIT2) are precharged high first. If the register is selected on port 1, then WORD1 goes high, providing an access path from the storage node DATA to BIT1. If data is high, then BIT1 will stay precharged. However, if data is low, then the pull-down transistor of the inverter driving data will start to discharge the bit line. The capacitance of the bit line and the strength of the pull-down network determine the speed with which the bit line is pulled low.

If the cell is not properly designed, a read operation can destroy the information stored in the cell. The reason is that when the access transistor turns on, the high level voltage of the precharged bit line will be divided between the series combination of the access transistor and the pull-down down transistor of the inverter. If the voltage drop across the pull-down transistor is sufficiently high to turn on the pull-down transistor of the the other inverter of the cell, then the cell is in danger of losing its data. This means that the cell has to be designed so that the disturbance caused by a read operation is not strong enough to destroy the cell information. This requires that the ratio of the β of the pull-down transistor to that of the access transistor be large enough to reduce the read disturbance voltage to the desired level. A safe design will reduce the read disturbance voltage to less than the threshold voltage of an n-transistor. This will ensure that the n-transistor of the other inverter of the cell will not be turned on by the read disturbance. In this design, the

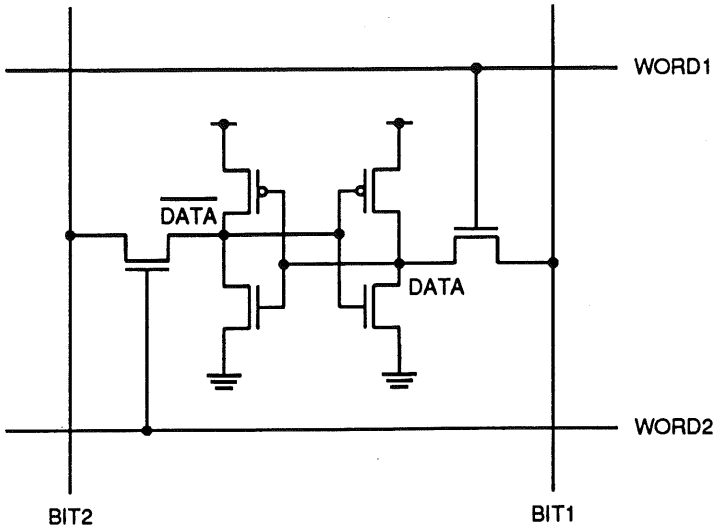


Figure 10: Dual-Port Register File Cell

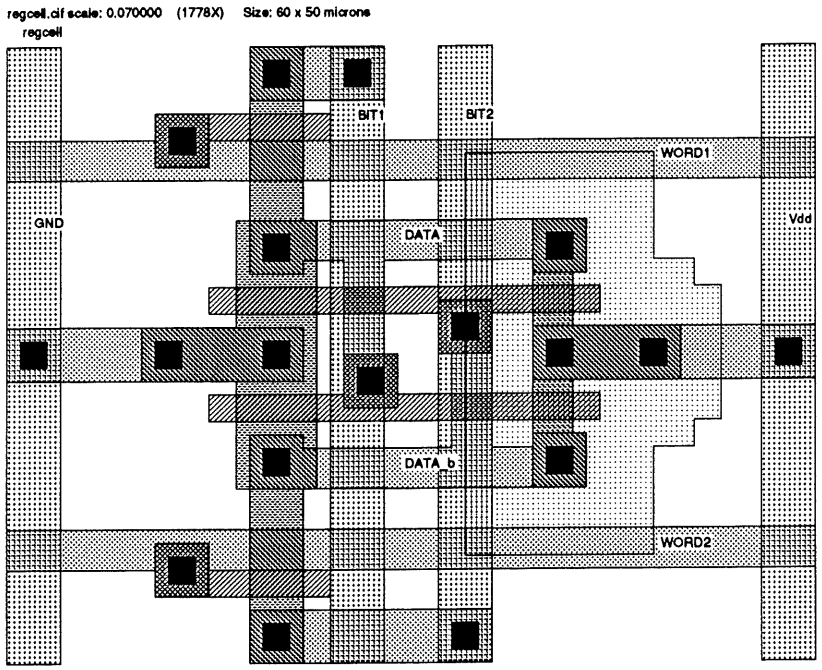


Figure 11: Layout of the Dual-Port Register File Cell

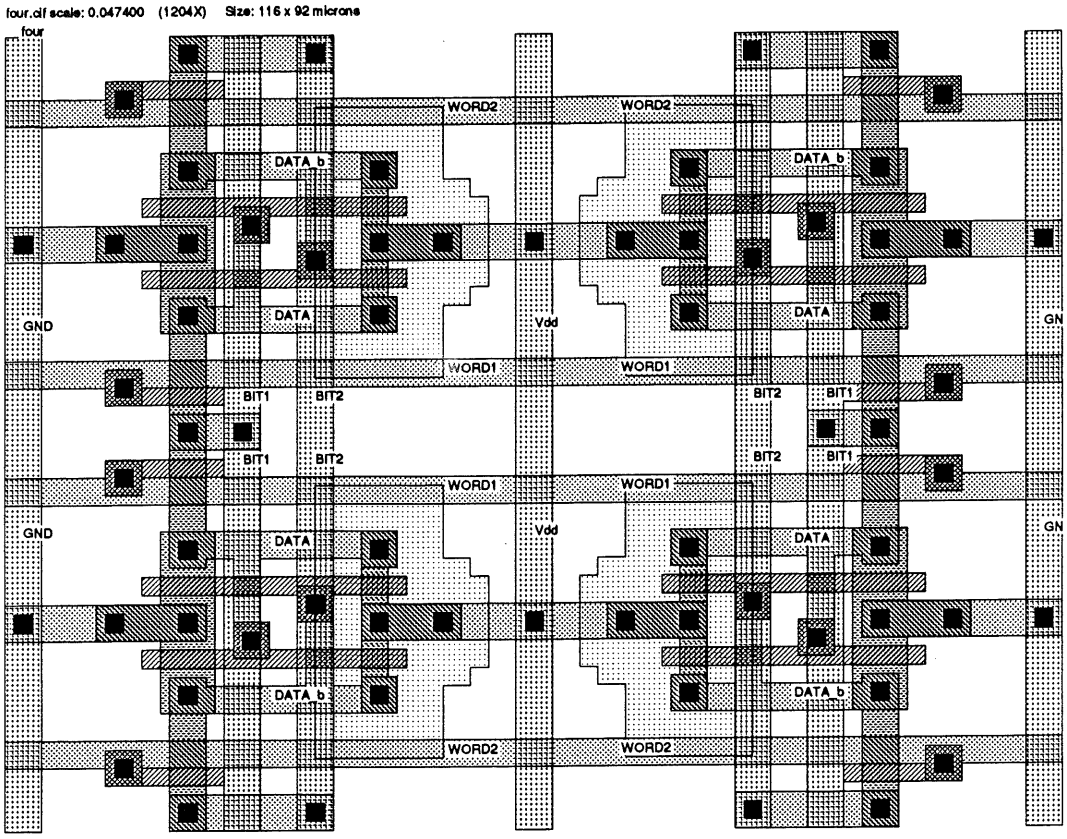


Figure 12: A Group of Four Register Cells

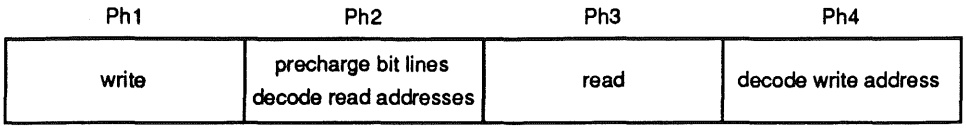


Figure 13: Timing of the Register File

width of the inverter pull-down transistor is 6 microns, and the width of each access transistor is 4 microns. The length of all transistors is minimum size, which is 2 microns.

The write operation is performed by setting the desired data on BIT1 line and the complement of the data on BIT2 and turning on both word lines. The cell is actually written to by the bit line carrying the low level. The bit line with the high level cannot disturb the cell because the cell is designed not to be disturbed by a high level on the bit lines (as mentioned earlier, this is done to ensure that the cell information is not destroyed by a read operation, which starts with precharged bit lines). The cell was designed such that the bit line with the low level can flip the state of the cell. This requires that the ratio of the β of the access transistor to that of the inverter pull-up transistor be large enough to initiate a write operation. In this design, the width of the inverter pull-up transistor is 3 microns, and the length of the transistor is 2 micron.

