AN ABSTRACT OF THE THESIS OF

Licheng Zhang for the degree of Master of Science in Electrical and Computer Engineering presented on June 7, 1989.

Title: The Design of A Reduced Instruction Set Computer Using A Silicon Compiler.

Redacted for Privacy

Abstract approved: _____
John Murray

The objective of this thesis is to describe the design and implementation of a VSLI reduced instruction set computer (RISC). The RISC machine constitutes a new style of computer architecture. It differs significantly from the complex instruction set computer architectures (CISC) of the past. RISC architectures are characterized by their high performance, simple instruction sets, minimal hardware requirements, and their ability to support block structured programming languages adequately.

In this thesis a 16-bit single chip RISC was designed using the Genesil Silicon Compiler. It has 14 instructions, an overlapped register window structure, and on chip memory. It can execute most instructions in a single clock cycle, including procedure calls and returns. The peak performance of this chip is approximately 6 MIPS. The chip was implemented in 2 micron CMOS technology. The chip size is 516.54 X 514.27 mils. This chip has not been fabricated.

THE DESIGN OF A REDUCED INSTRUCTION SET COMPUTER USING A SILICON COMPLIER

By

Licheng Zhang

A THESIS

submitted to

Oregon State University

in partial fulfillment of

the requirement for the

degree of

Master of Science

Completed June 7, 1989

Commencement June, 1990.

APPROVED:

Redacted for Privacy

Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented: June 7, 1989.

TABLE OF CONTENTS

TABLE OF FIGURES

THE DESIGN OF A REDUCED INSTRUCTION SET COMPUTER

USING A SILICON COMPLIER

1. INTRODUCTION.

The reduced instruction set computer or "RISC" computer is a new style of computer architecture which was developed in the late 70's and early 80's. RISC computers possess a small, simple instruction set. All instructions have the same format, and can be executed efficiently. Due to their simplicity, they are ideally suited to VLSI implementation. RISC architectures have become more popular over the years primarily due to their high performance capabilities.

This thesis deals with the architecture and design of a single chip 16-bit RISC computer. It is similar to the RISC architecture originally conceived at the University of California, Berkeley [1]. It implements 14 instructions and contains a three-bus datapath, a controller, a register file, and on-chip RAM and ROM. The architecture block digram of this single chip RISC is shown in figure 1.1.

Until recently, the strategy used for making fast computers was to implement a complex instruction set machine. Such complex instruction set computers or "CISC" computers were intended to efficiently support high level languages, so that complex operations could be achieved by executing a single instruction, instead of

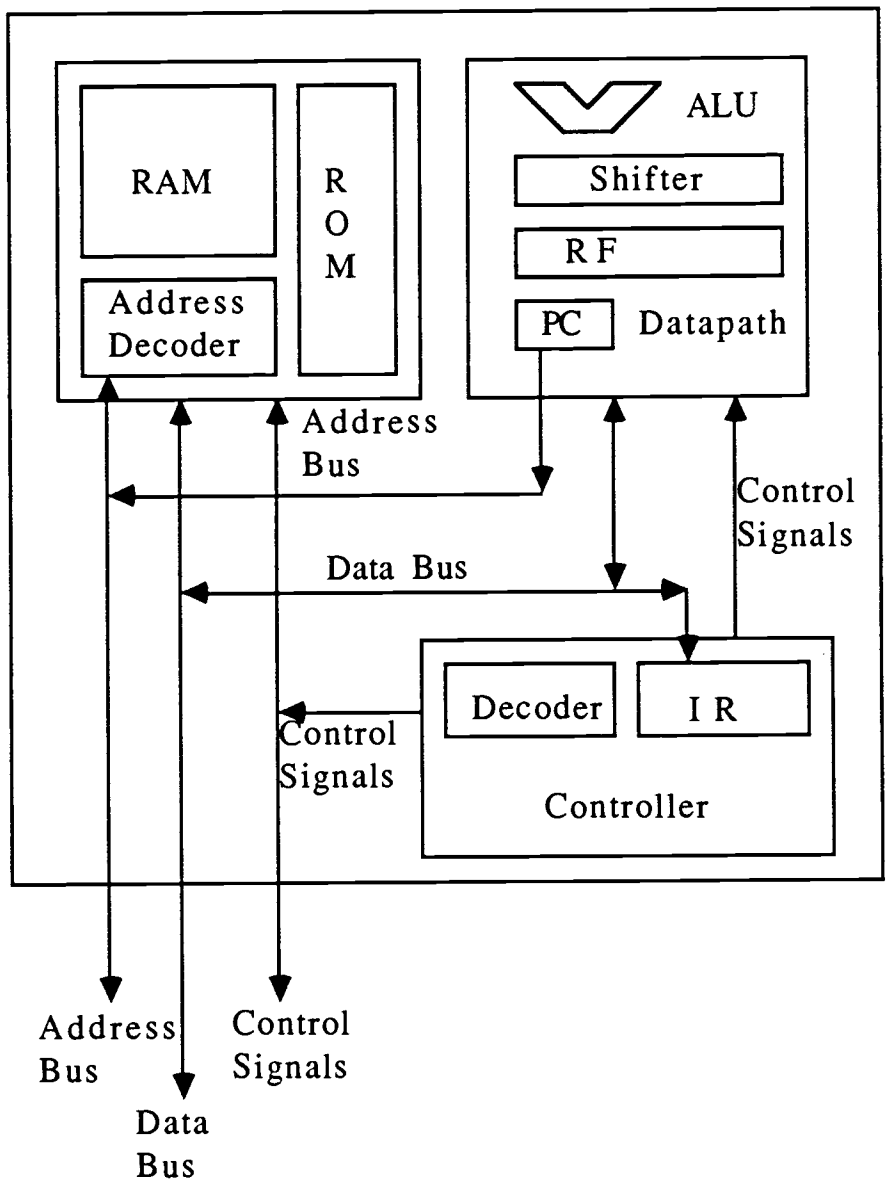


Fig. 1.1. A Single Chip RISC Machine

several simple instructions. Research in the late 1970's showed that although CISC machines can execute complex operations in one instruction, overall system performance is not necessarily high as a result.

A number of studies indicate that in CISC systems such as the DEC VAX, certain instructions such as the move, call subroutine, and conditional branch, are executed much more frequently than other instructions. This situation was found to be true over a wide range of user application programs. It was also found that some of the instructions that are executed most frequently are more time consuming to execute than other instructions of this class. These studies showed statistically that system performance depends more on efficiently executing the instructions which are executed most frequently than in having a wide repertoire of complex instructions.

Additional reasons for re-examination of the CISC paradigm included problems associated with CISC implementation. Among these problems are the fact that individual CISC instruction complexity varies widely. It is therefore not possible to make all instructions use a single word instruction format. Multi-word instructions require, by definition, additional memory access cycles. Fetching these additional instruction words from memory degrades system performance significantly. This fact, coupled with the wide range of possible instruction formats, makes the

instruction decoding process very complicated. Secondly, CISC machines require very complex controller hardware due to the sheer volume of instructions. Typical CSIC machines contain 100-300 multi-word instructions. In contrast, RISC machines typically include 30-70 single-word instructions. The wide range of possible instruction addressing modes found in CISC machines compounds the hardware problem to an even greater degree. CISC machines are also difficult to pipeline without incurring extraordinary chip area overhead penalties due to the specialized pipeline hardware required. The additional hardware required in CISC machines increases overall chip area and reduces system performance. Finally, it is a time consuming, expensive, and error-prone process to develop CISC CPUs due to their sheer complexity [2].

With these considerations in mind, the new RISC architectures were developed in late 70's and early 80's. The key motivation for these RISC computer implementations stemmed from a fervent belief that simpler VLSI-based RISC machines would yield higher performance in most application contexts. These RISC machines optimize the execution of simple, frequently used instructions through the use of specialized hardware mechanisms. In practice, the assumptions of the RISC pioneers have proven correct. A large number of computing machines recently released and currently under development employ RISC architectural principles.

It is important to note that when building computer systems, current programming language structures and available software development technologies are a key consideration. At approximately the same time that RISC hardware architectures were being introduced into the marketplace, programming language compiler technologies became available which were capable of transforming complex high level language operations into simple RISC instructions efficiently. These new compiler technologies were able to deal with the difficult compilation issues introduced by such highly pipelined machines. Thus, software development technologies came into existence simultaneously with the emerging RISC hardware architectures.

The main functional characteristics of RISC computers are as follows:

1) RISC computers execute one instruction per clock cycle. This includes jump, call, and return instructions.

2) All instructions are the same size. All instructions have the same format.

3) Only certain instructions (e.g. load and save) access memory. All other instructions perform register to register operations.

4) Many RISC computers contain hardware features to optimize block structured programming language execution.

Examples include large register arrays and register windows for efficient subroutine (procedure) implementation.

In order to design and implement a RISC computer in a reasonable amount of time with the minimum possible chip area, it was necessary to take advantage of the latest in computer-aided design technology. A commercial silicon compiler was used for this purpose. The silicon compiler is a software package which allows a designer to implement digital systems on silicon from well-defined parameterized building blocks contained in the compiler's library. The silicon compiler also provides the designer with the capability to functionally (logically) simulate the operation of the resulting chip, establish the chip's performance, and send the chip design file, via electronic mail, to a silicon foundry for fabrication.

2. REDUCED INSTRUCTION SET COMPUTER ARCHITECTURE.

2.1. From CISC to RISC.

Since the days of the earliest digital computers, instruction sets have tended to grow larger and more complex. The MARK-1 in 1948 had only seven instructions. They were very simple instructions, like add and jump. By contrast, a VAX in the 1980's has 278 instructions, and some of its instructions are very complicated. The reasons for this trend are many. Among these are the desire simplify compiler construction, the ability to better support high level languages, and attempts to improve system performance.

As computers have evolved, high level languages (HLLs) have become more powerful and complex. These high level languages allow programers to express their algorithms more concisely, and support the use of block structured (hierarchical) programing techniques. These activities have enlarged the differences between operations provided in the high level languages and those provided in the physical realization of the computer. This phenomena is known as the semantic gap. In order to reduce this semantic gap, computer architects enriched their instruction sets, adding more addressing modes, and implemented various high level language statements in hardware. Computer architectures which include such large, complex instruction sets

are called complex instruction set computers, or CISCs. Designers originally believed that CISC machines could simplify the task of generating language compilers, improve execution efficiency through the use of microcode to implement complex instructions, and provide better support for even more complex and sophisticated high level languages.

Over the years, a numbers of researchers have carefully analyzed the results of these CISC implementation efforts. Their results differ from what many computer designers had expected:

1) Most instructions in compiled programs are relatively simple. The most frequently used statement is the assignment statement (:=) or "move" instruction. The second most frequently used statement is the IF statement or "conditional branching" instruction.[1]

2) Most operand references are simple scalar variables, and most of these scalar variables are local.

3) With a large complex instruction set, it is hard to find an exact semantic match between the high level language and the available architecture. Since there are many possible choices and many ways to achieve this match, it is hard to optimize the generated code in such a way as to minimize physical code size. It is also more difficult to fully pipeline a machine with a complex

As discussed previously, RISC computers are different from CISC computers in several fundamental ways. First, RISC machines have simpler instruction sets than CISC machines. Second, many RISC machines have a large register file. RISC machines emphasize register rather than memory references in order to deal more effectively with local variables, and to reduce main memory traffic. Third, many RISC machines use some form of overlapped register windows, to handle procedure calls efficiently. Procedure calls constitute one of the more time consuming operations regularly performed by the CPU.

2.2. Characteristics of RISC Architectures.

There are many different approaches to the implementation of reduced instruction set architectures. Certain characteristics are common to all of them. These characteristics are as follows:

1) RISC machines execute one instruction per clock cycle. This includes fetching two operands from their respective source registers, performing appropriate ALU operations, and storing the results in the chosen destination register. Since instructions are executed in one cycle, there is almost no need for microcode. Machine instructions can be hardwired or implemented by an elementary finite state machine (FSM). Since there is no need to access a complex microprogram control store during the execution of a given instruction, instructions can be executed faster than on a machine with a microcoded controller.

2) Most operations in RISC machines are register-to-register. Typically only "load" and "store" instructions access the main memory. This simplifies the instruction set and control unit design considerably, and encourages the optimization of register use, so the most frequently used operands can be stored in very high-speed local storage. With an optimized compiler and a large register file, most operands can be held in the register file for a long time, thus reducing external or main memory access cycles. A typical register file in a RISC machine may contain 128 or more registers.

3) RISC machines incorporate simple addressing modes. Almost all instructions use register addressing. Some other simple addressing modes, such as displacement and PC-relative, may be included. Other complex addressing mode can be synthesized from these simple addressing modes.

4) All instructions have a fixed size. They generally use one or a small number of possible instruction formats. Field locations, especially the op code field, are fixed. This makes the design of both the instruction decoder and the control unit simpler.

The RISC architectural characteristics described above benefit system performance substantially. RISC architectures are also imminently suitable for VLSI implementation. If a machine is to have high performance today, it must be implemented in VLSI. Older implementation techniques, such as those in which

LSI/MSI/SSI components are interconnected on printed circuit boards, suffer from performance limitations imposed by off-chip communication delays. It is currently advantageous to place as much of a system on a single chip as possible in order to minimize any off-chip communication delays [3].

2.3. Overlapped Register Windows and Overflow/Underflow Handling.

Researchers have shown that procedure or subroutine calls are among the most time consuming operations associated with high-level language programs. Whenever a procedure call is performed, registers must be saved in memory on a stack and parameters must be passed to the procedure. When a return is performed, results must be passed back from the procedure and the registers must be restored from memory. This is especially important in the case of RISC architectures, because complex operations available through the execution of single instructions in a CISC machine are often implemented as subroutines in RISC machines. RISC machines potentially may have more calls than CISC machines. This fact may also establish an eventual upper limit on RISC performance.

In the execution of many high level language programs, it is common for procedures to be nested several levels deep. In the case of nested procedure calls, a set or group of registers within a register file may be used to maintain the parameters and data

associated with one particular procedure call. Other registers may be used to hold the parameters and data connected with subsequent procedure calls occurring within the original procedure.

A sophisticated way to organize small groups of registers within a register file in such a way as to reduce main memory traffic is known as an overlapped register window. This technique was developed in Berkeley in the early 1980's.

The register file is conceptually divided into two parts: global registers, which are not saved or restored on each procedure call; and the window, which is used by one procedure only. On each procedure call, only one window is visible. A new window is utilized on each new procedure call, and returns back to the old or previous window on each return instruction. Each window is divided into three fixed size parts: Low parameter registers, which hold parameters passed from the procedure that called the current procedure and the results to be passed back; Local registers, which are used for local variables; and High parameter registers, which are used to pass parameters and receive results from the next procedure called by the current procedure. All these windows used by the different procedures overlap, which means that the high parameter registers for the current window are physically the same as the low parameter registers for the next window. This allow parameters and results to be passed without actual data

movement from register to register. An overlapped register window is shown in figure 2.1.

In this thesis, a 16 word register file was implemented. It has four global registers, R0 to R3, and four overlapped windows. Each window has two local registers, one low parameter register, and one high parameter register. In this case, only the program counter can be passed from window to window.

As shown in Fig 2.2, the overlapped register window is circular. Therefore, when the procedure call nesting depth is larger than the number of windows, an overflow occurs. There are two hardware pointers in the controller indicating the status of the overlapped register window, CWP and SWP. The current window pointer, CWP, indicates which window is in current use. The save window pointer, SWP, indicates which window is going to be saved. So when CWP=SWP-1 and a procedure call is going to be performed, an overflow is about to occur. At this time, the oldest activations must be saved in memory. Additional time is required to save the registers in memory. In this project, only three registers must be saved in memory, so it only takes four extra clock cycles per overflow. Larger window will require corresponding more time to handle overflow conditions. When the procedure call nesting depth decreases, the old activation must also be restored from memory to perform the return instruction correctly. An underflow occurs when CWP = SWP and a return

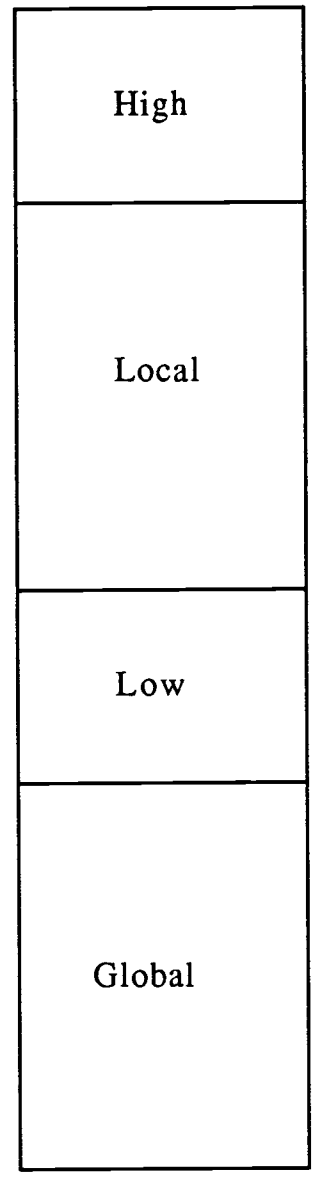


Fig. 2.1. Register Window

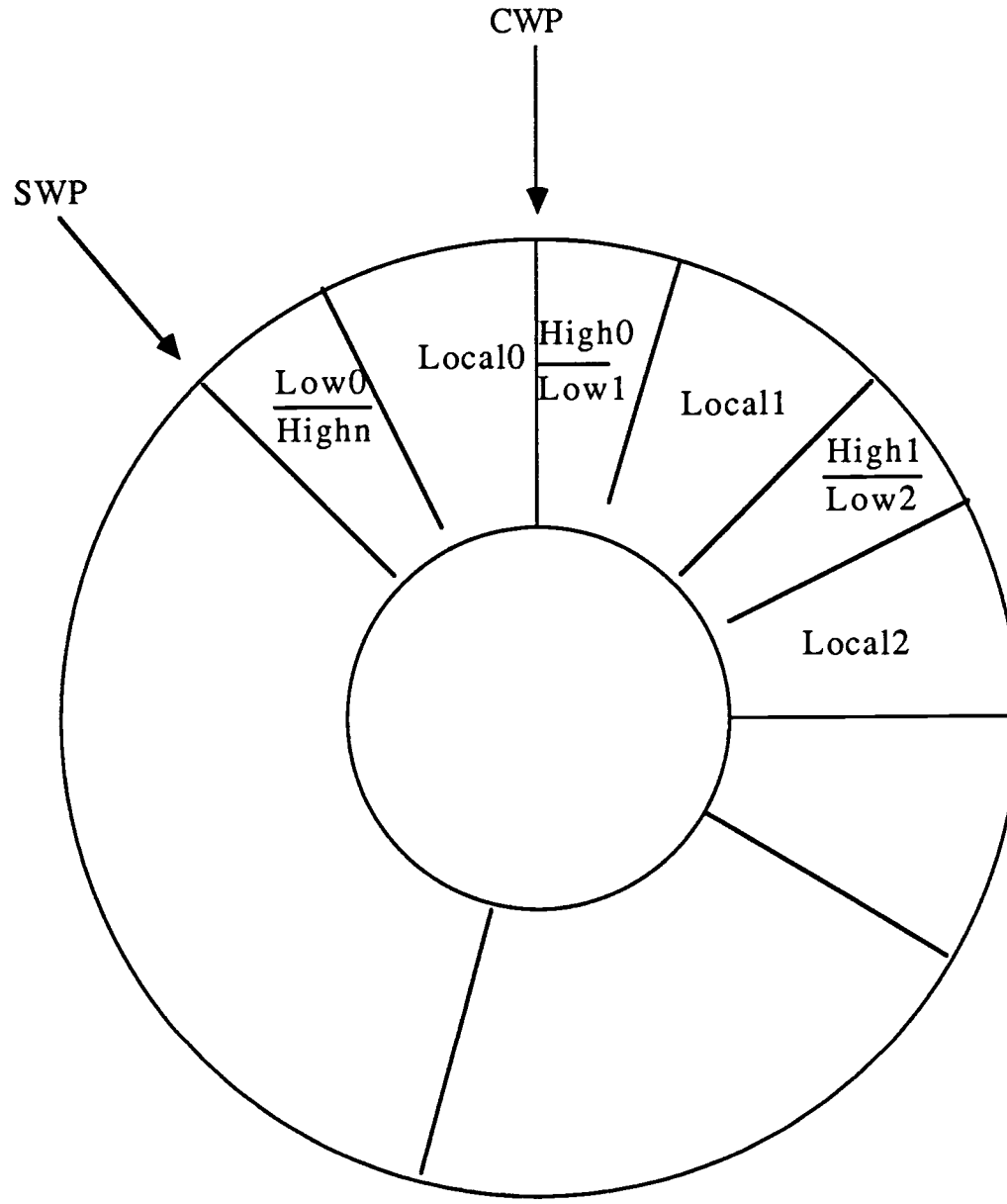


Fig. 2.2. Overlapped Register Window

instruction is about to be performed. At this point, data will be loaded into the window from memory. It costs same number of extra clock cycles to load the window as in the case of overflow.

3. DESIGN ENVIRONMENT AND METHODOLOGY.

To design a single chip VLSI computer is a very complex process. A top-down design strategy was used in this VLSI RISC design. The chip was decomposed hierarchically. The chip implementation was performed in a bottom-up fashion. The elements in the lowest level of the design hierarchy were synthesized using the Genesil Silicon Compiler, simulated, then incorporated into higher level structures or modules. These higher-level structures were then simulated and incorporated with other elements or modules to form even higher-level modules. This process was continued until the top level of the hierarchy, the chip level, was completed.

3.1. An Overview of The Genesil Silicon Compiler.

The Genesil Silicon Compiler system is an integrated VLSI computer-aided design system. It produces chip design files from their microarchitectural descriptions in the same way that software compilers produces machine code from high level language statements.

As a results of many years of VLSI design experience on the part of the individuals comprising the firm Silicon Compiler Systems, the Genesil Silicon Compiler contains most of the internal structures needed in VLSI chip design. It provides four major structural elements which are used to compose digital systems:

chip sets, the highest level object, which is made up of a collection of chips; chips, which are constructed by designer from lower level structures; modules, which are collection of blocks and other modules (including parallel datapath modules, random logic modules and general modules); and blocks, the lowest level design object, which encompasses such structures as RAM, ROM, PLA, etc. Each structure is highly parameterized, allowing much freedom in composition on the part of the designer. The relationship between these building blocks is shown in figure 3.1.

After using these building blocks to form appropriate structures at each level of the hierarchy, the designer utilizes the netlisting, floorplanning, and compilation tools. The netlisting tools are used to logically interconnect the structures. The floorplanning tools are used to define their proper place on the chip. Additional tools create the geometric design files necessary for chip production. The Genesil Silicon Compiler also provides functional (logical) simulation and timing analysis capabilities. The designer can use these verification tools to simulate the functions of and/or analyze the performance of those functional blocks at each level of the hierarchy.

3.2. Chip Design Methodology Using The Genesil Silicon Compiler.

The chip design process started with an analysis of the desired system architecture. The RISC architectural specification, which included the instruction set, associated addressing modes,

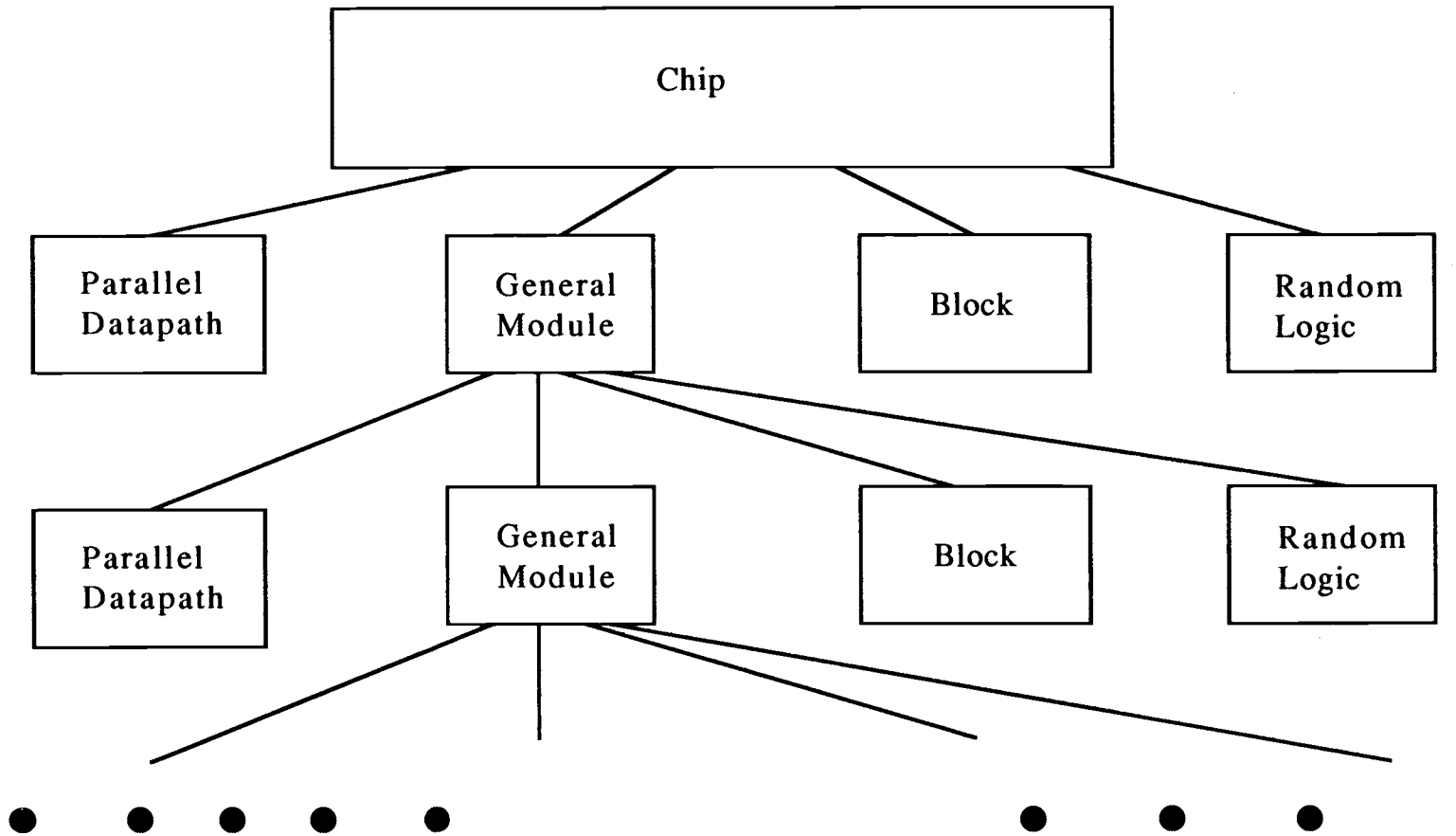


Fig. 3.1. Genesil Design Hierarchy

and the register file system was then transformed hierarchically to yield the desired microarchitecture for the machine. Each element of the microarchitecture is a Genesil block library element or group of these elements. It is important to note that for every architecture, there are many possible microarchitectural implementations. Chosing a suitable microarchitecture from the many possibilities is one of the greatest challenges of the design process. In this stage of the design process, most of the effort was focused on creating the core elements of the central processing unit, achieving a functional pipeline, assuring that instruction execution in one clock cycle was achieved, and establishing proper overflow/underflow handling. Finally, all these microarchitectral structures were compiled, simulated, and integrated using the Genesil Silicon Compiler to form a chip. Physical fabrication did not take place due to cost considerations. The last step in the design process involved chip level simulation, timing analysis, and plotting of the chip layout for viewing purposes.

The details associated with each step of the design process for the RISC chip are as follows:

1) Chip level definition. The fabline and package for the chip were selected.

2) Module specification. The detailed definition of the functions for each module or block were specified.

3) Simulation. Each module or block was functionally or logically simulated. This process included creating a test vector input file, applying the test vectors to the module or block, observing the outputs or results, and comparing the actual results with the expected results. The test vectors consisted of appropriate patterns of logical 1's and 0's.

4) Module Netlisting. The logical interconnections between all modules and blocks at the various levels of the hierarchy were specified.

5) Floorplanning and final chip compilation. After all modules were properly connected and simulated, floorplanning was used to move the design objects on the chip to their desired geographic locations and their proper orientations were established. The final chip design file was then compiled.

6). Timing analysis. The timing analyzer was used to check the performance of the chip and all modules and blocks within the chip. Included in the timing analyzer results at each level of the hierarchy were maximum clock rate, input setup and hold times, propagation delays, and critical timing paths throughout the chip.

7). Tapeout. During tapeout a geometric design file was created which could be transferred directly to an IC foundry for fabrication purposes.

The design hierarchy for the RISC chip in this thesis contains four levels and is shown in figure 3.2. The first level is the chip level. It contains a general module and input/output pads for the chip. The second level contains a datapath, a controller, and the memory. The third level contains operational modules which were used to form modules in the second level. In the case of the datapath module, the datapath contains two parallel datapaths to form a three busses structure architecture. In the controller module, an instruction register, instruction decoder, pointers, etc. are included. The memory module consists of a RAM, ROM, and memory address decoder. The fourth level of the hierarchy contains basic operational elements forming some of the modules in the third level. As an example, the pointers module in the controller module contains three pointer registers, the current window pointer, stack pointer, and saved window pointer.

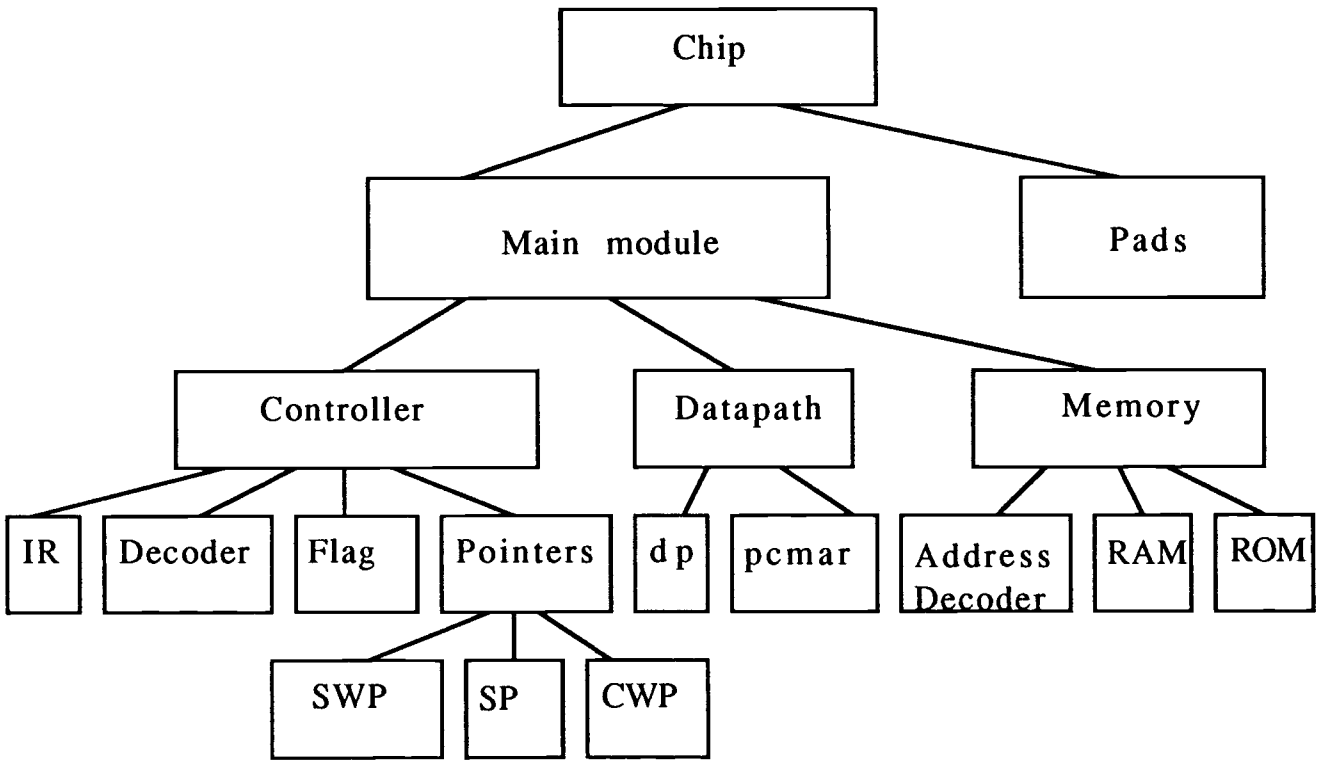


Fig. 3.2. Chip Hierarchy

4. SYSTEM DESIGN AND IMPLEMENTATION.

4.1. System Overview.

In this project, a Berkeley RISC I type reduced instruction set single chip computer was designed and implemented on a silicon compiler. It is a 16 bit architecture, with 14 basic instructions, overlapped register windows, overflow/underflow handling, and on chip memory including RAM and ROM. The block diagram of this computer is shown in figure 4.1. A simple two stage pipeline is implemented in this computer as well.

4.2. Instruction Set Design.

The RISC machine contains 14 instructions. They are ADD, SUB, AND, OR, NOT, Shift Left Logical (SLL), Shift Right Logical (SRL), Load (LD), Store (ST), JUMP, CALL, Return (RTN), Load High (LDH), and No Operation (NOP). All instructions are register-to-register except the Load and Store instructions. Load and Store instructions move data between memory and registers. The effective memory address is calculated using the contents of two registers or one register plus an immediate number. The Load High (LDH) instruction loads an immediate number contained in the instruction to the high eight bits of the specified destination register. Details of these instructions are shown in figure 4.2.

4.3. Instruction Format.

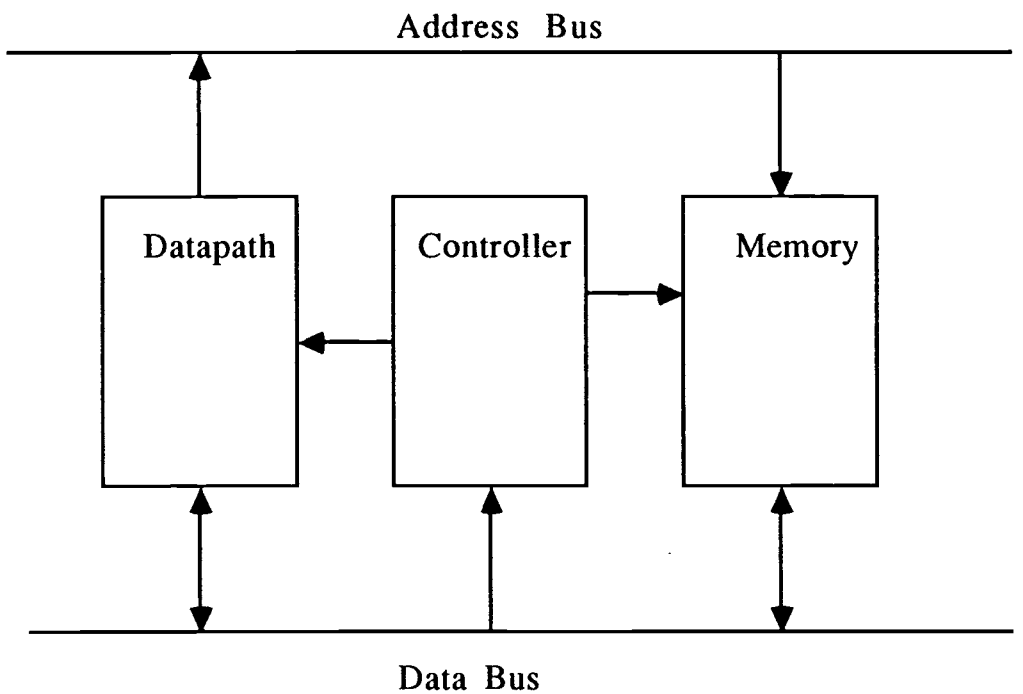


Fig.4.1 System Block Diagram

Instruction	Operands	Operation
ADD	DEST,SRC1,SRC2	DEST ← SRC1+SRC2
SUB	DEST,SRC1,SRC2	DEST ← SRC1-SRC2
AND	DEST,SRC1,SRC2	DEST ← SRC1&SRC2
OR	DEST,SRC1,SRC2	DEST ← SRC1 | SRC2
NOT	DEST,SRC1	DEST ← $\overline{\text{SRC1}}$
SLL	DEST,SRC1	DEST ← SRC1 shifted by 1
SRL	DEST,SRC2	DEST ← SRC2 shifted by 1
JUMP	COND,SRC1,SRC2	pc ←SRC1+SRC2
CALL	DEST,SRC1,SRC2	DEST ← pc pc ← SRC1+SRC2 CWP ← CWP+1
RTN	DEST	pc ← DEST CWP ← CWP-1
LD	DEST,SRC1,SRC2	DEST←Mem[SRC1+SRC2]
ST	DEST,SRC1,SRC2	Mem[SRC1+SRC2] ←DEST
LDH	DEST, Immediate	DEST ← Immediate
NOP	None	None

Fig. 4.2. Instruction Set

As indicated before, instructions, data, address and registers are all 16-bit quantities. All instructions are one word. There are few instruction formats used in this design. These formats are shown in figure 4.3. In the instruction, the OPCODE field contains 4-bits, indicating the operation to be performed. The DEST field contains 3-bits, indicating one of 8 internal registers as the destination of the result of a particular computation. The SRC1 field contains 3-bits, indicating a register containing one of two operands. Another operand is indicated by the SRC2 field. If the IMM field is zero, the register containing the second operand is indicated by the last 3-bits of the SRC2 field. If IMM field is one, SRC2's 4-bits is an immediate number.