The timing of the register file, as well as the rest of the processor, is based on a four-phase clock. The timing of the register file is shown in Figure 13. During Ph2, the bit lines are precharged, and the source register addresses are decoded. In Ph3, the word lines are driven by the decoders, and the read operation takes place. In Ph4, the source operands are driven onto the S1 and S2 busses. Also, the register address for the write operation to follow in the next Ph1 are decoded. During Ph1, the desired data is driven onto the bit lines, the output of the decoders are driven onto the word lines, and the write operation takes place.

5.2 Load/Store Unit

For load/store operations, the data memory is accessed via a 16-bit multiplexed bus (the DATA bus). The external bus protocol for a read operation is shown in Figure 14. The protocol for a write operation is shown in Figure 15. When Ph1 rises, the memory address (*src1*) is loaded into MADR (Memory

Address/Data Register) and driven onto the DATA bus. This address must be latched externally. Ph1 serves as the strobe signal for the external address latch. The address should be latched when Ph1 falls. The DATA bus holds the address until Ph2 rises. The dead time between Ph1 and Ph2 provides the proper hold time for the external address latch. For load operations, when Ph2 rises, the DATA bus is tri-stated to accommodate the incoming data from memory. The data is loaded into DIR (Data In Register) with the falling edge of Ph3 and is written to the register file in the WB stage. For store instructions, when Ph2 rises, the data to be stored to memory (*src2*) is driven onto the DATA bus and is then written to the data memory. The $\overline{\text{DRD}}$ (Data Read) and $\overline{\text{DWR}}$ (Data Write) signals are used to interface the data memory. The instruction memory is accessed via the INST bus, which is identical to the DATA bus except that there are no write operations to the instruction memory. The $\overline{\text{IRD}}$ (Instruction Read) signal is provided to interface to the instruction memory.

MADR is a special latch that has two inputs (I0 and I1) and two clock (or select) signals (S0 and S1). The select signals are assumed to be non-overlapping, i.e., they can never be both high at the same time. When S0 goes high, I0 is driven to the output. When S1 goes high, I1 is driven to the output. When S0 and S1 are both low, the output is latched through bistable action. The schematic of a multiplexing latch is shown in Figure 16. This circuit achieves both the logical operation and the timing required for load/store instructions.

6 Arithmetic/Logic Unit

The Arithmetic/Logic Unit (ALU) computes all arithmetic, logical, and comparison operations required by the instruction set. Table 2 lists the instructions for which the ALU is utilized to compute results.

For arithmetic and logical instructions, the specified operation is performed on the source operands (*src1* and *src2*) and the result is written back into the destination register (*dest*). For comparison operations, *src1* and *src2* are compared, and depending on the outcome of the comparison, a one or a zero is written back into *dest*. For example, suppose that R1 contains `0005H` and R2 contains a `0007H`. If the instruction `SLT R1,R2,R3` were executed, then `0001H` would be written to R3 to signify that the content of R1 is less than

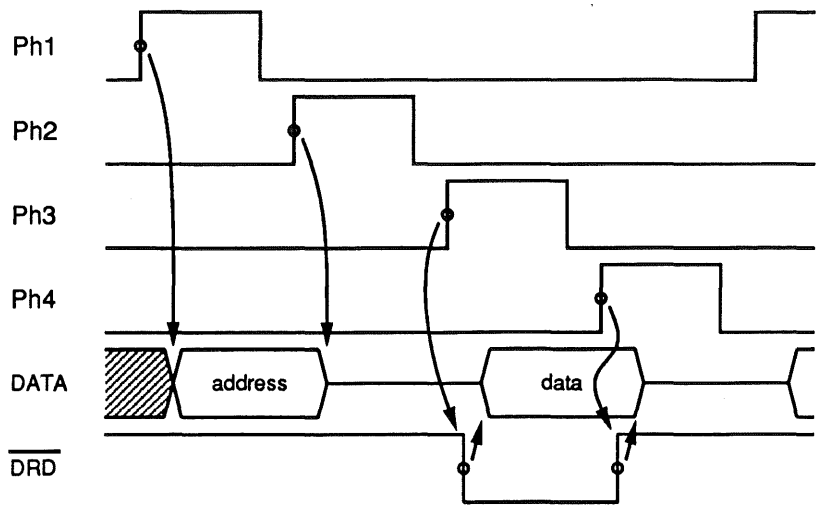


Figure 14: DATA Bus Protocol for Read Operations

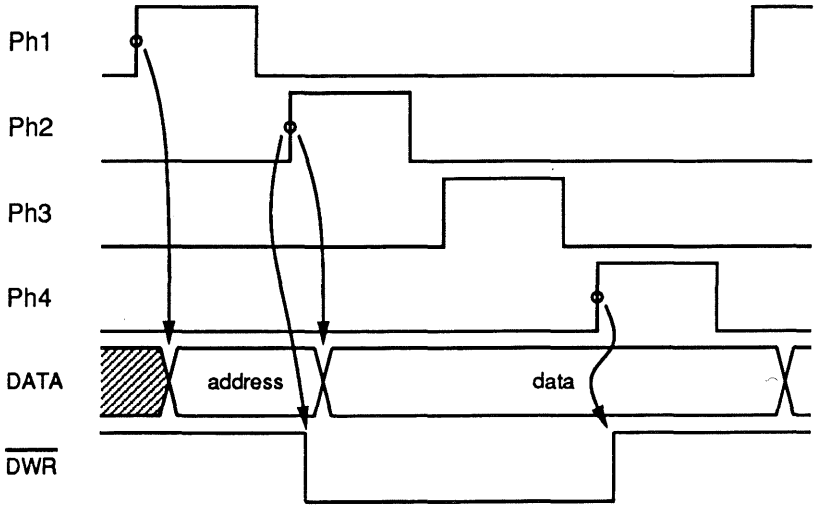


Figure 15: DATA Bus Protocol for Write Operations

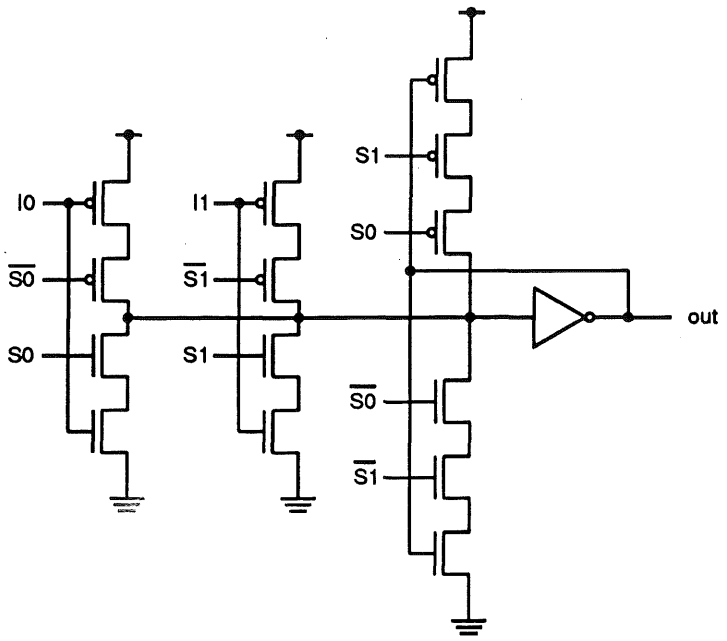


Figure 16: A Multiplexing Latch

Type	*Opcode*	*Operation*
Arithmetic	ADD	add
	ADDI	add immediate
	SUB	subtract
	SUBI	subtract immediate
Logical	AND	bitwise AND
	OR	bitwise OR
	XOR	bitwise exclusive-OR
	XNOR	bitwise exclusive-NOR
Comparison	SEQ	set if equal
	SLT	set if less than
	SLTU	set if less than unsigned
	SGE	set if greater than or equal
	SGEU	set if greater than or equal unsigned

Table 2: ALU Operations

the content of R2. If the instruction `SGE R1,R2,R3` were executed, then `0000H` would be written to R3 to signify that the content of R1 is not greater than or equal to the content of R2.

6.1 ALU Organization

The organization of the ALU is shown in Figure 17. This figure also reflects the actual floorplan of the ALU. Located at the top of the ALU is the PG block which generates the P (Propagate) and G (Generate) signals used by the carry chain. The carry chain itself is located below the PG block. The comparison logic can be found below the carry chain. Immediately below the comparison circuitry are the multiplexors which select either the output of the carry chain or the output of the comparison logic.

6.2 PG Block

The PG block is a Weinberger structure [7] that is used to generate the P (Propagate) and G (Generate) signals that are needed by the carry chain. The schematic of the PG block is shown in Figure 18. The PG block and the carry chain use domino logic. The P output can be made to be any logic function of S1 and S2, and the G output can be made to be one of four different logic functions of S1 and S2. The FP⟨3–0⟩ and FG⟨1–0⟩ control signals are used to select a specific function for P and G, respectively.

The P and G signals are generated in a similar manner. The P generator operates as follows. During Ph4, the p-transistor connected to the $\overline{P}$ node turns on, precharging $\overline{P}$. During Ph123, the evaluate phase, some of the FP signals are pulled low, creating possible discharge paths. If FP0 is pulled low, $\overline{P}$ will be discharged if $\overline{S1}$ and $\overline{S2}$ are high. If FP1 is pulled low, $\overline{P}$ will be discharged if S1 and $\overline{S2}$ are high. If FP2 is pulled low, $\overline{P}$ will be discharged if $\overline{S1}$ and S2 are high. If FP3 is pulled low, $\overline{P}$ will be discharged if S1 and S2 are high. Tables 3 and 4 show how a specified function can be generated by the PG block by controlling the FP and FG inputs.

To perform an addition, P should be S1$\oplus$S2, G should be S1$\cdot$S2, and C0 (carry in) should be zero. To subtract S2 from S1 (S1$-$S2), P should be S1$\odot$S2, G should be S1$\cdot\overline{S2}$, and C0 should be one. The logic functions are accomplished by putting the desired function on P, and making G and C0 both zero. This causes P to be passed right through to the output of the carry

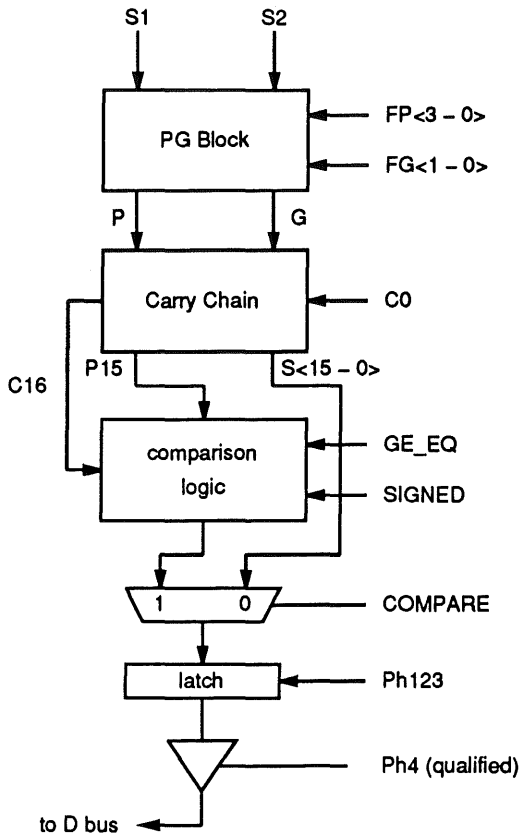


Figure 17: Block Diagram of the ALU

Function	*FP0*	*FP1*	*FP2*	*FP3*
S1·S2	1	1	1	0
S1+S2	1	0	0	0
S1⊕S2	1	0	0	1
S1⊙S2	0	1	1	0
S1	1	0	1	0
S2	0	1	0	1
0	1	1	1	1
1	0	0	0	0

Table 3: P Generator Functions

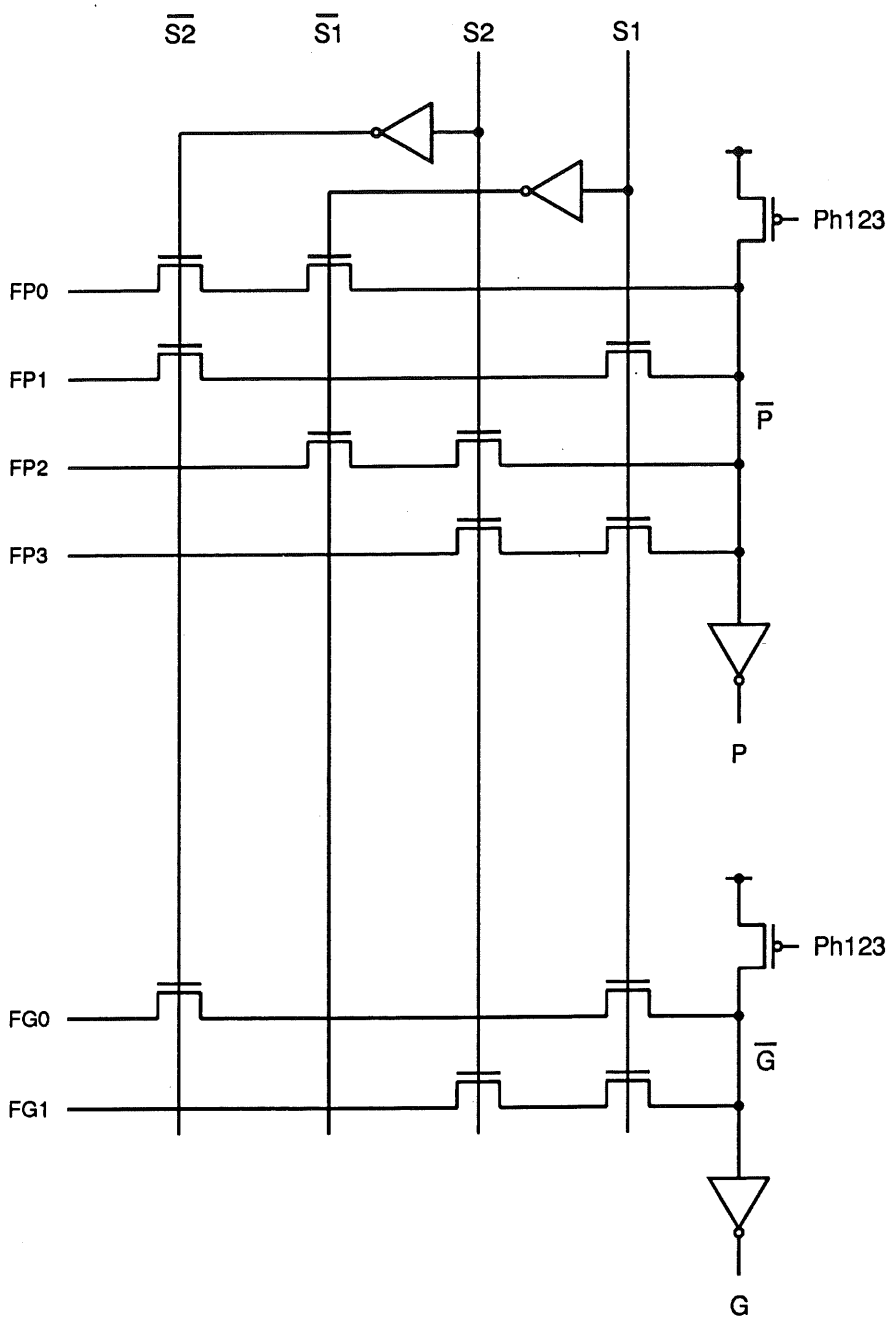


Figure 18: Circuit Diagram of the PG Block

Function	*FG0*	*FG1*
S1·S2	1	0
S1·$\overline{S2}$	0	1
S1	0	0
0	1	1

Table 4: G Generator Functions

chain. All of the comparisons (except for SEQ) are done by using the carry chain to subtract S2 from S1 and using the compare logic to check the result. The SEQ comparison is done by setting P to S1⊕S2, setting G to 0, setting C0 to 1, and checking C16 (carry out) with the comparison logic. C16 will only be one if all of the P's are one, meaning that S1 and S2 are bit-for-bit equal. Because this is a domino circuit, S1 and S2 must be stable when the precharge transistor turns off. A fluctuation of S1 or S2 could erroneously discharge the $\overline{P}$ node. For this reason, the S1 and S2 busses are driven during Ph4, the precharge phase.