For Jump operations, the Set Condition Code (SCC) bit indicates if the jump is conditional jump or not. SCC=0 implies an unconditional jump; SCC=1 yields a conditional jump. The condition for the jump is indicated by the DEST field. We can perform up to eight different conditional jumps using this approach, but only two of them were implemented in this thesis. The possible jump instructions include jump on carry if DEST=xx0, and jump on negative if DEST=xx1

For procedure Call instructions, the DEST field indicates the destination register for the PC. The DEST field indicates the source register for the PC in a Return instruction. The source register for

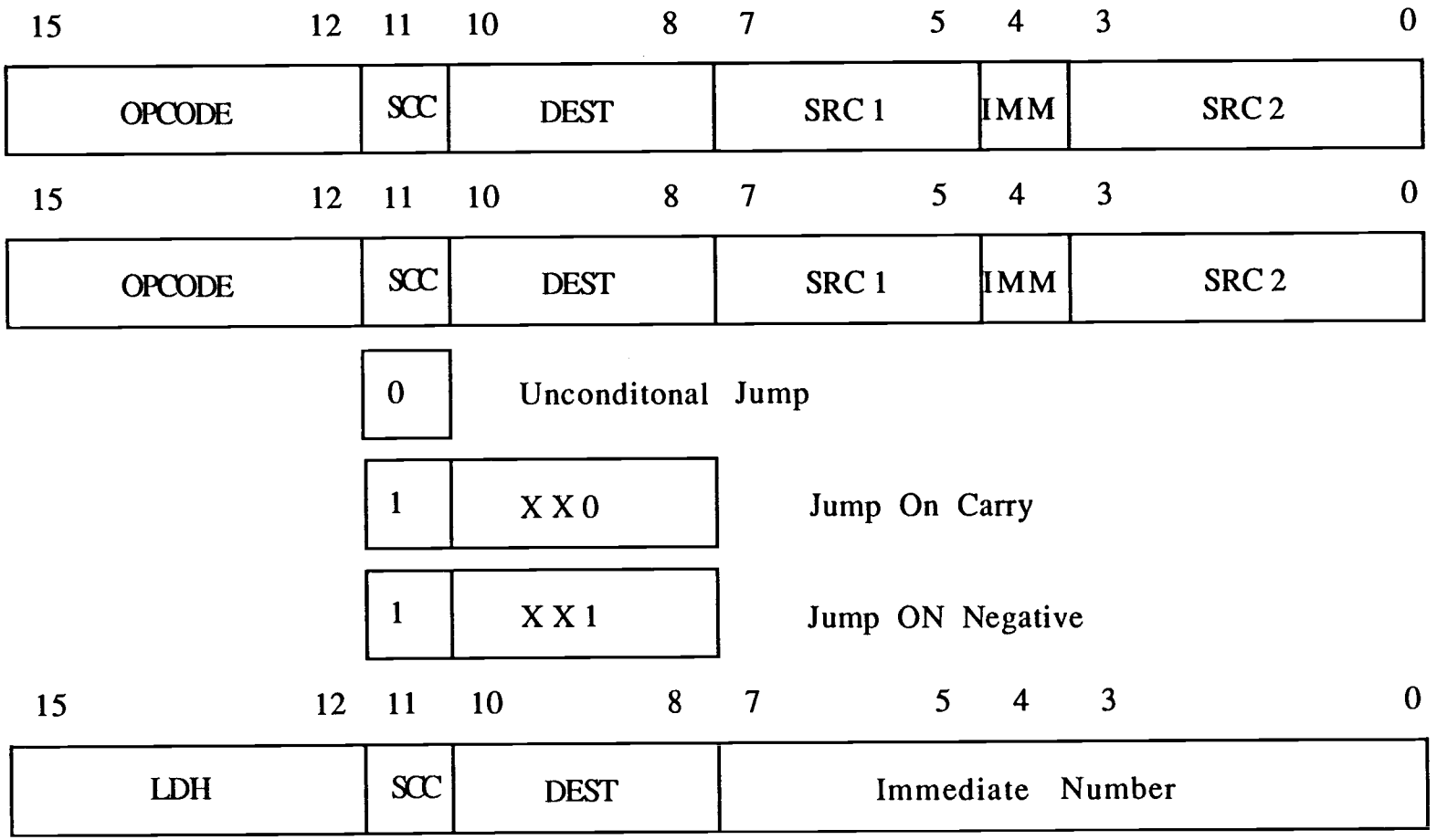


Fig. 4.3. Instruction Format

Shift Left Logical is defined in SRC1; for Shift Right Logical, it is defined in SRC 2.

4.4. Pipelining.

A two stage pipeline, which implements an elementary instruction prefetch function, is used in this RISC. This implies that while the machine is executing one instruction, the next instruction in the program is being fetched from memory. The time for fetching an instruction and that of instruction execution are the same, that is, one clock cycle. It is therefore best to use a two stage pipelining mechanism in this design. The pipeline timing for the machine is shown in figure 4.4.

It is important to note that pipelining will cause problems with proper instruction execution when a branching instruction like a jump, call, or return is executed. This problem arises because the instruction which is supposed to be executed next is not necessarily the one immediately following the branching instruction in the program. A delayed branching mechanism is used in these cases. A NOP operation is inserted after the branching instruction.

Pipelining will also cause a problem when a Load or Store instruction is executed, because these two instructions require two clock cycles for their execution. During these two clock cycles, only one instruction is supposed to be fetched. In this RISC design under consideration, during the first clock cycle of the load and store

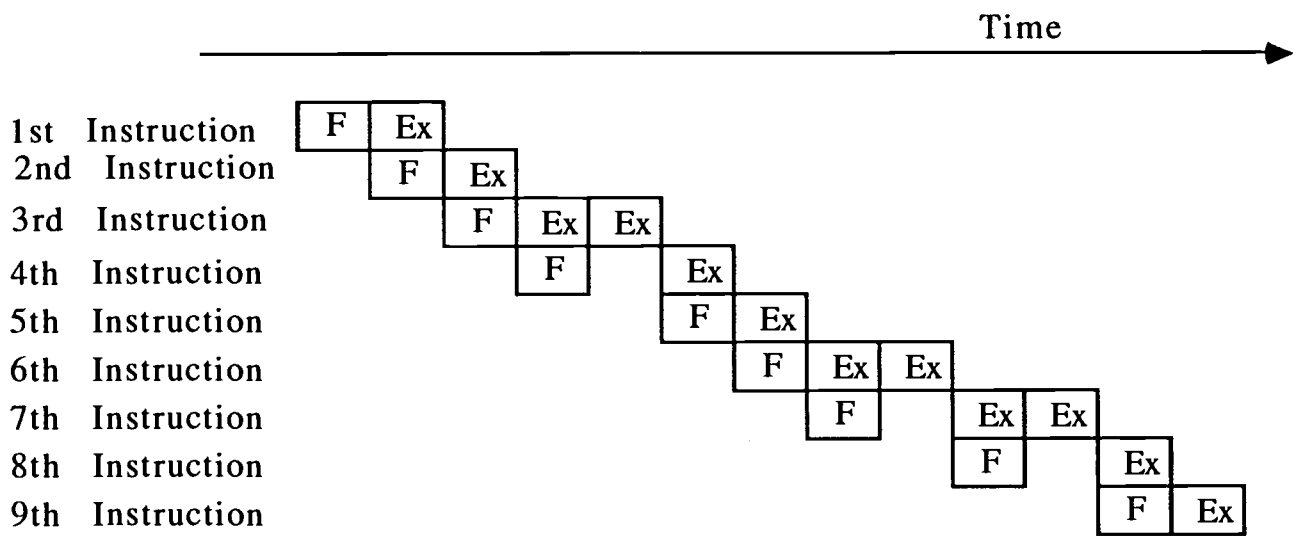


Fig. 4.4. Pipeline Timing

instructions, the effective address of the operand is calculated, and the next instruction fetched; the second cycle is used for moving data to or from memory.

4.5. Datapath Implementation.

The Datapath is the heart of the RISC machine. It is the main operational module, and consists of the ALU, Barrel Shifter, Register File, PC, MAR, and MDR. An overlapped register window is implemented in the register file. The block diagram of datapath is shown in figure 4.5.

In order to achieve an instruction execution time of one cycle, it was necessary to use a multiple bus system inside the datapath. As shown in figure 4.5, two buses are used to send operands from the register file to the ALU or Shifter simultaneously. Another bus is used to send the result to the register file. It is therefore possible to perform all of these operations in one clock cycle. Unfortunately, the parallel datapath module in the Silicon Compiler only has two internal global buses and two standard local interconnections. As we can see in figure 4.5, the datapath in this project has three (or four) buses. The solution for this problem was to use two parallel datapaths in the Silicon Compiler, then netlist them together to form a conglomerate datapath module. Up to four buses can be generated in this manner, with four standard local interconnections. Each bus in this design is precharged for higher performance.

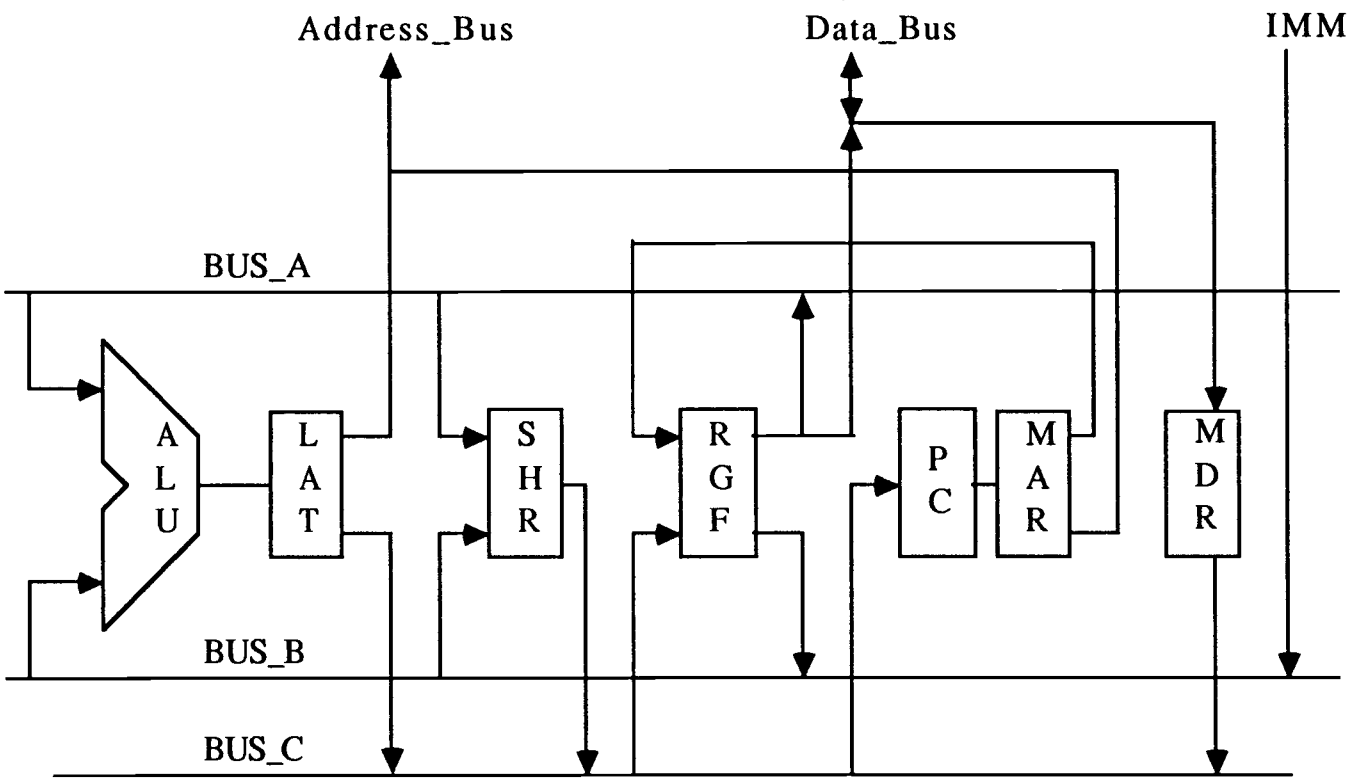


Fig. 4.5 Datapath Block Diagram

The actual datapath implemented in the Genesil Silicon Compiler is described in appendix A. It contains two parts: one part has a static ALU, a static barrel shifter, a register file, and some latches for storing immediate numbers from the instruction register; another part has the PC, (with MAR), a latch for the data bus, the Bus_C connection, and some elements for bootstrapping operations. The two datapath parts were netlisted together, so three buses, Bus_A, Bus_B and Bus_C, are actually contained in this datapath module.

The static ALU operates on data from Bus_A and Bus_B, and drives its output onto Bus_C, or directly to the Address Bus if the effective address is being calculated by the ALU. The static barrel shifter shifts the data from Bus_A or Bus_B depending on whether the left or right shift function is to be performed, and drives its output onto Bus_C. Notice that the output of the static element occurs on the same clock phase as the inputs. The Register File contain 16 16-bits registers driving both Bus_A, Bus_B, and the Data Bus. It receives input from Bus_C or the PC depending on what operation is being performed. The content of R0 is always 0. No other data can be stored in R0.

In the general case, the registers receive input data from Bus_C. In the case of the Load instruction, the data is loaded to the register file from Bus_C. Bus_C is driven from the data bus, which

receives data from the memory. This is illustrated in the view of the datapath contained in Appendix A.

In the case of the Store instruction, the data is placed on the Data Bus directly, so the MDR is not needed in the process.

In case of the Call instruction, the address of the instruction to be executed after the subroutine has completed is stored in the register file by a direct connection from the PC to the Register File. In the case of the Return operation, the PC value which points to the main program is read from the register file, through ALU to Bus_C, and is then loaded in the PC.

A four overlapped register window structure is implemented in the register file. Each window has four global registers, R0 to R3, two local registers, R5 and R6, one high parameter register, R7, and one low parameter register, R4. The high and low registers are used to pass parameters to and from subroutines. In this design, only the program counter can be passed, due to the small number of registers implemented. The overlapped register window structure is illustrated in figure 4.6. When an overflow occurs, four clock cycles are required for overflow handling. The Stack Pointer is sent to the Address Bus during the first cycle. Registers R5, R6, and R7 are sent to memory in the second, third, and fourth cycles, respectively. The same process applies for underflow, except that data are restored from the memory to the registers. The timing diagram for overflow and underflow is shown in figure 4.7.

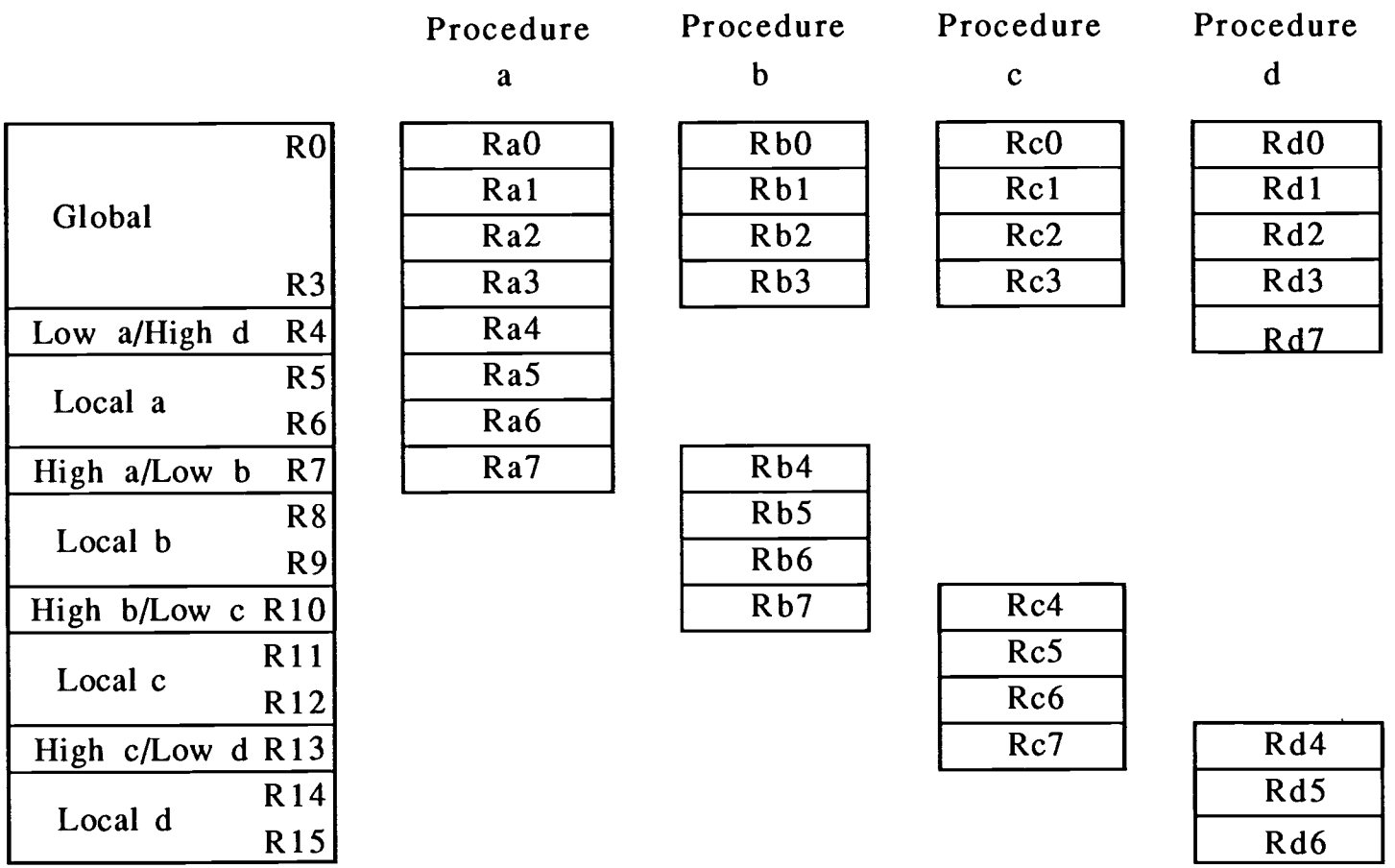


Fig. 4.6. Overlapped Register Window

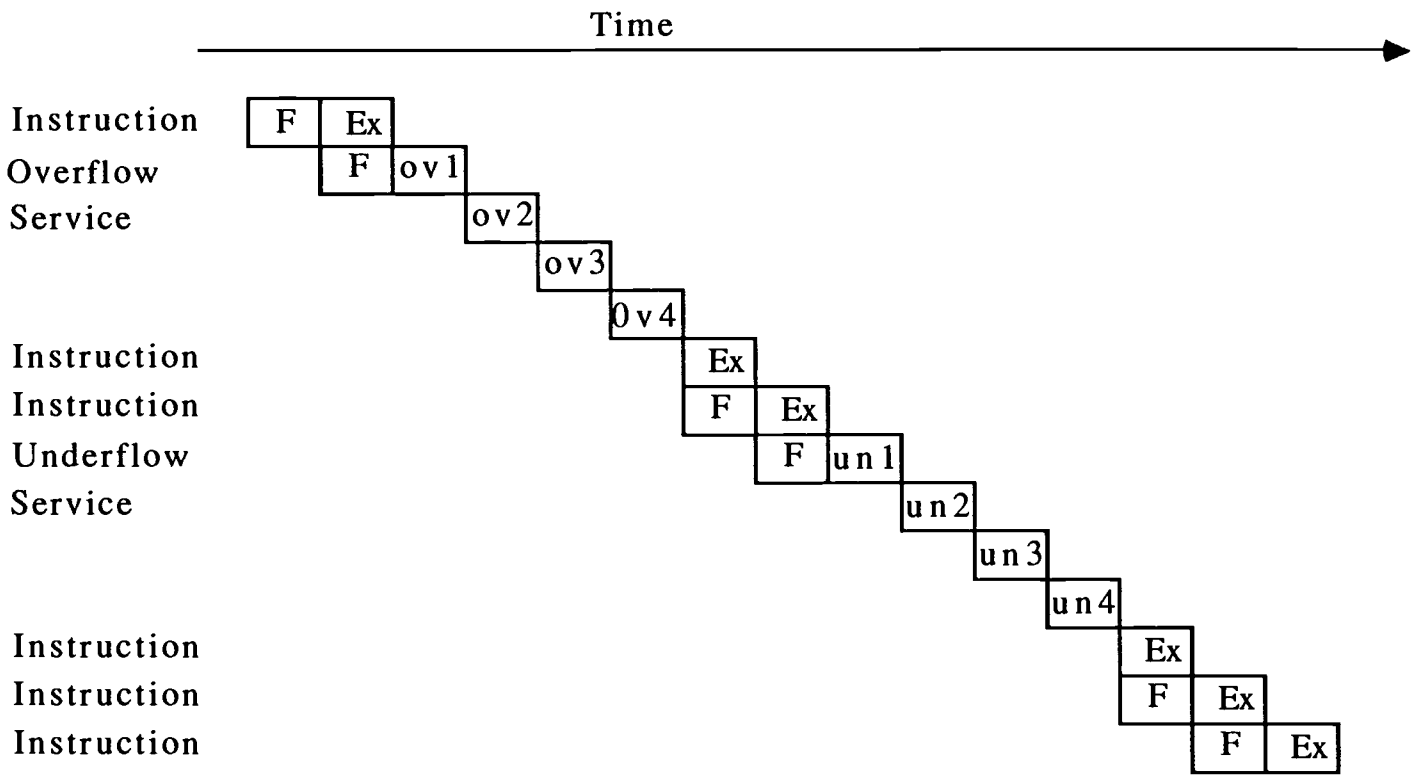


Fig. 4.7. Time For Overflow & Underflow

The MAR always is increased via the PC unless a branch is performed. In the case of a branch, the effective address is calculated by the ALU and is sent to the Address Bus directly. The effective branch address is loaded into the PC from Bus_C as well. This process saves transferring the branch address to the PC, and then on to the Address Bus.