6.3 Carry Chain

The ALU utilizes a dynamic Manchester carry chain [8]. The inputs to the carry chain are P⟨15–0⟩, G⟨15–0⟩, and C0 (carry input for bit 0). The chain consists of a cascade of Manchester carry elements (see Figure 19), which compute the carry output for each bit position (C1 thru C16). Notice that in the Manchester carry scheme the complement of the carry is actually propagated. During the precharge phase (Ph4), the carry output of each bit is precharged. During the subsequent evaluate phase (Ph123), the carry out nodes are conditionally discharged.

The worst case delay through the carry chain occurs when all P's are high, and C0 is high. In this situation, the carry signal has to propagate through 16 series pass transistors. To avoid the excessive delay through a long chain of pass transistors, the carry chain is broken into groups, and the carry output of each group is buffered before being fed into the next carry group (see Figure 20).

This strategy reduces the number of series pass transistors to six. The performance of the carry chain can be further improved by reducing the capacitance of the carry nodes in the Manchester chain. This will have the effect of decreasing the carry node discharge time. The input capacitance of the

XOR gates of the sum stage increases the load of the internal carry nodes of the carry chain and slows the carry evaluation time. This delay penalty can be eliminated by having a second Manchester stage. This second stage outputs the carry nodes to the sum stage and is divided in the same manner as the first stage. The carry input into each group is the carry output of the previous group in the first Manchester stage. The complete carry chain is shown in Figure 21. Thus, the first Manchester stage evaluates the carry out of each Manchester group very quickly, and distributes these carry outputs to the carry inputs of the second stage Manchester groups. In this way, each Manchester group in the second Manchester stage is evaluating the carry bits to be passed to the sum stage nearly in parallel with the other Manchester units of the second stage. Thus, the Manchester adder can be optimized to achieve a desired operating speed within technology constraints.

The second stage of the Manchester carry chain outputs the complements of the carry bits for all 16 bits. These carry bits are XORed with the P signals in the sum stage to generate the 16 sum bits (S⟨15–0⟩). The sum stage uses static XOR gates.

6.4 Comparison Logic

The comparison logic checks C16 and P15 from the carry chain, and two control signals (GE_EQ and SIGNED) to compute the result of a comparison instruction. Table 5 lists the logic expression that corresponds to each comparison condition (SEQ is treated as SGEU; see Section 6.2).

In Table 5, V refers to overflow, which is C16⊕C15. Since S15 can be expanded to P15⊕C15, the term S15⊕V expands to (P15⊕C15)⊕(C16⊕C15), which in turn simplifies into P15⊕C16. These simplifications are shown in Table 6.

Since C16 becomes available before S15, the comparison logic begins evaluating before the sum stage is completely done. The comparison logic is quite simple (see Figure 22). The comparison logic is unusual in that it is not the same in each bit-slice. It was put into the data path mostly because the two critical inputs P15 and C16 are on the far side of the data path and would otherwise need to be routed about 900 microns to where the control logic is located (next to bit 0). The layout of the comparison circuit is very similar to the arrangement of the circuit diagram.

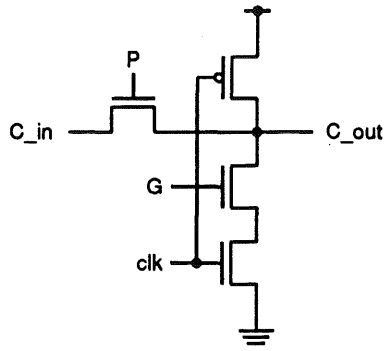


Figure 19: Circuit Diagram of the Manchester Carry Element

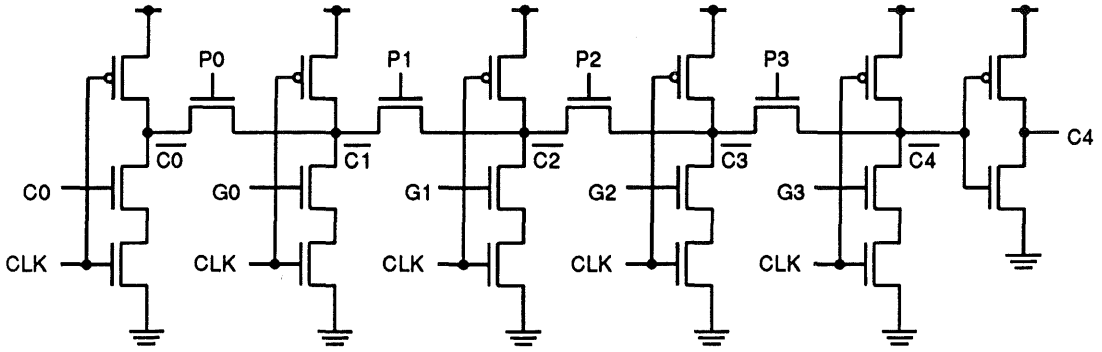


Figure 20: Circuit Diagram of a 4-Bit Manchester Carry Group

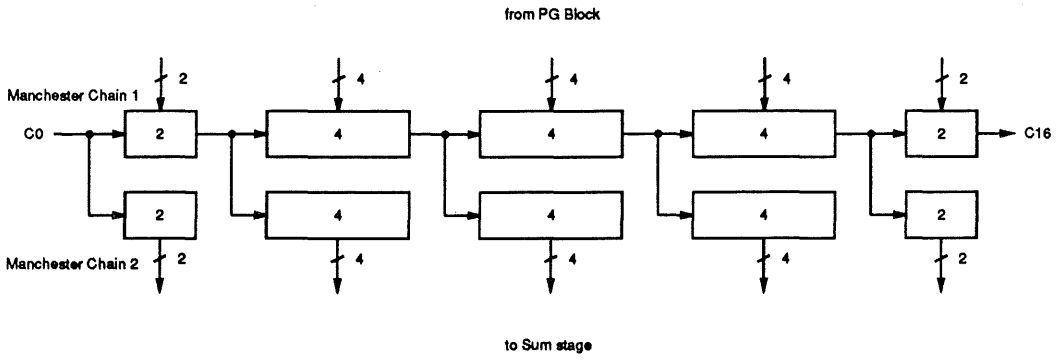


Figure 21: Block Diagram of the 16-bit Manchester Carry Chain

Comparison	*Expression*
SLT	$S15 \oplus V$
SGE	$\overline{S15 \oplus V}$
SLTU	$\overline{C16}$
SGEU	C16

Table 5: Logic Expressions for Comparison Operations

Comparison	*Expression*
SLT	$P15 \oplus C16$
SGE	$\overline{P15 \oplus C16}$
SLTU	$\overline{C16}$
SGEU	C16

Table 6: Simplified Logic Expressions for Comparison Operations

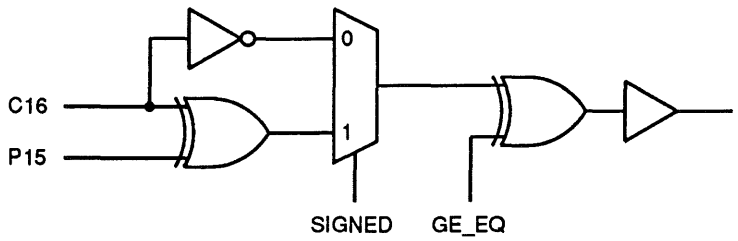


Figure 22: Circuit Diagram of the Comparison Circuit

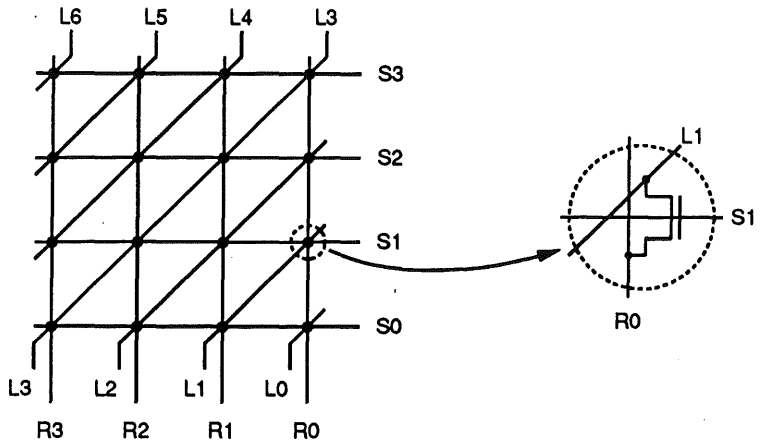


Figure 23: Basic Configuration for a 4-Bit Shifter

7 Shifter Unit

The Shifter Unit is responsible for performing all of the shift operations specified in the Tiny RISC instruction set. These include logical left shifts, logical right shifts, and arithmetic right shifts. The function of the Shifter Unit is to shift the operand on the S1 bus by the amount specified by the operand on the S2 bus (only the least significant four bits of *src2* are used). Since the Operand Unit drives immediate values onto the S2 bus in the same way as it does register operands, the Shifter Unit and its control logic handle constant shifts in the same way that they handle variable shifts.

The core of the shifter consists of a 16×16 matrix of pass transistors, forming a crossbar switch [6]. The basic configuration of a 4×4 crossbar shifter is shown in Figure 23. For the sake of simplicity, the operation of the shifter will be explained using this 4×4 configuration.

The literal lines (L0–L6) are inputs, and the result lines (R0–R3) are outputs. The decoded shift amount is driven onto the select lines (S0–S3). Each node in the shifter core represents a single n-transistor, with its gate, source, and drain connected to a select, literal, and result line, respectively (see Figure 23). Using this core, we can accomplish any of the required shift operations through appropriate selection of how data is applied to the literal and select lines.

For right shifts, the data is applied onto L3–L0 (L3 gets the MSB), and

LS	*AR*	*Operation*
0	0	logical right shift
0	1	arithmetic right shift
1	0	logical left shift
1	1	unused

Table 7: Definition of the Control Signals of the Shifter Unit

the shift-in value is applied to L6–L4. This shift-in value is equal to the MSB of the input data (S1_3 for the 4×4 configuration) for arithmetic shifts, and is always zero for logical shifts. The shift amount is decoded, and one of the select lines is driven high (S0 for a zero-bit shift, S1 for a one-bit shift, S2 for a two-bit shift, etc.).

For left shifts, the data is applied to lines L6–L3 (L6 gets the MSB), and the shift-in value (always zero) is applied to lines L2–L0. The one difficulty with this method is that it requires that the order of the select lines be reversed, i.e., S3 now corresponds to a zero-bit shift, S2 to a one-bit shift, etc. This is easily accomplished by decoding the one's complement of the shift amount and driving the result on the select lines. This method works when the number of select lines is a power of two. The complete block diagram of a 4×4 shifter is shown in Figure 24. The Shifter Unit is controlled by two control signals: LS (Left Shift) and AR (ARithmetic shift). These signals are explicitly provided in the instruction opcode. The function of these control signals is defined in Table 7.

The shifter makes extensive use of dynamic logic, requiring precharging for the literal lines, the result lines, and the decoder. The circuit diagram of a precharge/discharge path is shown in Figure 25. A dynamic design for the decoder was chosen because it could be fit into the same vertical space that was taken by the shifter core. The logic driving the literal line is a dynamic circuit that takes the place of the multiplexor/AND gate combination that was shown in the block diagram. This portion of the circuit selectively discharges the literal line based on the value on the *src1*, the MSB of *src1*, and the control lines. The discharged literal line then discharges the result line if the corresponding select line is selected.

The Shifter Unit has the same timing as the rest of the data path. The shifter evaluates during Ph123, and the result is driven onto the D bus during Ph4 if the instruction in the EX stage is a shift instruction. The shifter is precharged during Ph4. Because of the dynamic operation of the shifter, all