4.6. Controller Implementation.

The controller in this project is comprised of six parts. They are the instruction register (IR), instruction decoder, finite state machine (FSM), flags, pointers (including the current window pointer (CWP), stack pointer (SP), and saving window pointer (SWP)), and control structures for the register window. A block diagram of the controller is shown in figure 4.8.

4.6.1. The Instruction Register.

A block diagram of the instruction register is shown in figure 4.9. As shown in figure 4.8, there are two possible sources for the IR: one from a latch which connected to the Data Bus; the other from a ROM whose contents are zero. Usually, instructions are fetched from the Data Bus through the latch. Whenever a branch instruction is being executed, the IR has to fetch a NOP, whose opcode is 0000, from the ROM to clean out the pipeline. In this case, a delayed branch is performed in machine.

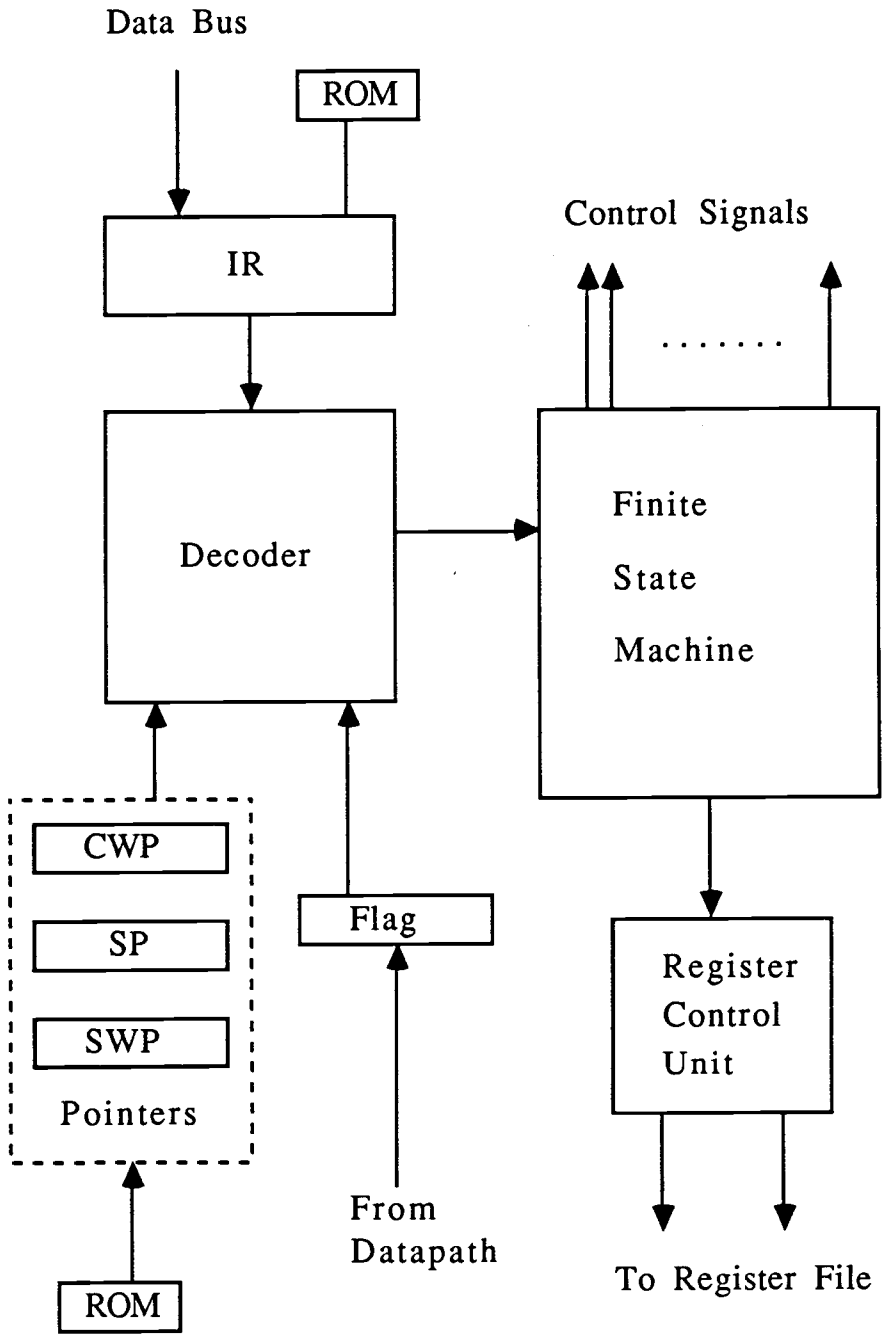


Fig. 4.8. Controller

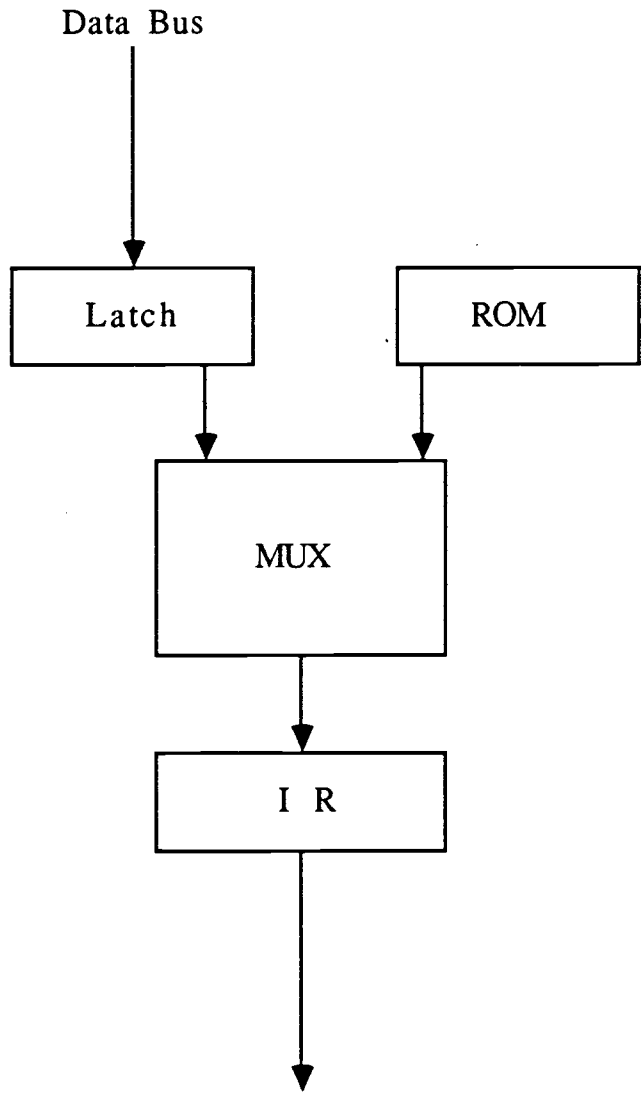


Fig. 4.9. Instruction Register

The latch between the Data Bus and the MUX is used when Load or Store instructions are executed. In this case, the Load or Store instruction must be maintained in the IR for two cycles. As discussed earlier, the next instruction is fetched during the first cycle, then data is loaded or stored to or from memory in the second cycle. Therefore, the next instruction should be stored in the latch for one cycle, then transferred to IR at a later time.

The actual instruction register is shown in appendix A It is implemented as a small datapath. The latch and instruction register are a gated latch element. This configuration insures that the instruction will stay in the register until a new instruction is loaded.

4.6.2. The Instruction Decoder and Finite State Machine.

As shown in figure 4.8, the instruction decoder gets the instruction from the IR, flag, CWP, SWP, decodes the opcode, jump condition and register address, and then sends this information to the finite state machine (FSM). The FSM produces the appropriate control signals for the rest of the system as a result. The actual decoder is implemented on silicon in a PLA. The PLA description file is given in appendix B.

The reasons for using a finite state machine for the controller are as follows: First, the FSM is fast; Secondly, although most instructions are single cycle instructions, the Load and Store instructions require two cycles for their execution. A finite state

machine makes the implementation of two cycle control signals easier than that which could be achieved via other control structures. The PLA description file for the FSM is given in appendix B.

4.6.3. Flags and Pointers.

As shown in figure 4.8, there are three pointers, CWP, SP, SWP, and a flag in the controller. There is no requirement that these structures must be part of controller, but it was convenient to do so.

The Flag Register gets its data from the ALU contained in the datapath. The carry bit and sign bit are used in deciding if the condition for a conditional jump has been met or not. The silicon compiler implementation of flag register is shown in appendix A. It is simply a gated latch.

The Current Window Pointer and Saving Window Pointer are set to 00 when the system is booted. On each Call instruction, the CWP will be incremented; on each Return instruction, the CWP will be decremented. The same thing happens to the SWP. Each time overflow occurs, SWP will be incremented and each time underflow occurs, SWP will be decremented.The actual implementation of CWP and SWP on the silicon compiler is provided in appendix A. Their structures are similar. The adder/subtracter blocks perform the increment and decrement functions.

4.7. Memory.

The memory in this project consists of three parts: the memory address decoder (memcontr), a 128 words half-cycle RAM, and a ROM which holds the simulation program. The memory is shown in figure 4.10.

The actual system memory map is shown in figure 4.11. The half-cycle RAM and ROM make single cycle instruction fetch and memory load/store instructions possible. The ROM holds a simulation program in which simulations of all instructions and special situations like overflow and underflow reside. The description of the Genesil RAM and ROM structures is provided in appendix B.

The memory address decoder takes the data from the Address Bus, determines if the address is in ROM, external memory, or in RAM. It then sends read or write signals to the appropriate devices. This module is implemented in random logic. The diagram is shown in appendix A.

4.8. Netlisting, Floorplanning, and Simulation.

The whole RISC chip was implemented in the silicon compiler hierarchically. The history of implementation process is provided in figure 4.12.

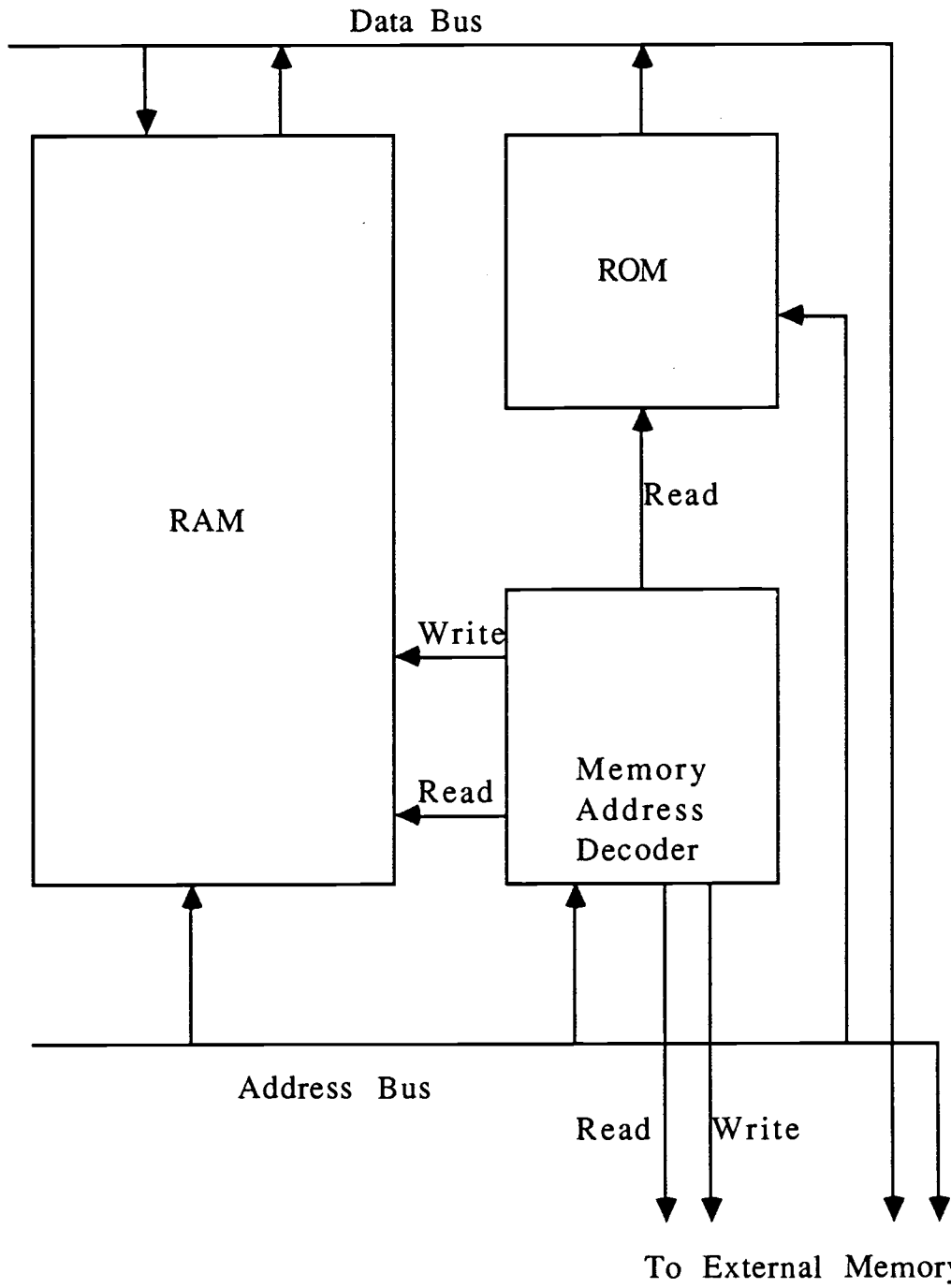


Fig. 4.10. Memory

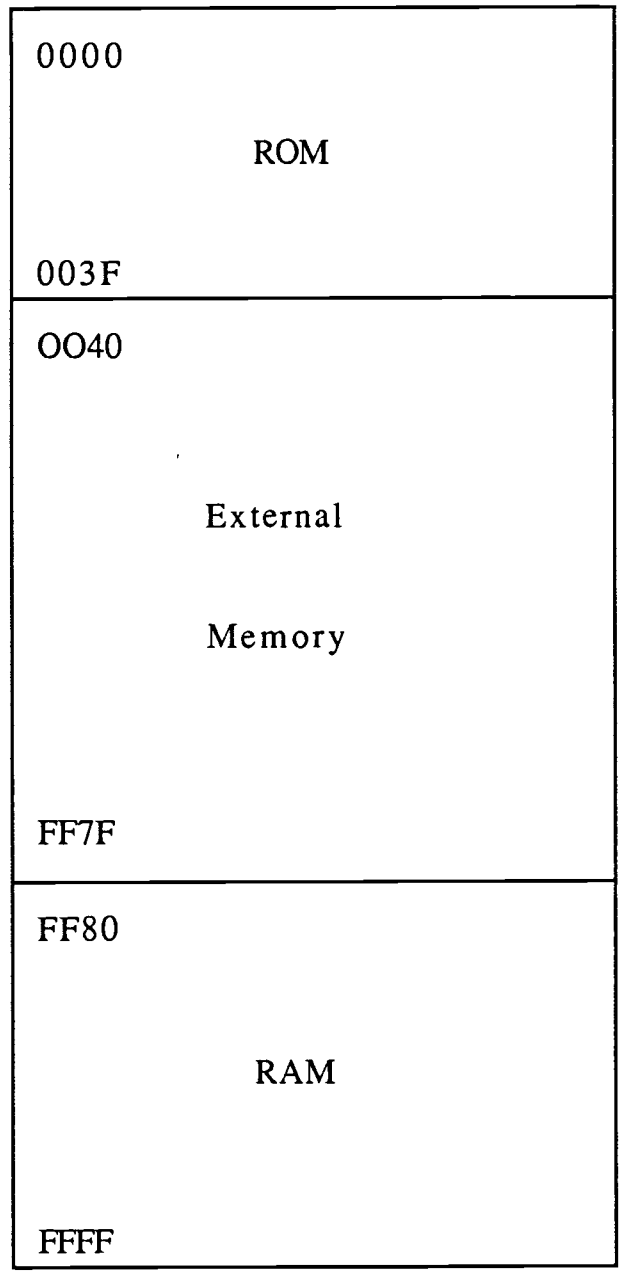


FIG. 4.11. Memory Map

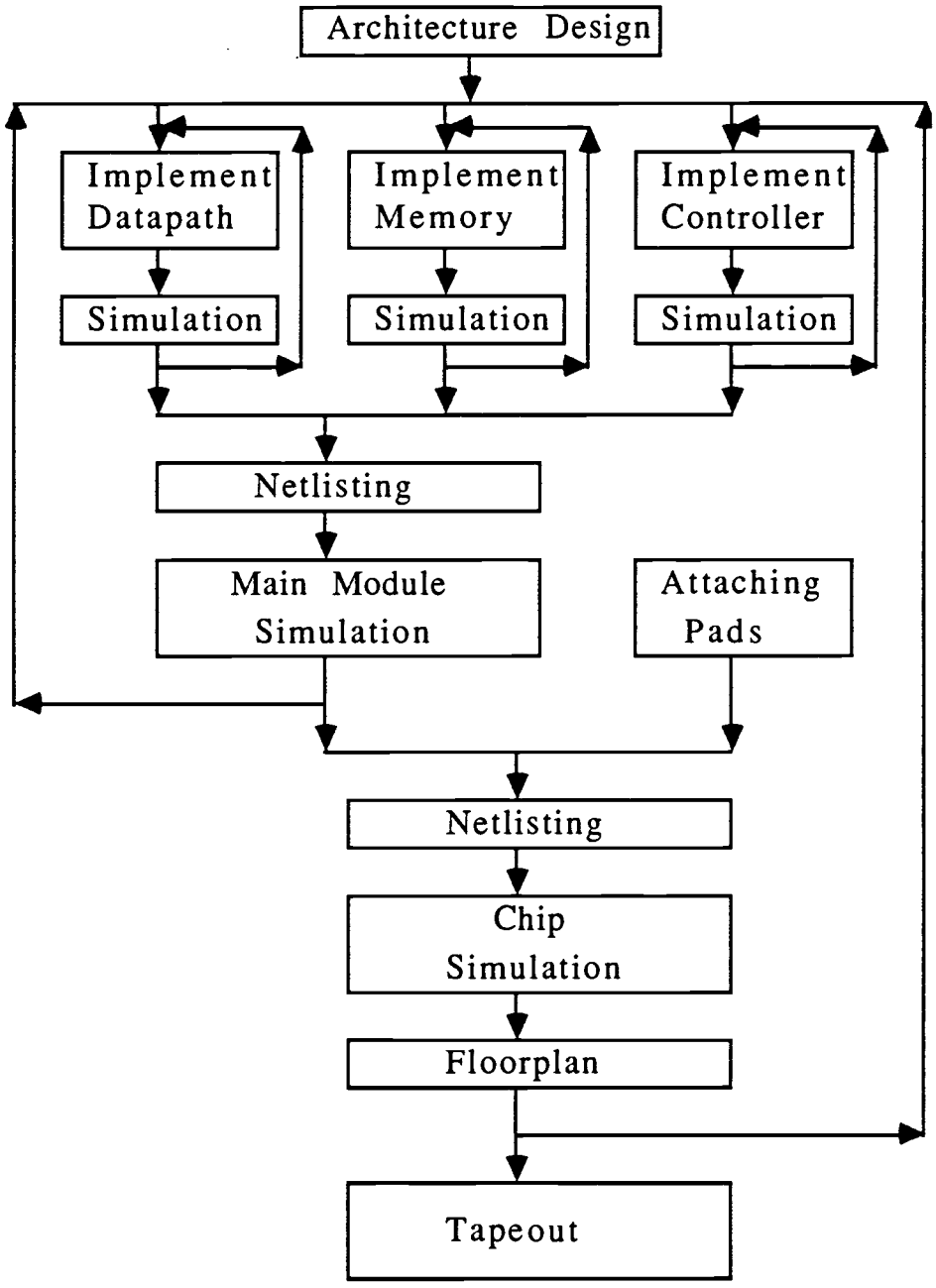


Fig. 4.12. History of Implementation Process

As previously discussed, the datapath module has two parts. They were implemented individually, and then netlisted together. All instructions were then simulated in the datapath module to make sure they worked properly.