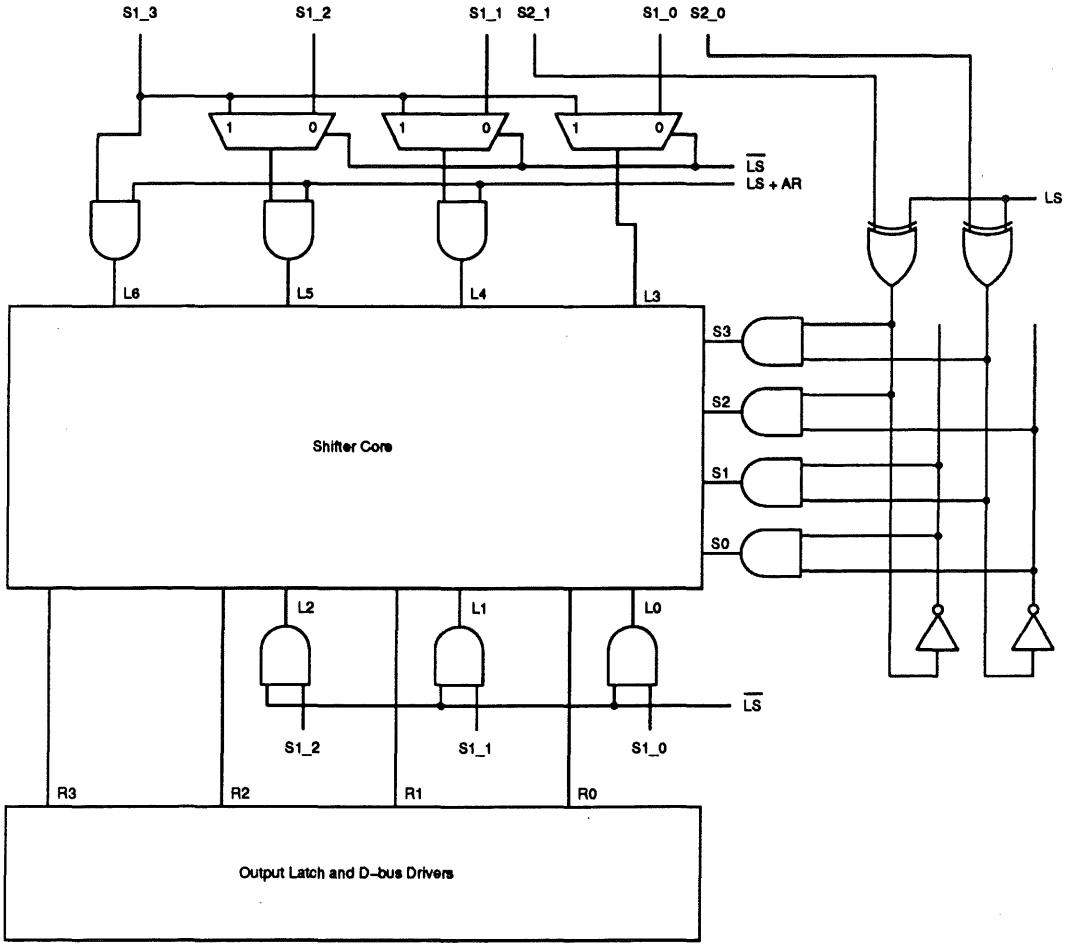


Figure 24: Block Diagram of a 4-Bit Barrel Shifter

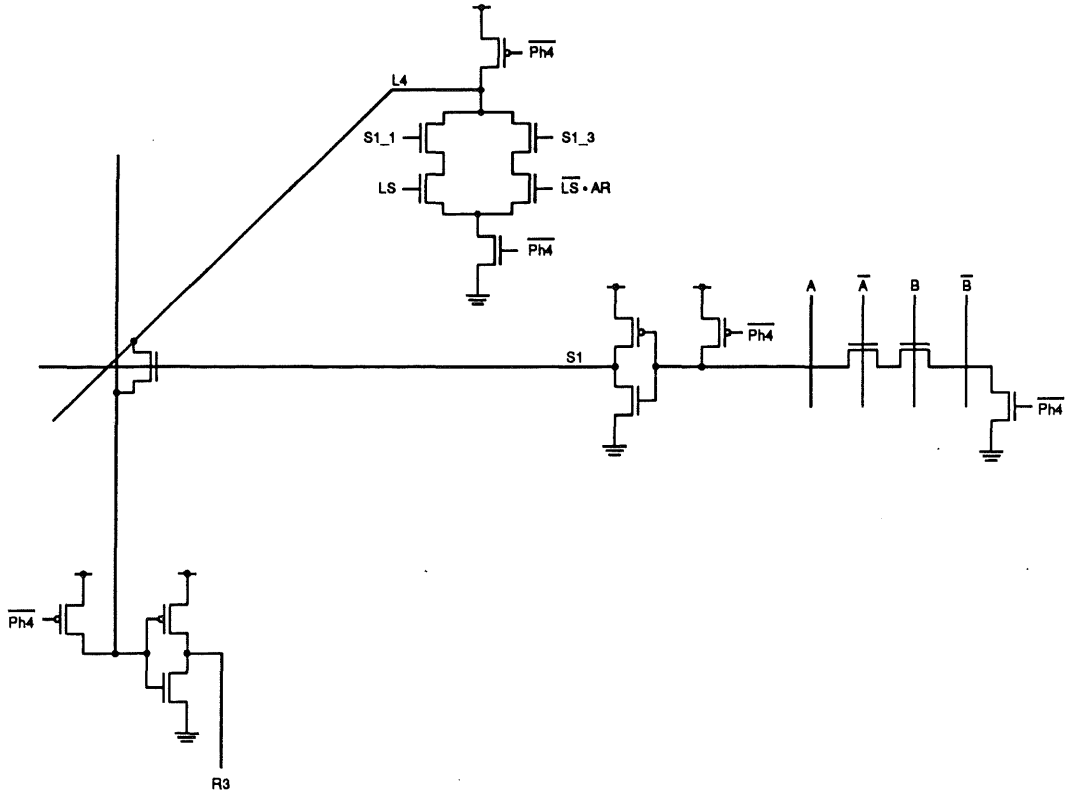


Figure 25: Circuit Diagram of the 4-Bit Shifter

input data and control signals to the shifter must reach steady state before Ph123 starts. The shifter requires 3 ns to precharge and 14 ns to evaluate in the worst case.

8 Branch Unit

The main function of the Branch Unit is to execute instructions that affect the Program Counter (PC), i.e., JUMP, CALL, BRF, and BRT. The block diagram of the Branch Unit is shown in Figure 26. The JUMP instruction changes the PC by loading it with *src1*. The CALL instruction is similar to JUMP except that it also saves a return address into *dest*. For example, `CALL R4,R6` loads the PC with the content of R4 and saves the return address into R6. The return address is the address of the instruction following the delay slot of the CALL in the input program. Branch instructions specify a 16-bit signed offset (O) that is added to the content of the PC to compute the address of the target of the branch. The branch condition is the LSB (least significant bit) of the *src1* register that is specified by the instruction. For the BRT instruction, the PC is loaded with $PC + O$ if the LSB of *src1* is high; for BRF, the PC is loaded with the target address of the branch if the LSB of *src1* is low.

The layout of the Branch Unit follows the structure shown in Figure 26. The branch offset enters the Branch Unit from the right and is distributed down to each bit-slice, with the MSB going to bits 7–15 for sign extension. The sign-extended branch offset is latched into BOR (Branch Offset Register). The content of BOR is added to that of the PC to compute the target address of a branch instruction. This target address is held in the Target PC (TPC) latch. Also, the PC is incremented, and the result is saved in the NPC (Next PC) latch. At the end of Ph4, depending on the instruction that is being executed, the PC latches one of the following values:

1. `0000H` when the processor is reset
2. *src1* for a CALL or JUMP instruction
3. TPC for a taken branch instruction
4. NPC

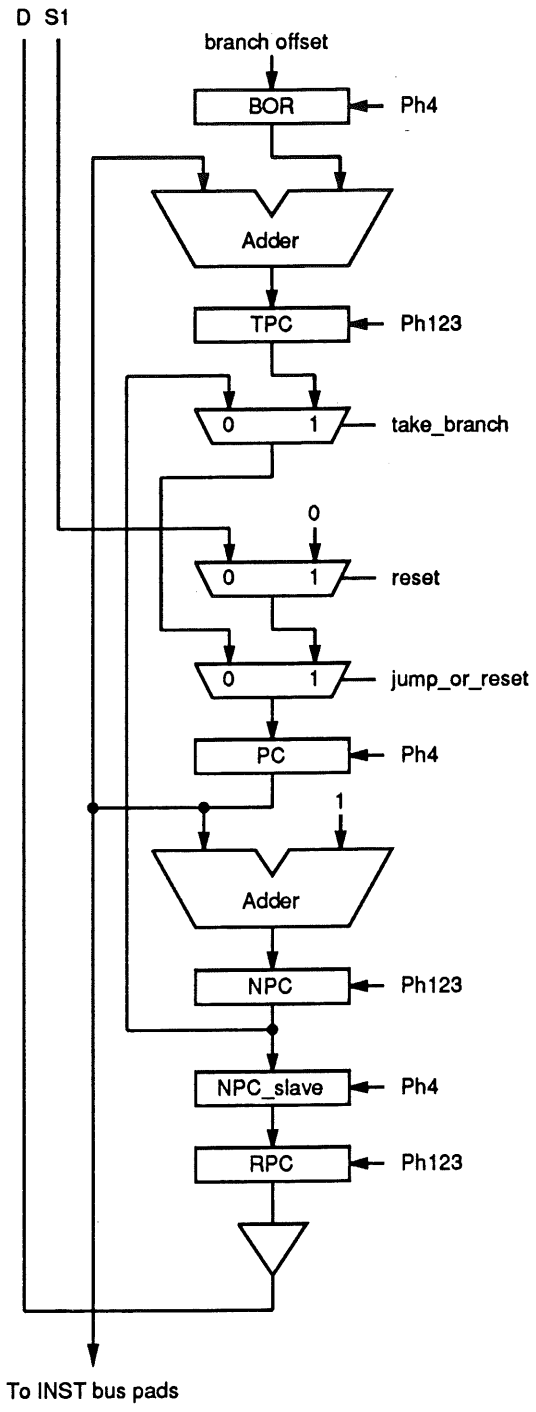
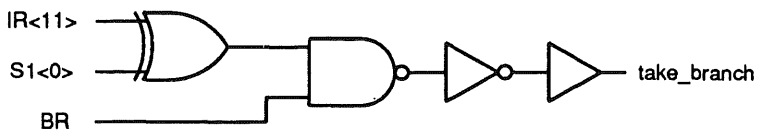


Figure 26: Block Diagram of the Branch Unit



BR is high for branch instructions.
IR<11> is bit 11 of the instruction Register.

Figure 27: Circuit Diagram of the Branch Evaluation Logic

The value of the PC is driven onto the INST bus during the subsequent instruction fetch step. The RPC (Return PC) latch holds the return address for CALL instructions. The content of RPC is driven onto the D bus and saved into *dest* for CALL instructions.

The branch evaluation logic shown in Figure 27. This logic is not conducive to being designed in a bit-sliced style, and is thus placed to the right of the data path. Bit zero of the S1 bus is passed out of the data path to this logic.

The two adders used in the Branch Unit have the same timing as the ALU; in fact, they utilize the same carry chain that is used in the ALU. In order to keep the number of branch delay slots to one, the branch target address and the branch condition must be evaluated by the end of the ID stage. Both of these tasks are performed in parallel during the ID stage, and the results are ready by the end of Ph123. The value to be loaded into the PC is selected during Ph4.