The controller module has six sub-modules, the instruction register, decoder, finite state machine, pointers, flag, and register address decoder. All of them were implemented and simulated individually to assure they worked correctly. Finally, they were netlisted together, and all functions, including all instruction operations, overflow/underflow handling, booting, etc, were simulated in the controller module. All controller signals and register addresses for different windows were produced correctly at this stage.

In the memory module, the RAM, ROM and memory decoder were implemented and simulated individually. The simulation program for the whole chip was written into ROM at this stage of the design process. The memory was simulated after netlisting. The simulation verified correct timing and address mapping for the memory module.

After all these modules were implemented correctly and were proven to work correctly, they were netlisted together. A reset signal was applied to the system. The system then started to execute the simulation program contained in the ROM. The assembly program associated with this simulation is shown in

appendix C. The program is a simple search program. It exercises every instruction and every instruction which can be performed using the instruction set, e.g., add immediate, conditional jump, procedure call, and overflow/underflow conditions. After simulation verified that the system worked as expected, pads were added. All modules and pads were then netlisted to from a complete chip. The simulation was then performed at the chip level again.

Finally, the RISC chip floorplanning activity was carried out. Modules were arranged to minimize extraneous wiring and to create the smallest possible layout. External pin connections for chip were also defined during floorplanning. The floorplan for this chip is shown in figure 4.13. The chip pinout is shown in figure 4.14.

4.9. Chip Performance.

Timing analyze shows that RISC chip can operate at a maximum clock of 5.88 Mhz using 2-micron CMOS technology. That implies a 170ns maximum clock cycle. Since most RISCs instructions are executed in one clock cycle, the peak performance for this chip is 5.88 MIPS. In the worst case, when overflow or underflow occurs, it takes five cycles for one instruction. Therefore, at least 1 MIPS performance is achieved in this RISC under these adverse conditions. The Genesil timing analyze form is shown in appendix E.

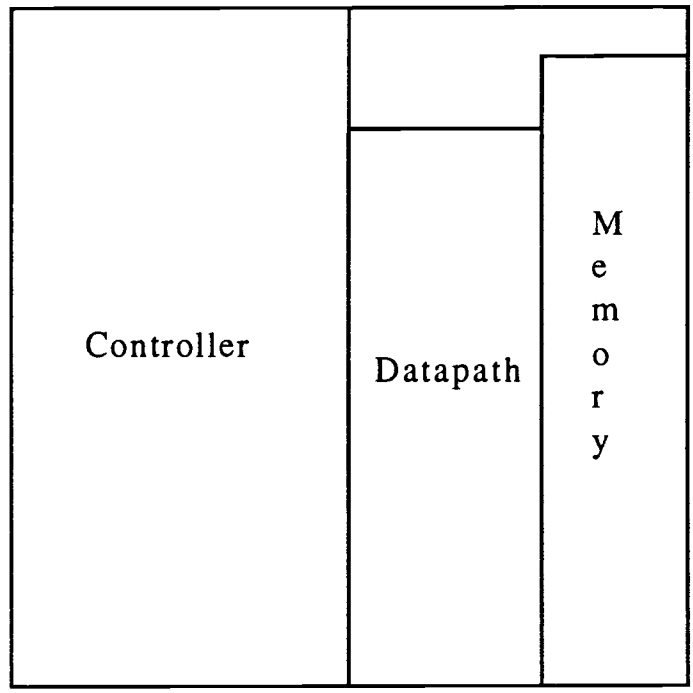


Fig. 4.13. Chip Floorplan

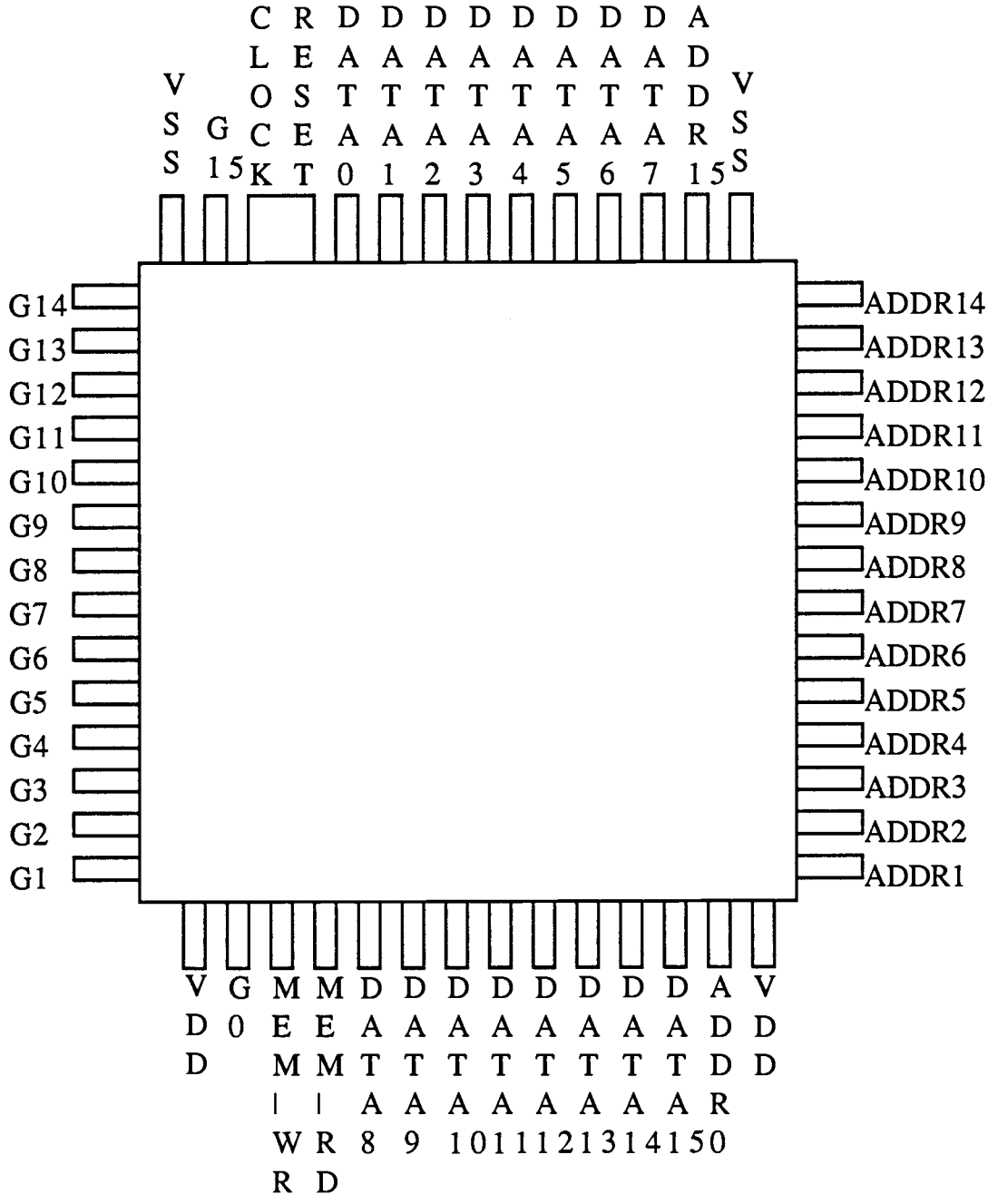


FIG. 4.14. Chip Pinout

5. CONCLUSIONS.

The RISC architecture described in this thesis is similar to the Berkeley RISC I machine. It has 14 basic instructions. Most of them are register-to-register, and can be executed in a single clock cycle. Like the RISC I, an overlapped register window structure is included. This single chip RISC was implemented and simulated using the Genesil Silicon Compiler. The chip is capable of operating at a clock rate of 5.88 Mhz. An instruction execution rate of somewhat less than 5 MIPS can be expected as a result. Further benchmark studies would be required in order to verify the performance over a range of applications. The chip size is 516.54 X 514.27 mils.

Many things were learned in the process of completing this thesis. Among these are: A knowledge of why RISC architectures are valuable commercially, an understanding of register window structure design, the difficulties associated with controller design, the strengths and weaknesses of CAD tools, the need to perform several iterations during the design process. These items are described in more detail in the following paragraphs.

It is now obvious why RISC machines can achieve much greater performance at a given cost than CISC designs. The reasons are fundamentally related to chip architectural complexity, on-chip

communication issues, and the statistical properties of the instruction set mix in typical application programs.

The RISC machine design in this thesis has four overlapped register windows. In this case, each window contains only four registers. The number of global registers is also four. Therefore a total of eight registers are available for servicing each procedure. The small register file and small window size make this RISC machine somewhat impractical. This is acceptable, given the exploratory nature of the thesis. Studies have shown that with eight register windows, overflow will occur on less than one percent of the calls [1]. With four register overflow will occur at a much higher rate. This will severely limit performance. It would have been difficult to implement a larger register file in this 16 bit machine. This is due to the fact that the limits exist on the size of the register file address fields in the instruction. For a program with deep procedure nesting, this machine will take more time to perform overflow/underflow handling. It is important to note that even with these minor limitations, the overlapped register window structure was implemented correctly and is fully functional.

Another result from the Berkeley RISC I project is that controllers for RISC machines are much simpler than CISC, and use much less chip area. Even so, they are complicated and time consuming to design. The controller in this project is somewhat larger than that in the RISC I. There are three reasons for this:

1). The controller defined in this project includes many modules which are not necessarily part of a standard controller. These modules include the pointers, flag, and register address decoder. They consume a significant amount of chip area. The pointers, for example, account for almost 40% of the controller area.

2). The CWP, SWP in the pointer module and register address decoder are actually part of the overlapped register window structure. In the Genesil Silicon Compiler, only a simple register file can be implemented inside a parallel datapath module. Although the CWP, SWP, and register address decoder are integral parts of the overlapped register window structure, these modules were included in the controller rather than in the datapath.

3). The VLSI CAD tools used in RISC I project were different. Pure custom design tools were used rather than a silicon compiler. The Genesil Silicon Compiler can not use silicon as efficiently as these special purpose VLSI CAD tools. Of course, the design performed here was done much more quickly and with far fewer people.

The Genesil Silicon Compiler used in this project is a sophisticated VLSI CAD tool. With this VLSI CAD tool, the system designer can design special purpose VLSI chips quickly. Several design iterations were performed using this set of tools. At each iteration, improvements were made in the overall design. This

exploratory design activity helps reduce the overall design cycle by reducing redesign costs, and results in a superior final system architecture. Due to the correctness-by-construction capabilities of the silicon compiler, many of the sources of human error in the design process are eliminated.

There were some restrictions on the possible microarchitectures used to implement the RISC chip due to the Genesil Silicon Compiler as well. Because of its limited library of functional blocks only two buses can be defined in a single parallel datapath. If a parallel datapath with more than two buses must be designed, it must be constructed using two separate parallel datapath modules. The modules must then be connect together through netlisting. This requires additional design effort and chip area. Extensions to the library in the future would prove helpful.

6. BIBLIOGRAPHY

[1] Patterson, D., and Sequin, C. "A VLSI RISC." Computer, September, 1982.

[2] Brooks, F.P.,"The Mythical Man Month", Addison Wesley Publishers, 1970

[3] Mead,C.A., and Conway, L., "Introduction to VLSI Systems", Addison Wesley Publishers, 1980.

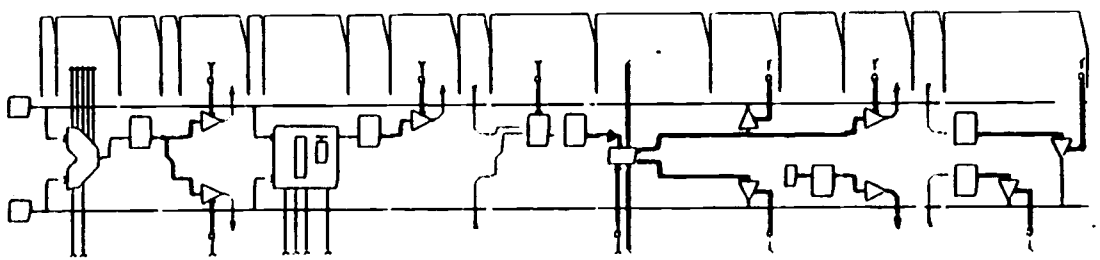
[4] Colwell, R.; Hitchcock, C.; Jensen, E.; Brinkley-Sprunt, H.; and Kollar, C. "Computers, Complexity, and Controversy." Computer, September, 1985.

[5] Wallich, P. "Toward Simpler, Faster computers." IEEE Spectrum, August, 1985.

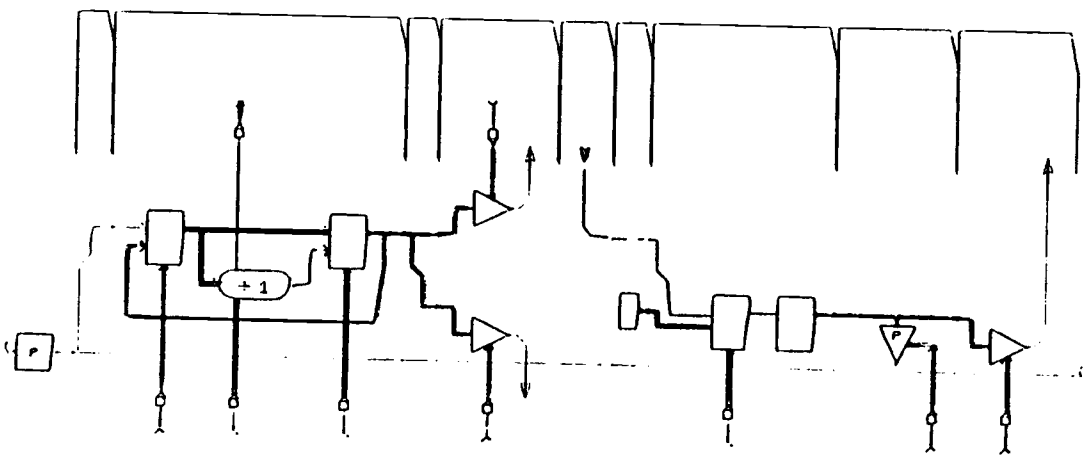
7. APPENDICES

APPENDIX A

Views of Silicon Compiler Functional Blocks

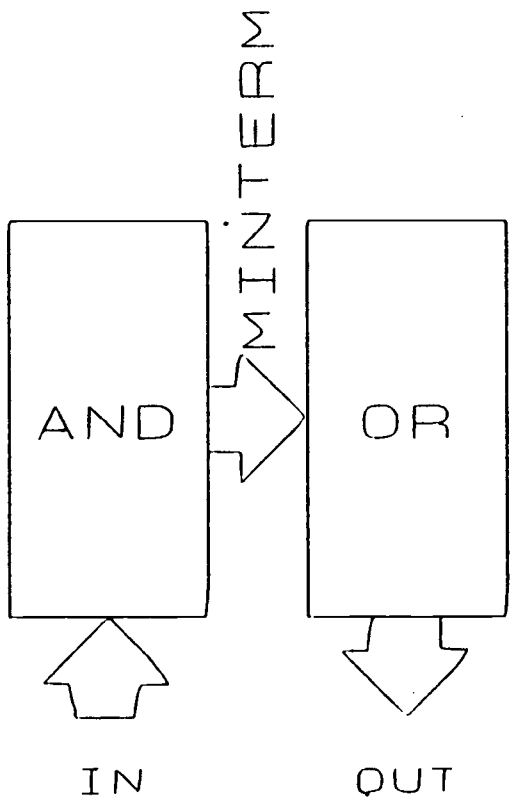


Silicon Compiler Systems	Object: dp	User: zhang	Date. May 6 89 3: 09

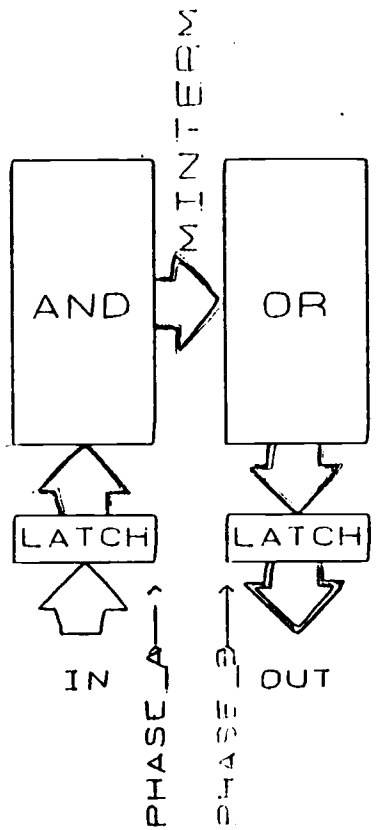


Silicon Compiler Systems	Object:	User:	Date:
	DCmar	zhang	May 6 89 3: 12

Silicon Compiler Systems	Object: IR	User: zhang	Date: May 6 89 3:14

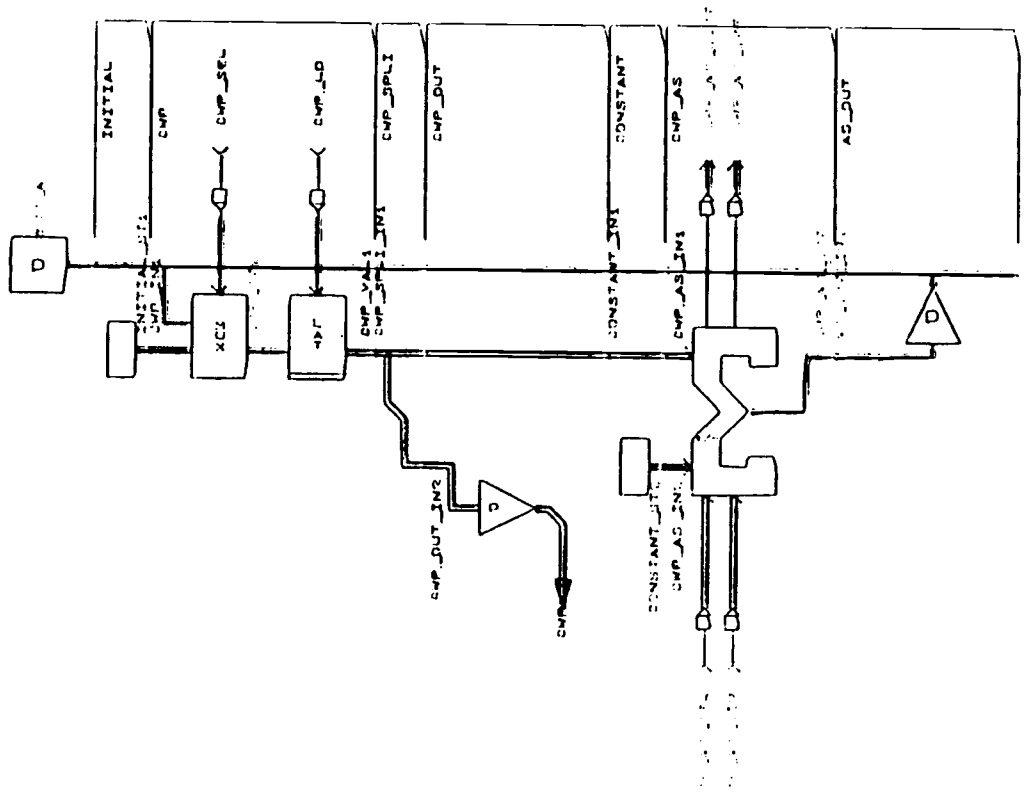


Silicon Compiler Systems	Object: decode	User: zhang	Date: May 6 89 3:15

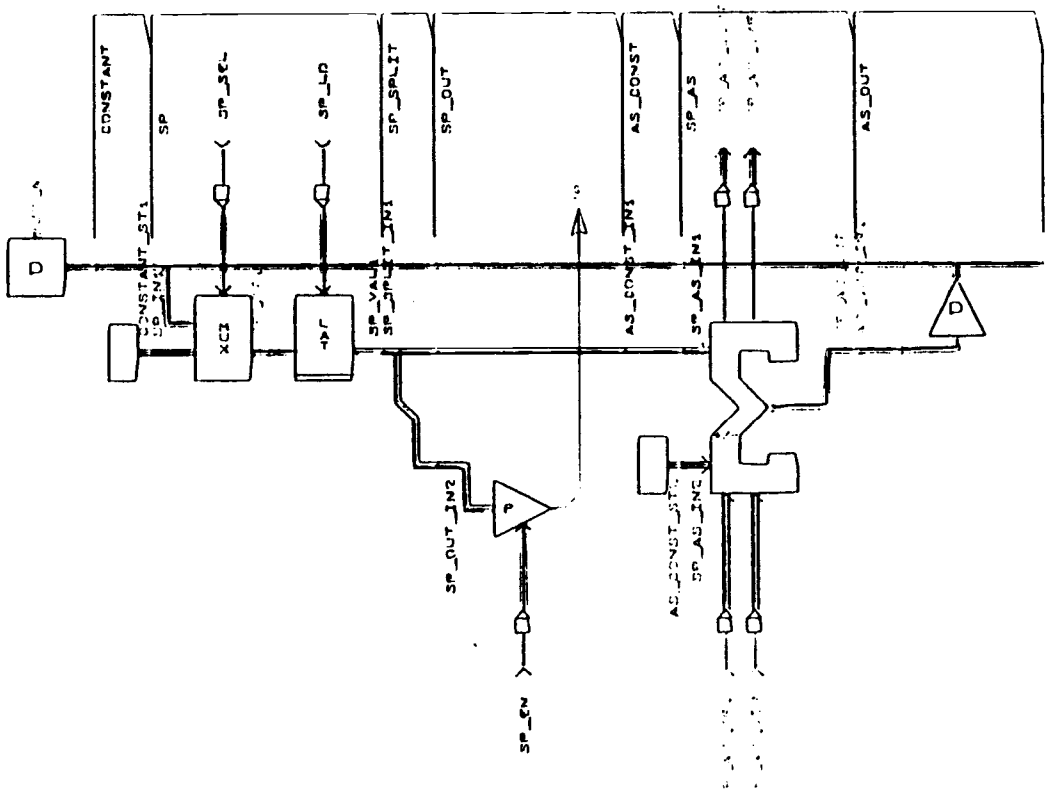


Silicon Compiler Systems	Object: finite	User: zhang	Date: May 6 89 3.17

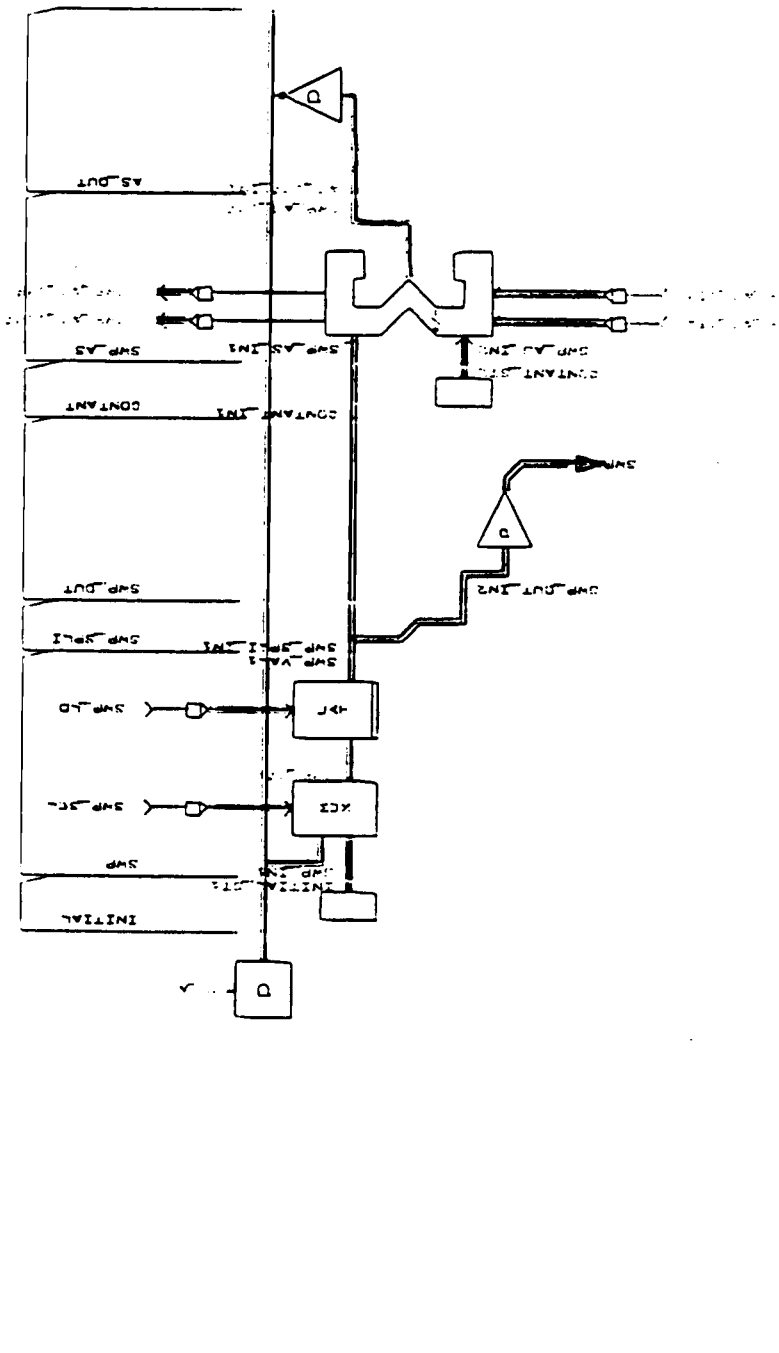
Silicon Compiler Systems	Object: flag	User: zhang	Date: May 6 89 3:18

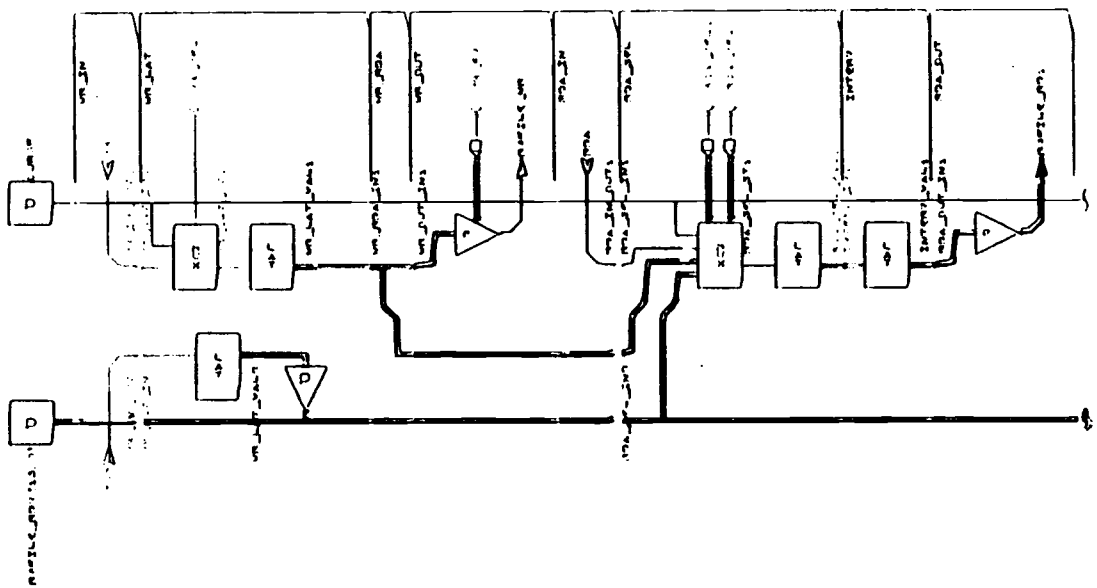


Silicon Compiler Systems	Object: cwp	User: zhang	Date: May 6 89 3:20



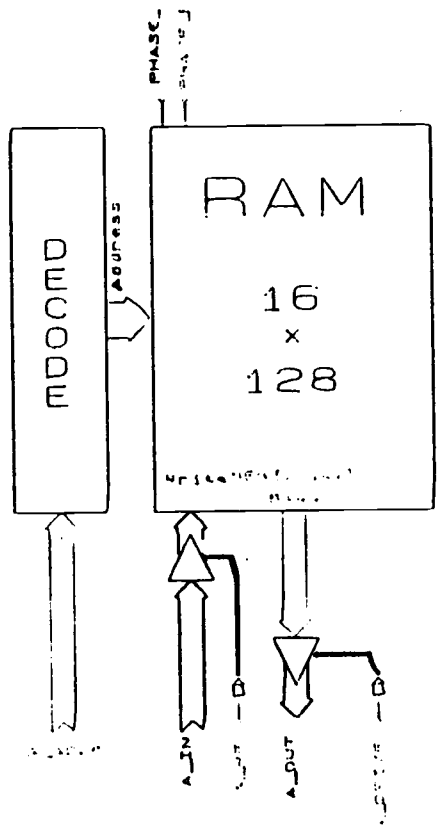
Silicon Compiler Systems	Object: sp	User: zhang	Date: May 6 89 3:21