9 Clock Generator

The clock generator produces the internal clock phases from which the timing of the entire processor complex is derived. In total there are ten distinct clock signals that are generated. There are four non-overlapping clocks with equal pulse widths (Ph1, Ph2, Ph3, and Ph4) and a fifth clock which is continuously high during Ph1, Ph2, and Ph3, called Ph123; in addition, each clock has a global complement. The Ph123 clock is used specifically for the evaluation phase of the dynamic circuitry used for the computations performed in the ALU, the Shifter Unit, and the Branch Unit because they take the largest part of each machine cycle. It should be noted that the falling edge of Ph123 must precede the rising edge of Ph4, hence the distinction between Ph123 and

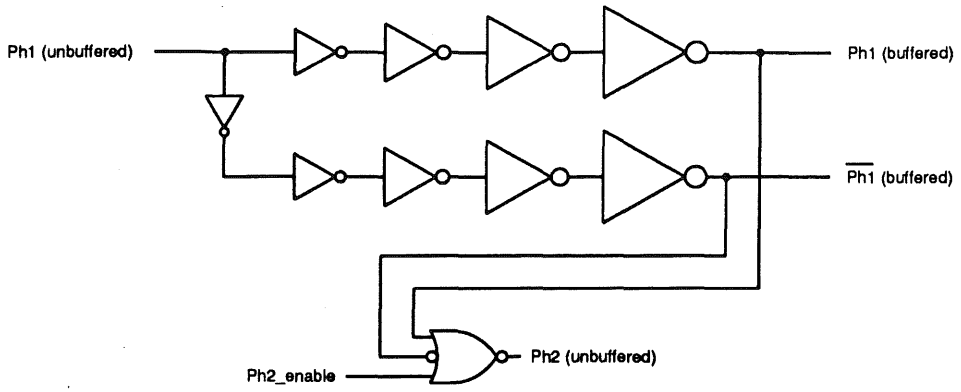


Figure 28: Circuit to Ensure Non-Overlapping Clock Phases

the complement of Ph4. Global clock complements were provided in order to allow the designers of the data path blocks to select the polarity of a clock signal without the additional logic and skew that would be introduced by locally inverting the clock. Because of the regularity of the data path design, the required additional clock routing was minimal.

In order to guarantee non-overlapping clock pulses, the activation of each clock signal is conditional upon the deactivation of the previous phase. This is accomplished with a NOR gate, the inputs to which are a phase enable signal, the actual buffered signal of the previous clock phase, and its complement (see Figure 28). Only when all three are zero is the clock signal itself allowed to rise. In this way, there is a guarantee of some "dead time" in between clock phases. This dead time is approximately equal to the delay time of the buffer chain driving the clock to the rest of the chip and adequately compensates for the skew that might be introduced by RC delay in the clock lines themselves. The clock enable signals are produced by a four-stage ring counter which is initialized upon reset to contain a '1' in the first flip-flop and '0' in the other three. The '1' circulates around the ring counter at the frequency of the external clock, producing a series of pulses that are similar to the four clock phases but are not guaranteed to be non-overlapping; the complements of these pulses are fed to the NOR gates as the clock enable signals. Thus the operating frequency (the inverse of the instruction cycle time) is one-fourth the frequency of the input clock. Figure 29 shows the complete circuit diagram of the clock generator.

The five positive-polarity clock signals were brought out to the pins of the

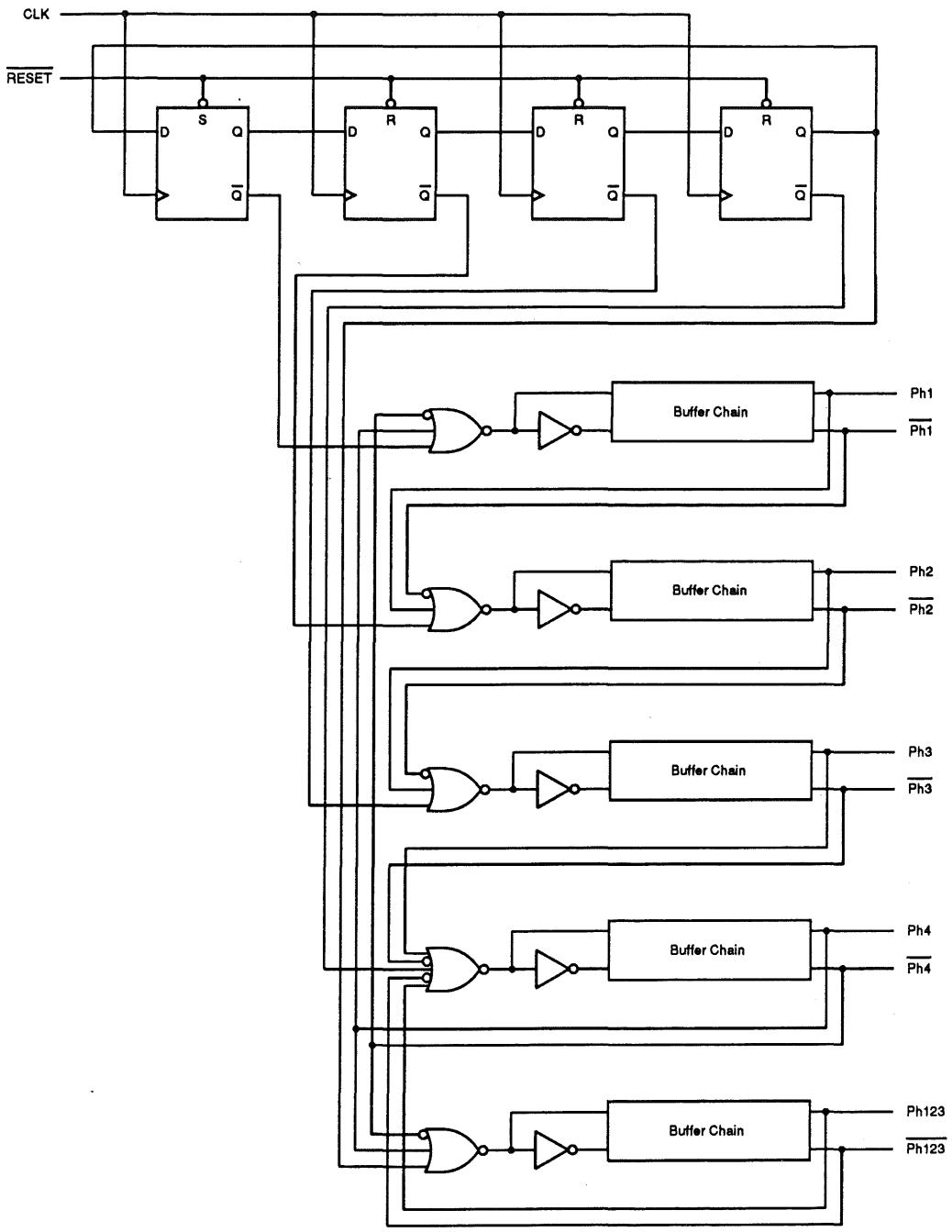


Figure 29: Circuit Diagram of the Clock Generator

chip to verify proper operation of the clock generator. With an input clock of 20 megahertz, the dead time was measured and averaged 7 ns.

10 Simulation and Testing

In order to verify the functionality of our design, we developed a simulation environment that allowed us to easily compare the expected state of the processor to the actual state produced by a circuit or switch-level simulator.

The first component that was needed for this method was a means of producing the expected result. This was accomplished by writing an RTL (Register Transfer Level) simulator in C (TPSIM). The input to TPSIM is an assembly program. TPSIM simulates the execution of the input program and computes the expected state of the processor at the end of each machine cycle. The output of TPSIM is written to a file. A helpful feature of TPSIM is that it allows random data to be loaded into the processor; all load instructions that access memory address zero receive a random number. This allows long, non-repeating test programs to be written conveniently using small looping sequences of instructions.

IRSIM was used to simulate the extracted layout of the entire processor (including the pad frame). IRSIM was used in the linear mode. The simulation process is driven by a command file that supplies stimuli to IRSIM. The command files are generated by TPSIM, which also produces test vector files that were used to test the fabricated chips. The command files also contain commands that instruct IRSIM to produce an output file that contains the state of the processor at the end of each machine cycle. This output file, along with the expected state file generated by TPSIM, are then passed to the comparison stage of the testing system.

The SIMCOMP program was written to compare the output state file produced by IRSIM to the expected state file generated by TPSIM. SIMCOMP detects and reports any mismatches that it encounters. SIMCOMP produces a file in which the expected and actual state of the processor are compared. All errors are visibly marked; this allows designers to easily track down the errors and solve the associated problems.

11 Conclusion

We have presented the VLSI design of a RISC-style 16-bit microprocessor. The microprocessor was fabricated in a $2\mu m$ n-well CMOS technology. The chips were tested and found to be working on first silicon. The processor has a cycle time of 70 ns and achieves a peak performance of 14 16-bit MIPS. The design required about 12,000 transistors.

12 Acknowledgments

We are grateful to Western Digital for providing two generous fellowships to our students. Moreover, they supported us with the testing facility for the final phase of this project. Finally, we would like to thank NKK for their donations to our Advanced Computer Architecture laboratory.

References

[1] C. Tomovich, *MOSIS User Manual*, Release 3.1, USC/Information Sciences Institute, Marina Del Rey, CA, 1988.

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

[3] A. Salz and M. Horowitz, "IRSIM: An Incremental MOS Switch-Level Simulator," *Proceedings of the 26th Design Automation Conference*, pages 173-178, ACM/IEEE, Las Vegas, Nevada, June 1989.

[4] W. S. Scott, R. N. Mayo, G. T. Hamachi, and J. K. Ousterhout, *1986 VLSI Tools: Still More Works by the Original Artists*, Report UCB/CSD 86/272, University of California at Berkeley, Berkeley, CA, December 1985.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Palo Alto, 1990.

[6] 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*, MIT, January 1982.

[7] A. Weinberger, "Large Scale Integration of MOS Logic: A Layout Method," *IEEE Journal of Solid State Circuits*, April 1967.

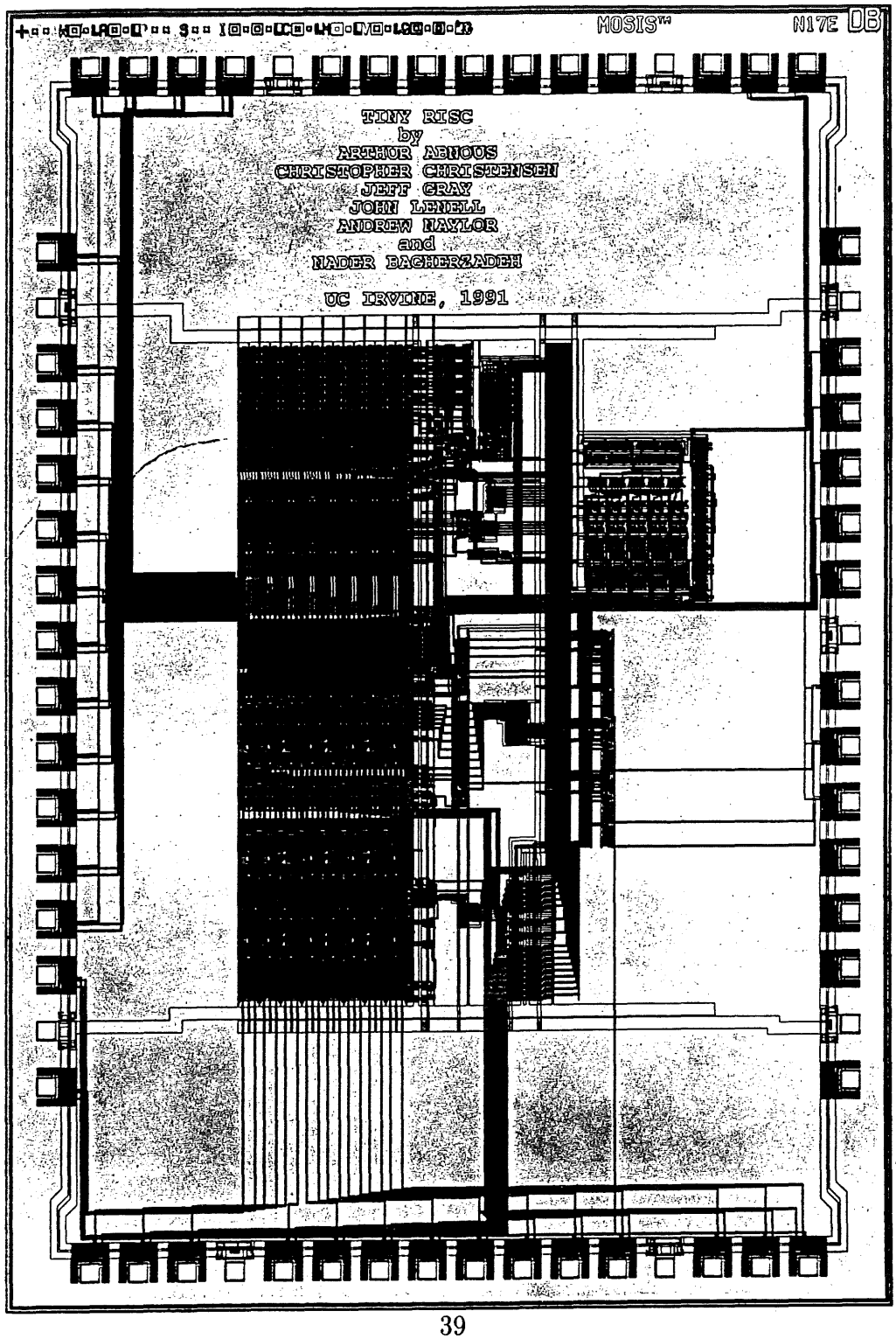
[8] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, Reading, MA, 1985.

A RTL Description

This appendix contains the RTL description of the Tiny RISC.

```
I (Immediate) is a 5-bit immediate value.
LI (Long Immediate) is an 8-bit immediate value.
IMM is the actual immediate value constructed from I or LI by zero-extension.
src1, src2, and dest are register addresses.
R[0..7] are general purpose registers.
R[0] is hardwired to contain zero at all times. Writing to it has no effect.
S1_bus is the src1 bus.  S2_bus is the src2 bus. D_bus is the dest bus.
DEST is a latch that contains the data to be written to the register file.
MADR is the Memory Address/Data Register.
INST_bus is the external instruction bus.
DATA_bus is the external data bus.
IR[0..2] contain the instructions being processed in the pipelined. They form
a shift register.
IR[].op_type is the type of the instruction in the instruction register.
Instruction types include "compute", "load", "store", "branch", "jump", "call".
IR[].opcode is the opcode of the instruction in the instruction register.
PC is the Program Counter.
NPC is the Next PC (PC + 1).
NPC_slave is the slave latch of NPC.
TPC is the Target PC. It contains the target address for a branch instruction.
RPC (Return PC) contains the return address for a CALL instruction.
BO is the Branch Offset field of branch instructions.
BOR is the Branch Offset Register.

IF_4:  IR[2] <= IR[1] <= IR[0] <= INST_Bus
       BOR <= BO directly from INST_Bus

ID_1:  decode instruction
       perform bypass comparison
       IMM <= I  or  IMM <= LI
       PC incrementer and branch offset adders start evaluating
ID_2:  precharge register file
       decode register addresses
ID_3:  read R[src1] and R[src2] from register file
       NPC <= output of PC incrementer
       TPC <= output of branch offset adder
ID_4:  S1_bus <= R[src1]  or  S1_bus <= D_bus
       S2_bus <= R[src2]  or  S2_bus <= IMM  or  S2_bus <= D_bus
       NPC_slave <= NPC
       if  IR[1].op_type = "jump"  or  IR[1].op_type = "call"  then
          PC <= S1_bus
       else if  IR[0].opcode = "BRF"  and  S1_bus(0) = 0  then
          PC <= TPC
       else if  IR[0].opcode = "BRT"  and  S1_bus(0) = 1  then
          PC <= TPC
       else
          PC <= NPC

EX_1:  ALU and shifter start evaluating
       MADR <= S1_bus
       INST_bus <= PC
       if  IR[1].op_type = "load"  or  IR[1].op_type = "store"  then
```

```
          DATA_Bus <= MADR
EX_2:   MADR <= S2_bus
        INST_bus <= Hi-Z
        if  IR[1].op_type = "store"  then
          DATA_Bus <= MADR
        else
          DATA_bus <= Hi-Z
EX_3:   latch output of ALU, output of shifter, and data from memory
        RPC <= MPC_slave
EX_4:   if  IR[1].op_type = "compute"  then
          D_bus <= output of ALU or shifter
        else if  IR[1].op_type = "load"  then
          D_bus <= data from memory
        else if  IR[1].op_type = "jump"  or  IR[1].op_type = "call"  then
          D_bus <= RPC
        DEST <= D_bus
        decode register address

WB_1:   R[dest] <= DEST
WB_2:
WB_3:
WB_4:
```


MOSIS
N17E
DB
TINY RISC
by
ARTHUR ABNOUS
CHRISTOPHER CHRISTENSEN
JEFF GRAY
JOHN LENELL
ANDREW NAYLOR
and
NADER BAGHERZADEH
UC IRVINE, 1991