Silicon Compiler Systems
Object:
swp
User:
zhang
Date:
May 6 89 3:23

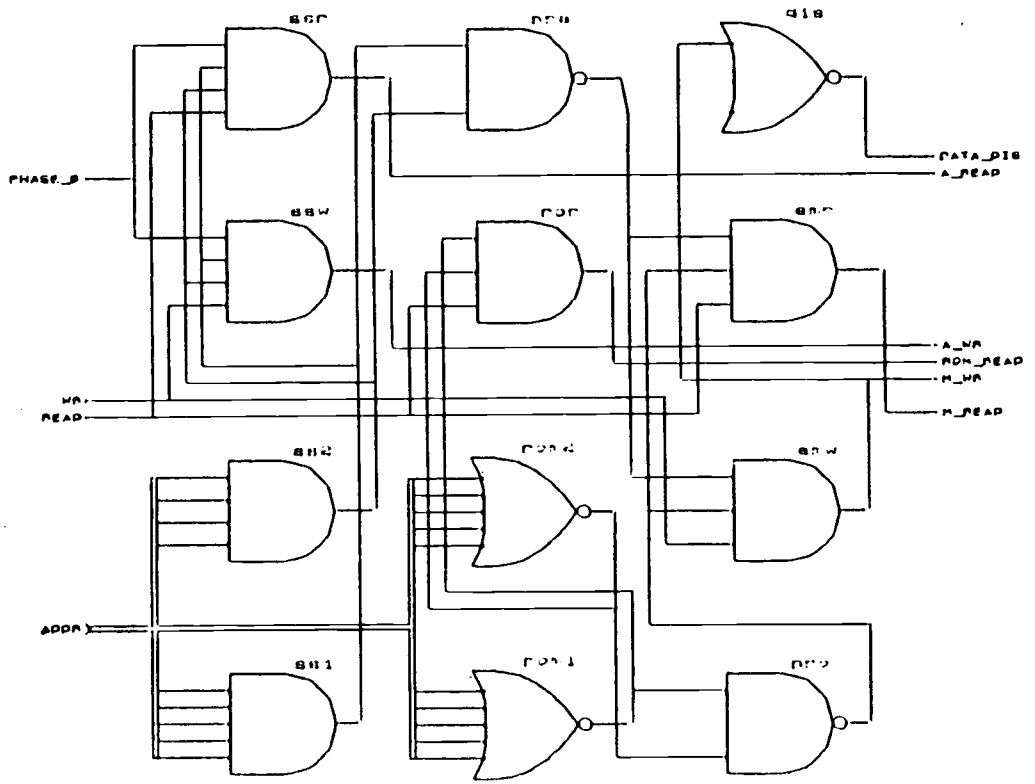


Silicon Compiler Systems	Object: rgfcontr	User: zhang	Date: May 6 89 3:26

Silicon Compiler Systems	Object: rom	User: zhang	Date: May 6 89 3:29



Silicon Compiler Systems	Object: ram	User. zhang	Date. May 6 89 3. 30



Silicon Compiler Systems	Object: memcontr	User: zhang	Date: May 6 89 3:32

APPENDIX B

Description of decoder, FSM, and ROM

Decoder

```
PLA_SOURCE
INPUTS OP[3:0],SCC,DEST[2:0],SRC1[2:0],IMM,SRC2[2:0],CWP[2:0],
       SWP[3:0],RT,FLAG[4],FALG[0];
OUTPUTS O1,O2,O3,O4,O5,
       WR[15:0],RDA[15:0],RDB[15:0],
       OURGF[15:0];

STATE
   NAME = IN;
   SIGNALS = OP[3:0],IMM,RT,CWP[2:1],SWP[3:2],SCC,DEST[0],
           FLAG[4],FALG[0];
   VALUE = iboot, .....1........;
   VALUE = inop,  0000.0........
                 +0101.0....10.0
                 +0101.0....110.;
   VALUE = iadd,  000100........;
   VALUE = iaddi, 000110........;
   VALUE = isub,  001000........;
   VALUE = isubi, 001010........;
   VALUE = iand,  001100........;
   VALUE = iandi, 001110........;
   VALUE = inot,  0100.0........;
   VALUE = ijump, 010100....0...
                 +010100....10.1
                 +010100....111.;
   VALUE = ijmpi, 010110....0...
                 +010110....10.1
                 +010110....111.;
   VALUE = icall, 011000.1.1....
                 +0110000.00....
                 +011000100.....
                 +011000001.....
                 +0110001.10....;
   VALUE = icalli,011010.1.1....
                 +0110100.00....
                 +011010100.....
                 +011010001.....
                 +0110101.10....;
   VALUE = irtn,  0111.00.1.....
                 +0111.01.0.....
                 +0111.000.1....
                 +0111.0.100....
                 +0111.0.110....
                 +0111.010.1....;
   VALUE = ishl,  1000.0........;
   VALUE = ishr,  1001.0........;
   VALUE = iload, 101000........;
   VALUE = iloadi,101010........;
   VALUE = istore,101100........;
   VALUE = istori,101110........;
   VALUE = ildh,  1100.0........;
   VALUE = iover, 0110.00001....
                 +0110.00110....
                 +0110.01011....
                 +0110.01100....;
```

```
   VALUE = iunder,0111.00000....
                  +0111.00101....
                  +0111.01010....
                  +0111.01111....;
ENDSTATE

STATE
   NAME = OUT;
   SIGNALS = O1,O2,O3,O4,O5;
   VALUE = oboot,   00000;
   VALUE = onop,    00001;
   VALUE = oadd,    00010;
   VALUE = oaddi,   00011;
   VALUE = osub,    00100;
   VALUE = osubi,   00101;
   VALUE = oand,    00110;
   VALUE = oandi,   00111;
   VALUE = onot,    01000;
   VALUE = ojump,   01001;
   VALUE = ojumpi,  01010;
   VALUE = ocall,   01011;
   VALUE = ocalli,  01100;
   VALUE = ortn,    01101;
   VALUE = oshl,    01110;
   VALUE = oshr,    01111;
   VALUE = oload,   10000;
   VALUE = oloadi,  10001;
   VALUE = ostore,  10010;
   VALUE = ostorei, 10011;
   VALUE = oldh,    10100;
   VALUE = ooverfl, 10101;
   VALUE = oundefl, 10110;
ENDSTATE

STATE
   NAME = DECDEST;
   SIGNALS = DEST[2:0],CWP[2:0],RT;
   VALUE = WW,  ......1;
   VALUE = W0,  000...0;
   VALUE = W1,  001...0;
   VALUE = W2,  010...0;
   VALUE = W3,  011...0;
   VALUE = W4,  1000000+1111100;
   VALUE = W5,  1010000;
   VALUE = W6,  1100000;
   VALUE = W7,  1110000+1000100;
   VALUE = W8,  1010100;
   VALUE = W9,  1100100;
   VALUE = W10, 1110100+1001000;
   VALUE = W11, 1011000;
   VALUE = W12, 1101000;
   VALUE = W13, 1111000+1001100;
   VALUE = W14, 1011100;
   VALUE = W15, 1101100;
ENDSTATE
```

```
STATE
   NAME = DECSRC1;
   SIGNALS = SRC1[2:0],CWP[2:1];
   VALUE = RA0,  000..;
   VALUE = RA1,  001..;
   VALUE = RA2,  010..;
   VALUE = RA3,  011..;
   VALUE = RA4,  10000+11111;
   VALUE = RA5,  10100;
   VALUE = RA6,  11000;
   VALUE = RA7,  11100+10001;
   VALUE = RA8,  10101;
   VALUE = RA9,  11001;
   VALUE = RA10, 11101+10010;
   VALUE = RA11, 10110;
   VALUE = RA12, 11010;
   VALUE = RA13, 11110+10011;
   VALUE = RA14, 10111;
   VALUE = RA15, 11011;
ENDSTATE

STATE
   NAME = DECSRC2;
   SIGNALS = SRC2[2:0],CWP[2:1];
   VALUE = RB0,  000..;
   VALUE = RB1,  001..;
   VALUE = RB2,  010..;
   VALUE = RB3,  011..;
   VALUE = RB4,  10000+11111;
   VALUE = RB5,  10100;
   VALUE = RB6,  11000;
   VALUE = RB7,  11100+10001;
   VALUE = RB8,  10101;
   VALUE = RB9,  11001;
   VALUE = RB10, 11101+10010;
   VALUE = RB11, 10110;
   VALUE = RB12, 11010;
   VALUE = RB13, 11110+10011;
   VALUE = RB14, 10111;
   VALUE = RB15, 11011;
ENDSTATE

STATE
   NAME=OVUNRGF;
   SIGNALS=SWP[3:0];
   VALUE = S0, 0000;
   VALUE = S1, 0001;
   VALUE = S2, 0010;
   VALUE = S3, 0100;
   VALUE = S4, 0101;
   VALUE = S5, 0110;
   VALUE = S6, 1000;
   VALUE = S7, 1001;
   VALUE = S8, 1010;
```

```
   VALUE = S9, 1100;
   VALUE = S10,1101;
   VALUE = S11,1110;
ENDSTATE

STATE
   NAME = RFWR;
   SIGNALS = WR[15:0];
   VALUE = RR,  1111111111111111;
   VALUE = R0,  1111111111111110;
   VALUE = R1,  1111111111111101;
   VALUE = R2,  1111111111111011;
   VALUE = R3,  1111111111110111;
   VALUE = R4,  1111111111101111;
   VALUE = R5,  1111111111011111;
   VALUE = R6,  1111111110111111;
   VALUE = R7,  1111111101111111;
   VALUE = R8,  1111111011111111;
   VALUE = R9,  1111110111111111;
   VALUE = R10, 1111101111111111;
   VALUE = R11, 1111011111111111;
   VALUE = R12, 1110111111111111;
   VALUE = R13, 1101111111111111;
   VALUE = R14, 1011111111111111;
   VALUE = R15, 0111111111111111;
ENDSTATE

STATE
   NAME = RFS1;
   SIGNALS = RDA[15:0];
   VALUE = A0,  1111111111111110;
   VALUE = A1,  1111111111111101;
   VALUE = A2,  1111111111111011;
   VALUE = A3,  1111111111110111;
   VALUE = A4,  1111111111101111;
   VALUE = A5,  1111111111011111;
   VALUE = A6,  1111111110111111;
   VALUE = A7,  1111111101111111;
   VALUE = A8,  1111111011111111;
   VALUE = A9,  1111110111111111;
   VALUE = A10, 1111101111111111;
   VALUE = A11, 1111011111111111;
   VALUE = A12, 1110111111111111;
   VALUE = A13, 1101111111111111;
   VALUE = A14, 1011111111111111;
   VALUE = A15, 0111111111111111;
ENDSTATE

STATE
   NAME = RFS2;
   SIGNALS = RDB[15:0];
   VALUE = B0,  1111111111111110;
   VALUE = B1,  1111111111111101;
   VALUE = B2,  1111111111111011;
   VALUE = B3,  1111111111110111;
```

```
   VALUE = B4,  1111111111101111;
   VALUE = B5,  1111111111011111;
   VALUE = B6,  1111111110111111;
   VALUE = B7,  1111111101111111;
   VALUE = B8,  1111111011111111;
   VALUE = B9,  1111110111111111;
   VALUE = B10, 1111101111111111;
   VALUE = B11, 1111011111111111;
   VALUE = B12, 1110111111111111;
   VALUE = B13, 1101111111111111;
   VALUE = B14, 1011111111111111;
   VALUE = B15, 0111111111111111;
ENDSTATE

STATE
   NAME = OVUN;
   SIGNALS = OURGF[15:0];
   VALUE = OV0, 1111111111011111;
   VALUE = OV1, 1111111110111111;
   VALUE = OV2, 1111111101111111;
   VALUE = OV3, 1111111011111111;
   VALUE = OV4, 1111110111111111;
   VALUE = OV5, 1111101111111111;
   VALUE = OV6, 1111011111111111;
   VALUE = OV7, 1110111111111111;
   VALUE = OV8, 1101111111111111;
   VALUE = OV9, 1011111111111111;
   VALUE = OV10,0111111111111111;
   VALUE = OV11,1111111111101111;
ENDSTATE

EQUATIONS
   SWITCH IN
      CASE iboot   : oboot
      CASE inop    : onop
      CASE iadd    : oadd
      CASE iaddi   : oaddi
      CASE isub    : osub
      CASE isubi   : osubi
      CASE iand    : oand
      CASE iandi   : oandi
      CASE inot    : onot
      CASE ijump   : ojump
      CASE ijmpi   : ojumpi
      CASE icall   : ocall
      CASE icalli  : ocalli
      CASE irtn    : ortn
      CASE ishl    : oshl
      CASE ishr    : oshr
      CASE iload   : oload
      CASE iloadi  : oloadi
      CASE istore  : ostore
      CASE istori  : ostorei
      CASE ildh    : oldh
      CASE iover   : ooverfl
```

```
   CASE iunder  : oundef1
ENDSWITCH
SWITCH DECDEST
   CASE WW: RR
   CASE W0: R0
   CASE W1: R1
   CASE W2: R2
   CASE W3: R3
   CASE W4: R4
   CASE W5: R5
   CASE W6: R6
   CASE W7: R7
   CASE W8: R8
   CASE W9: R9
   CASE W10:R10
   CASE W11:R11
   CASE W12:R12
   CASE W13:R13
   CASE W14:R14
   CASE W15:R15
ENDSWITCH
SWITCH DECSRC1
   CASE RA0: A0
   CASE RA1: A1
   CASE RA2: A2
   CASE RA3: A3
   CASE RA4: A4
   CASE RA5: A5
   CASE RA6: A6
   CASE RA7: A7
   CASE RA8: A8
   CASE RA9: A9
   CASE RA10:A10
   CASE RA11:A11
   CASE RA12:A12
   CASE RA13:A13
   CASE RA14:A14
   CASE RA15:A15
ENDSWITCH
SWITCH DECSRC2
   CASE RB0: B0
   CASE RB1: B1
   CASE RB2: B2
   CASE RB3: B3
   CASE RB4: B4
   CASE RB5: B5
   CASE RB6: B6
   CASE RB7: B7
   CASE RB8: B8
   CASE RB9: B9
   CASE RB10:B10
   CASE RB11:B11
   CASE RB12:B12
   CASE RB13:B13
   CASE RB14:B14
```

```
      CASE RB15:B15
   ENDSWITCH
   SWITCH OVUNRGF
      CASE S0: OV0
      CASE S1: OV1
      CASE S2: OV2
      CASE S3: OV3
      CASE S4: OV4
      CASE S5: OV5
      CASE S6: OV6
      CASE S7: OV7
      CASE S8: OV8
      CASE S9: OV9
      CASE S10:OV10
      CASE S11:OV11
   ENDSWITCH
END
```

Finite State Machine

```
PLA_SOURCE
INPUTS O1,O2,O3,O4,O5;
OUTPUTS OPCODE[6:0],CIN,ALU_BUSC_EN1,ALU_AB_EN2,SHIFTER_DIR,
      SHIFTER_SE[3:0],SHR_BUSC_EN1,RF_SE_SEL1,RD_A,
      RD_B,RF_DBUS_EN1,IMM_BUSB_DRB1,IMM_BUSB_DRB2,
      PC_LOAD,PC_CE,
      PC_RF_EN1,PC_RF_EN2,DB_IN_SEL2,DB_BUSC_DRB2,
      BOOT_ADD_EN2,IR_A_LD1,IR_B_SEL1,IR_B_LD1,WR_SEL,
      WR_EN,RDA_SEL1,RDA_SEL2,CWP_SEL,CWP_LD,
      CWP_AS_SEL,
      CWP_AS_CIN,SWP_SEL,SWP_LD,SWP_AS_SEL,SWP_AS_CIN,
      SP_SEL,SP_LD,SP_EN,SP_AS_SEL,SP_AS_CIN,FLAG_LD,
      A_WE,A_DRIVE,RGF0;
FEEDBACK F1,F2,F3,F4;

STATE
    NAME=IN1;
    SIGNALS=O1,O2,O3,O4,O5;
    VALUE=boti, 00000;
    VALUE=nopi, 00001;
    VALUE=addi, 00010;
    VALUE=adii, 00011;
    VALUE=subi, 00100;
    VALUE=sbii, 00101;
    VALUE=andi, 00110;
    VALUE=anii, 00111;
    VALUE=noti, 01000;
    VALUE=jmpi, 01001;
    VALUE=jpii, 01010;
    VALUE=cali, 01011;
    VALUE=caii, 01100;
    VALUE=rtni, 01101;
    VALUE=shli, 01110;
    VALUE=shri, 01111;
    VALUE=lodi, 10000;
    VALUE=ldii, 10001;
    VALUE=stri, 10010;
    VALUE=stii, 10011;
    VALUE=ldhi, 10100;
    VALUE=over, 10101;
    VALUE=unde, 10110;
ENDSTATE

STATE
   NAME=OUT;
   SIGNALS=OPCODE[6:0],CIN,ALU_BUSC_EN1,ALU_AB_EN2,SHIFTER_DIR,
         SHIFTER_SE[3:0],SHR_BUSC_EN1,RF_SE_SEL1,RD_A,
         RD_B,RF_DBUS_EN1,IMM_BUSB_DRB1,IMM_BUSB_DRB2,
         PC_LOAD,PC_CE,
         PC_RF_EN1,PC_RF_EN2,DB_IN_SEL2,DB_BUSC_DRB2,
         BOOT_ADD_EN2,IR_A_LD1,IR_B_SEL1,IR_B_LD1,WR_SEL,
         WR_EN,RDA_SEL1,RDA_SEL2,CWP_SEL,CWP_LD,
         WP_AS_SEL,
         CWP_AS_CIN,SWP_SEL,SWP_LD,SWP_AS_SEL,SWP_AS_CIN,
         SP_SEL,SP_LD,SP_EN,SP_AS_SEL,SP_AS_CIN,FLAG_LD,
```

```
        A_WE,A_DRIVE,RGF0;
    VALUE=boto, 00000000000000000000001000111011111011001100110000000;
    VALUE=nopo, 00000000000000000000001100001011010000000000000000011;
    VALUE=addo, 10011110100000000110000110000101111000000000000001011;
    VALUE=adio, 10011110100000000100100110000101111000000000000001011;
    VALUE=subo, 10001111100000000110000110000101111000000000000001011;
    VALUE=sbio, 10001111100000000100100110000101111000000000000001011;
    VALUE=ando, 10101010100000000110000110000101111000000000000000011;
    VALUE=anio, 10101010100000000100100110000101111000000000000000011;
    VALUE=noto, 01000010100000000100000110000101111000000000000000011;
    VALUE=jmpo, 10011110110000000110001000000011101000000000000000001;
    VALUE=jpio, 10011110110000000100101000000011101000000000000000001;
    VALUE=calo, 10011110110000001110001001000011111001100000000000001;
    VALUE=caio, 10011110110000001100101001000011111001100000000000001;
    VALUE=rtno, 01000110110000000100001000000011101001010000000000001;
    VALUE=shlo, 00000000001111110100000110000101111000000000000000011;
    VALUE=shro, 00000000000000110010000110000101111000000000000000011;
    VALUE=ldlo, 10011110010000000110000000000100101000000000000000011;
    VALUE=lilo, 10011110010000000100100000000100101000000000000000011;
    VALUE=ld2o, 00000000000000000000001100100011110000000000000000011;
    VALUE=stlo, 10011110010000000110000000000100100000000000000000011;
    VALUE=silo, 10011110010000000100100000000100100000000000000000011;
    VALUE=st2o, 00000000000000000001000110000011010000000000000000101;
    VALUE=ldho, 01011000100000000000101100001011110000000000000000011;
    VALUE=ov1o, 00000000000000000000000000000000011000001100010000001;
    VALUE=ov2o, 00000000000000000001000000000000011000001100110000101;
    VALUE=ov3o, 00000000000000000001000000000000011000001110110000101;
    VALUE=ov4o, 00000000000000000001000000000000010000000010000010100;
    VALUE=un1o, 00000000000000000000000000000000010000010001111000101;
    VALUE=un2o, 00000000000000000000000000010000011000001010111100111;
    VALUE=un3o, 00000000000000000000000000010000011000001010111100111;
    VALUE=un4o, 00000000000000000000000000010000011000000000000000011;
ENDSTATE

STATE
    NAME=CURSTATE;
    SIGNALS=F1,F2,F3,F4;
    VALUE=singal, 0000;
    VALUE=ld2,    0001;
    VALUE=st2,    0010;
    VALUE=und2,   0011;
    VALUE=und3,   0100;
    VALUE=und4,   0101;
    VALUE=ove2,   0110;
    VALUE=ove3,   0111;
    VALUE=ove4,   1000;
ENDSTATE

EQUATIONS
    FSM CURSTATE:
      ON boti GOTO singal DRIVING boto
      STATE singal
          ON nopi DRIVING nopo
          ON addi DRIVING addo
          ON adii DRIVING adio
```

```
        ON subi DRIVING subo
        ON sbii DRIVING sbio
        ON andi DRIVING ando
        ON anii DRIVING anio
        ON noti DRIVING noto
        ON jmpi DRIVING jmpo
        ON jpii DRIVING jpio
        ON cali DRIVING calo
        ON caii DRIVING caio
        ON rtni DRIVING rtno
        ON shli DRIVING shlo
        ON shri DRIVING shro
        ON lodi GOTO ld2 DRIVING ld1o
        ON ldii GOTO ld2 DRIVING li1o
        ON stri GOTO st2 DRIVING st1o
        ON stii GOTO st2 DRIVING si1o
        ON ldhi DRIVING ldho
        ON over GOTO ove2 DRIVING ov1o
        ON unde GOTO und2 DRIVING un1o
    STATE ld2
        ALWAYS GOTO singal DRIVING ld2o
    STATE st2
        ALWAYS GOTO singal DRIVING st2o
      STATE und2
        ALWAYS GOTO und3 DRIVING un2o
    STATE und3
        ALWAYS GOTO und4 DRIVING un3o
    STATE und4
        ALWAYS GOTO singal DRIVING un4o
    STATE ove2
        ALWAYS GOTO ove3 DRIVING ov2o
    STATE ove3
        ALWAYS GOTO ove4 DRIVING ov3o
    STATE ove4
        ALWAYS GOTO singal DRIVING ov4o
  ENDFSM
END
```

ROM

```
PLA_SOURCE
INPUTS ADDR_BUS[5:0];
OUTPUTS ROM_OUT[15:0];

STATE
    NAME = in;
    SIGNALS = ADDR_BUS[5:0];
    VALUE = r00, 6x00;
    VALUE = r01, 6x01;
    VALUE = r02, 6x02;
    VALUE = r03, 6x03;
    VALUE = r04, 6x04;
    VALUE = r05, 6x05;
    VALUE = r06, 6x06;
    VALUE = r07, 6x07;
    VALUE = r08, 6x08;
    VALUE = r09, 6x09;
    VALUE = r0a, 6x0a;
    VALUE = r0b, 6x0b;
    VALUE = r0c, 6x0c;
    VALUE = r0d, 6x0d;
    VALUE = r0e, 6x0e;
    VALUE = r0f, 6x0f;
    VALUE = r10, 6x10;
    VALUE = r11, 6x11;
    VALUE = r12, 6x12;
    VALUE = r13, 6x13;
    VALUE = r14, 6x14;
    VALUE = r15, 6x15;
    VALUE = r16, 6x16;
    VALUE = r17, 6x17;
    VALUE = r18, 6x18;
    VALUE = r19, 6x19;
    VALUE = r1a, 6x1a;
    VALUE = r1b, 6x1b;
    VALUE = r1c, 6x1c;
    VALUE = r1d, 6x1d;
    VALUE = r1e, 6x1e;
    VALUE = r1f, 6x1f;
    VALUE = r20, 6x20;
    VALUE = r21, 6x21;
    VALUE = r22, 6x22;
    VALUE = r23, 6x23;
    VALUE = r24, 6x24;
    VALUE = r25, 6x25;
    VALUE = r26, 6x26;
    VALUE = r27, 6x27;
    VALUE = r28, 6x28;
    VALUE = r29, 6x29;
    VALUE = r2a, 6x2a;
    VALUE = r2b, 6x2b;
    VALUE = r2c, 6x2c;
    VALUE = r2d, 6x2d;
    VALUE = r2e, 6x2e;
    VALUE = r2f, 6x2f;
```

```
    VALUE = r30, 6x30;
    VALUE = r31, 6x31;
    VALUE = r32, 6x32;
    VALUE = r33, 6x33;
    VALUE = r34, 6x34;
    VALUE = r35, 6x35;
    VALUE = r36, 6x36;
    VALUE = r37, 6x37;
    VALUE = r38, 6x38;
ENDSTATE

STATE
    NAME = out;
    SIGNALS = ROM_OUT[15:0];
    VALUE = c00, 16xc501;
    VALUE = c01, 16x11b1;
    VALUE = c02, 16x12b3;
    VALUE = c03, 16x13b2;
    VALUE = c04, 16x15bc;
    VALUE = c05, 16x16a5;
    VALUE = c06, 16x46c0;
    VALUE = c07, 16x26d7;
    VALUE = c08, 16x86c0;
    VALUE = c09, 16x2022;
    VALUE = c0a, 16x58a0;
    VALUE = c0b, 16x2023;
    VALUE = c0c, 16x5ab2;
    VALUE = c0d, 16x2043;
    VALUE = c0e, 16x5eb4;
    VALUE = c0f, 16x67a3;
    VALUE = c10, 16xb1d1;
    VALUE = c11, 16xb2d2;
    VALUE = c12, 16xb3d3;
    VALUE = c13, 16xb5d4;
    VALUE = c14, 16xa6d4;
    VALUE = c15, 16x9606;
    VALUE = c16, 16x5000;
    VALUE = c17, 16x0000;
    VALUE = c18, 16x0000;
    VALUE = c19, 16x0000;
    VALUE = c1a, 16x0000;
    VALUE = c1b, 16x0000;
    VALUE = c1c, 16x67b8;
    VALUE = c1d, 16x501b;
    VALUE = c1e, 16x67bc;
    VALUE = c1f, 16x501d;
    VALUE = c20, 16xb2c0;
    VALUE = c21, 16x1260;
    VALUE = c22, 16xa3c0;
    VALUE = c23, 16x501f;
    VALUE = c24, 16x1520;
    VALUE = c25, 16x1140;
    VALUE = c26, 16x12a0;
    VALUE = c27, 16x7080;
    VALUE = c28, 16x1620;
```

```
    VALUE = c29, 16x1160;
    VALUE = c2a, 16x13c0;
    VALUE = c2b, 16x7080;
    VALUE = c2c, 16x0000;
    VALUE = c2d, 16xc503;
    VALUE = c2e, 16x67a0;
    VALUE = c2f, 16x7080;
    VALUE = c30, 16xc503;
    VALUE = c31, 16x67b3;
    VALUE = c32, 16x7080;
    VALUE = c33, 16xc503;
    VALUE = c34, 16x67b6;
    VALUE = c35, 16x7080;
    VALUE = c36, 16x35a1;
    VALUE = c37, 16x36b0;
    VALUE = c38, 16x7080;
ENDSTATE

EQUATIONS
    SWITCH in
       CASE r00: c00
       CASE r01: c01
       CASE r02: c02
       CASE r03: c03
       CASE r04: c04
       CASE r05: c05
       CASE r06: c06
       CASE r07: c07
       CASE r08: c08
       CASE r09: c09
       CASE r0a: c0a
       CASE r0b: c0b
       CASE r0c: c0c
       CASE r0d: c0d
       CASE r0e: c0e
       CASE r0f: c0f
       CASE r10: c10
       CASE r11: c11
       CASE r12: c12
       CASE r13: c13
       CASE r14: c14
       CASE r15: c15
       CASE r16: c16
       CASE r17: c17
       CASE r18: c18
       CASE r19: c19
       CASE r1a: c1a
       CASE r1b: c1b
       CASE r1c: c1c
       CASE r1d: c1d
       CASE r1e: c1e
       CASE r1f: c1f
       CASE r20: c20
       CASE r21: c21
       CASE r22: c22
```

```
        CASE r23: c23
        CASE r24: c24
        CASE r25: c25
        CASE r26: c26
        CASE r27: c27
        CASE r28: c28
        CASE r29: c29
        CASE r2a: c2a
        CASE r2b: c2b
        CASE r2c: c2c
        CASE r2d: c2d
        CASE r2e: c2e
        CASE r2f: c2f
        CASE r30: c30
        CASE r31: c31
        CASE r32: c32
        CASE r33: c33
        CASE r34: c34
        CASE r35: c35
        CASE r36: c36
        CASE r37: c37
        CASE r38: c38
    ENDSWITCH
END
```

APPENDIX C

Test Program

Address	Instruction
0000	LDH R5,0010
0001	ADDi R1,R5,0001
0002	ADDi R2,R5,0003
0003	ADDi R3,R5,0002
0004	ADDi R5,R5,000C
0005	ADD R6,R5,R5
0006	NOT R6,R6
0007	SUBi R6,R6,0007
0008	SL R6,R6,R0
0009	SUB R0,R1,R2
000A	JUMP N,R5,R0
000B	SUB R0,R1,R3
000C	JUMPi N,R5,0002
000D	SUB R0,R2,R3
000E	JUMPi N,R5,0004
000F	CALL R7,R5,R3
0010	STi R1,R6,0001
0011	STi R2,R6,0002
0012	STi R3,R6,0003
0013	STi R5,R6,0004
0014	LDi R6,R6,0004
0015	SR R6,R0,R6
0016	JUMP R0,R0

```
001C                    CALLi R7,R5,0008
001D                    JUMPi     R0,000B

001E                    CALLi R7,R5,000C
001F                    JUMPi     R0,000D

0020                    ST    R2,R6,R0
0021                    ADD   R2,R3,R0
0022                    LD    R3,R6,R0
0023                    JUMPi   R0,000F

0024                    ADD   R5,R1,R0
0025                    ADD   R1,R2,R0
0026                    ADD   R2,R5,R6
0027                    RTN         R4

0028                    ADD   R6,R1,R0
0029                    ADD   R1,R3,R0
002A                    ADD   R3,R6,R0
002B                    RTN         R4

002D                    LDH   R5,0030
002E                    CALL R7,R5,R0
002F                    RTN         R4

0030                    LDH   R5,0030
0031                    CALLI R7,R5,0003
0032                    RTN         R4
```

```
0033                    LDH    R5,0030
0034                    CALLi  R7,R5,0006
0035                    RTN           R4

0036                    AND    R6,R5,R1
0037                    ANDi   R6,R5,0000
0038                    RTN           R4
```

APPENDIX D

Test Vectors and Test Results

Test Vectors

```
$define Pos Position
$define Len Length
$define Sig Signal
$define In Input, Par="to=1"
$define Out Output, Par="to=2"
$define Expr Expression

Fields{

RESET                          (In, Pos=0, Len=1 )    {Default=0;}
TRUE                           (In, Pos=1, Len=1 )    {Default=1;}
FALSE                          (In, Pos=2, Len=1 )    {Default=0;}

ADDR15                         (Out,Pos=0, Len=16)    {}
DATA15                         (Out,Pos=16,Len=16)    {}
MEM_READ                       (Out,Pos=32,Len=1 )    {}
MEM_WR                         (Out,Pos=33,Len=1 )    {}
PHASE_A                        (Out,Pos=34,Len=1 )    {}
PHASE_B                        (Out,Pos=35,Len=1 )    {}
datapath/dp/BUS_A              (Out,Pos=36,Len=16)    {}
datapath/dp/BUS_B              (Out,Pos=52,Len=16)    {}
datapath/dp/BUS_C              (Out,Pos=68,Len=16)    {}
DATA_BUS                       (Out,Pos=84,Len=16)    {}
memory/memcontr/A_READ         (Out,Pos=100,Len=1 )   {}
memory/memcontr/A_WR           (Out,Pos=101,Len=1 )   {}
memory/memcontr/ROM_READ       (Out,Pos=102,Len=1 )   {}
RT                             (Out,Pos=103,Len=1 )   {}
controller/decode/O1           (Out,Pos=104,Len=1 )   {}
controller/decode/O2           (Out,Pos=105,Len=1 )   {}
controller/decode/O3           (Out,Pos=106,Len=1 )   {}
controller/decode/O4           (Out,Pos=107,Len=1 )   {}
controller/decode/O5           (Out,Pos=108,Len=1 )   {}
controller/finite/F1           (Out,Pos=109,Len=1 )   {}
controller/finite/F2           (Out,Pos=110,Len=1 )   {}
controller/finite/F3           (Out,Pos=111,Len=1 )   {}
controller/finite/F4           (Out,Pos=112,Len=1 )   {}
controller/IR/IR_OUT_EXT1      (Out,Pos=113,Len=16)   {}
controller/RGFILE_WR           (Out,Pos=129,Len=16)   {}
}

Templates{

op[]=RESET\@0;
}

Lineaction::Expr(.=.+5);

Data{

op[0];
op[0];
op[1];
op[1];
op[1];
op[1];
```

```
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[0];
op[0];
op[0];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
```

```
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
```

```
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
```

```
op[1];
op[1];
op[1];
op[1];
op[1];
op[1];
}
```

Test Results

```
RUN_VECTORS
simul
)Running test vector Assembler.
)Created Ancillary file simul.083.SMO & .SXR.
)trace running from simul      Wed Mar 22 23:25:19 1989
)    FTR  c    c    cccccccc cRmmm D    d    d    d    PPMM D    A
)    ARE  o    o    oooooooo oTeee A    a    a    a    HHEE A    D
)    LUS  n    n    nnnnnnnn n mmm T    t    t    t    AAMM T    D
)    SEE  t    t    tttttttt t ooo A    a    a    a    SS__ A    R
)    E T  r    r    rrrrrrrr r rrr _    p    p    p    EEWR 1    1
)         o    o    oooooooo o yyy B    a    a    a     __RE 5    5
)         l    l    llllllll l /// U    t    t    t    BA A [    [
)         l    l    llllllll l mmm S    h    h    h       D 1    1
)         e    e    eeeeeeee e eee [    /    /    /         5    5
)         r    r    rrrrrrrr r mmm 1    d    d    d         :    :
)         /    /    //////// / ccc 5    p    p    p         0    0
)         R    I    ffffdddd d ooo :    /    /    /         ]    ]
)         G    R    iiiieeee e nnn 0    B    B    B
)         F    /    nnnncccc c ttt ]    U    U    U
)         I    I    iiiioooo o rrr      S    S    S
)         L    R    ttttdddd d ///
)         E         eeeeeeee e RAA      C    B    A
)              O    //////// / O__      [    [    [
)         W    U    FFFFOOOO O MWR      1    1    1
)         R    T    43215432 1 _RE      5    5    5
)         [                    R A      :    :    :
)         1    E                E D     0    0    0
)         5    X                A       ]    ]    ]
)         :    T                D
)         0    1
)         ]    [
)              1
)              5
)              :
)              0
)              ]
)    ---  ---- ---- -------- ----- ---- ---- ---- ---- ---- ---- ----
)    bbb  xxxx xxxx bbbbbbbb bbbbb xxxx xxxx xxxx xxxx bbbb xxxx xxxx
rvshowdots 0
qckq
)  0:010 >IIII IIII iiiiiiii iiiii IIII IIII IIII IIII iiii IIII IIII
)  5:010 >IIII IIII iiiiiiii iiiii IIII IIII IIII IIII iiii IIII IIII
) 10:011 >IIII IIII iiii0000 01iii IIII IIII IIII IIII iiii IIII IIII
) 15:011 >IIII IIII iiii0000 01iii IIII IIII IIII IIII iiii IIII IIII
) 20:011 >IIII IIII iiiiiiii i0iii IIII IIII IIII IIII iiii IIII IIII
) 25:011 >IIII IIII iiiiiiii i0000 IIII ffff IIII IIII 01ii IIII ffff
) 30:011 >ffff IIII iiiiiiii i0iii ffff IIII ffff ffff 10ii IIII IIII
) 35:011 >IIII IIII iiiiiiii i0000 IIII ffff IIII IIII 01ii IIII ffff
) 40:011 >ffff IIII iiiiiiii i0iii ffff IIII ffff ffff 10ii IIII IIII
) 45:011 >IIII IIII iiiiiiii i0000 IIII ffff IIII IIII 01ii IIII ffff
) 50:011 >ffff IIII iiiiiiii i0iii ffff IIII ffff ffff 10ii IIII IIII
) 55:011 >IIII IIII iiiiiiii i0000 IIII ffff IIII IIII 01ii IIII ffff
) 60:011 >ffff IIII iiiiiiii i0iii ffff IIII ffff ffff 10ii IIII IIII
) 65:011 >IIII IIII iiiiiiii i0000 IIII ffff IIII IIII 01ii IIII ffff
) 70:010 >ffff IIII iiiiiiii i0iii ffff IIII ffff ffff 10ii IIII IIII
```

```
)  75:010 >IIII IIII iiiiiiii i0000 IIII ffff IIII IIII 01ii IIII ffff
)  80:010 >ffff IIII iiii0000 01iii ffff IIII ffff ffff 10ii IIII IIII
)  85:011 >IIII IIII iiii0000 01000 IIII ffff IIII IIII 01ii IIII ffff
)  90:011 >ffff IIII 00000000 01000 ffff IIII ffff ffff 10ii IIII IIII
)  95:011 >fffe 0000 00001000 00000 ffff ffff ffff ffff 0100 zzzz ffff
)100:011 >fffe 0000 00001000 00000 ffff ffff ffff ffff 0100 zzzz ffff
)105:011 >ffff 0000 00001000 00100 ffff 0000 ffff ffff 1000 zzzz 0000
)110:011 >ffff c501 00000010 10000 c501 ffff ffff ffff 0100 zzzz ffff
)115:011 >ffff c501 00000010 10100 ffff ffff ffff ffff 1000 zzzz 0001
)120:011 >ffdf 11b1 00001100 00000 11b1 ffff 0010 ffff 0100 zzzz ffff
)125:011 >ffff 11b1 00001100 00100 ffff 0010 ffff ffff 1000 zzzz 0002
)130:011 >fffd 12b3 00001100 00000 12b3 ffff 0001 0010 0100 zzzz ffff
)135:011 >ffff 12b3 00001100 00100 ffff 0011 ffff ffff 1000 zzzz 0003
)140:011 >fffb 13b2 00001100 00000 13b2 ffff 0003 0010 0100 zzzz ffff
)145:011 >ffff 13b2 00001100 00100 ffff 0013 ffff ffff 1000 zzzz 0004
)150:011 >fff7 15bc 00001100 00000 15bc ffff 0002 0010 0100 zzzz ffff
)155:011 >ffff 15bc 00001100 00100 ffff 0012 ffff ffff 1000 zzzz 0005
)160:011 >ffdf 16a5 00000100 00000 16a5 ffff 000c 0010 0100 zzzz ffff
)165:011 >ffff 16a5 00000100 00100 ffff 001c ffff ffff 1000 zzzz 0006
)170:011 >ffbf 46c0 00000001 00000 46c0 ffff 001c 001c 0100 zzzz ffff
)175:011 >ffff 46c0 00000001 00100 ffff 0038 ffff ffff 1000 zzzz 0007
)180:011 >ffbf 26d7 00001010 00000 26d7 ffff ffff 0038 0100 zzzz ffff
)185:011 >ffff 26d7 00001010 00100 ffff ffc7 ffff ffff 1000 zzzz 0008
)190:011 >ffbf 86c0 00000111 00000 86c0 ffff 0007 ffc7 0100 zzzz ffff
)195:011 >ffff 86c0 00000111 00100 ffff ffc0 ffff ffff 1000 zzzz 0009
)200:011 >ffbf 2022 00000010 00000 2022 ffff ffff ffc0 0100 zzzz ffff
)205:011 >ffff 2022 00000010 00100 ffff ff80 ffff ffff 1000 zzzz 000a
)210:011 >ffff 58a0 00001001 00000 58a0 ffff 0013 0011 0100 zzzz ffff
)215:011 >ffff 58a0 00001001 00000 ffff fffe ffff ffff 1000 zzzz 000b
)220:011 >ffff 0000 00001000 00000 ffff ffff 0000 001c 0100 zzzz ffff
)225:011 >ffff 0000 00001000 00100 ffff 001c ffff ffff 1000 zzzz 001c
)230:011 >ffff 67b8 00000011 00000 67b8 ffff ffff ffff 0100 zzzz ffff
)235:011 >ffff 67b8 00000011 00000 ffff ffff ffff ffff 1000 zzzz 001d
)240:011 >ff7f 0000 00001000 00000 ffff ffff 0008 001c 0100 zzzz ffff
)245:011 >ffff 0000 00001000 00100 ffff 0024 ffff ffff 1000 zzzz 0024
)250:011 >ffff 1520 00000100 00000 1520 ffff ffff ffff 0100 zzzz ffff
)255:011 >ffff 1520 00000100 00100 ffff ffff ffff ffff 1000 zzzz 0025
)260:011 >feff 1140 00000100 00000 1140 ffff 0000 0011 0100 zzzz ffff
)265:011 >ffff 1140 00000100 00100 ffff 0011 ffff ffff 1000 zzzz 0026
)270:011 >fffd 12a0 00000100 00000 12a0 ffff 0000 0013 0100 zzzz ffff
)275:011 >ffff 12a0 00000100 00100 ffff 0013 ffff ffff 1000 zzzz 0027
)280:011 >fffb 7080 00001011 00000 7080 ffff 0000 0011 0100 zzzz ffff
)285:011 >ffff 7080 00001011 00000 ffff 0011 ffff ffff 1000 zzzz 0028
)290:011 >ffff 0000 00001000 00000 ffff ffff ffff 001d 0100 zzzz ffff
)295:011 >ffff 0000 00001000 00100 ffff 001d ffff ffff 1000 zzzz 001d
)300:011 >ffff 501b 00000101 00000 501b ffff ffff ffff 0100 zzzz ffff
)305:011 >ffff 501b 00000101 00000 ffff ffff ffff ffff 1000 zzzz 001e
)310:011 >ffff 0000 00001000 00000 ffff ffff 000b 0000 0100 zzzz ffff
)315:011 >ffff 0000 00001000 00100 ffff 000b ffff ffff 1000 zzzz 000b
)320:011 >ffff 2023 00000010 00000 2023 ffff ffff ffff 0100 zzzz ffff
)325:011 >ffff 2023 00000010 00100 ffff ffff ffff ffff 1000 zzzz 000c
)330:011 >ffff 5ab2 00001000 00000 5ab2 ffff 0012 0013 0100 zzzz ffff
)335:011 >ffff 5ab2 00001000 00100 ffff 0001 ffff ffff 1000 zzzz 000d
)340:011 >ffff 2043 00000010 00000 2043 ffff ffff ffff 0100 zzzz ffff
)345:011 >ffff 2043 00000010 00100 ffff ffff ffff ffff 1000 zzzz 000e
```

```
)350:011 >ffff 5eb4 00000101 00000 5eb4 ffff 0012 0011 0100 zzzz ffff
)355:011 >ffff 5eb4 00000101 00000 ffff ffff ffff ffff 1000 zzzz 000f
)360:011 >ffff 0000 00001000 00000 ffff ffff 0004 001c 0100 zzzz ffff
)365:011 >ffff 0000 00001000 00100 ffff 0020 ffff ffff 1000 zzzz 0020
)370:011 >ffff b2c0 00000100 10000 b2c0 ffff ffff ffff 0100 zzzz ffff
)375:011 >ffff b2c0 01000100 10100 ffff ffff ffff ffff 1000 zzzz 0021
)380:011 >ffff b2c0 01000100 10000 1260 ffff 0000 ff80 0100 zzzz ffff
)385:011 >ffff b2c0 00000100 10010 ffff ffff ffff ffff 1000 zzzz ff80
)390:011 >ffff 1260 00000100 00000 0011 ffff ffff ffff 0100 zzzz ffff
)395:011 >ffff 1260 00000100 00100 ffff ffff ffff ffff 1000 zzzz 0022
)400:011 >fffb a3c0 00000000 10000 a3c0 ffff 0000 0012 0100 zzzz ffff
)405:011 >ffff a3c0 10000000 10100 ffff 0012 ffff ffff 1000 zzzz 0023
)410:011 >ffff a3c0 10000000 10000 501f ffff 0000 ff80 0100 zzzz ffff
)415:011 >ffff a3c0 00000000 10001 ffff ffff ffff ffff 1000 zzzz ff80
)420:011 >fff7 501f 00000101 00000 0011 ffff ffff ffff 0100 zzzz ffff
)425:011 >ffff 501f 00000101 00000 ffff 0011 ffff ffff 1000 zzzz 0024
)430:011 >ffff 0000 00001000 00000 ffff ffff 000f 0000 0100 zzzz ffff
)435:011 >ffff 0000 00001000 00100 ffff 000f ffff ffff 1000 zzzz 000f
)440:011 >ffff 67a3 00001101 00000 67a3 ffff ffff ffff 0100 zzzz ffff
)445:011 >ffff 67a3 00001101 00000 ffff ffff ffff ffff 1000 zzzz 0010
)450:011 >ff7f 0000 00001000 00000 ffff ffff 0011 001c 0100 zzzz ffff
)455:011 >ffff 0000 00001000 00100 ffff 002d ffff ffff 1000 zzzz 002d
)460:011 >ffff c503 00000010 10000 c503 ffff ffff ffff 0100 zzzz ffff
)465:011 >ffff c503 00000010 10100 ffff ffff ffff ffff 1000 zzzz 002e
)470:011 >feff 67a0 00001101 00000 67a0 ffff 0030 ffff 0100 zzzz ffff
)475:011 >ffff 67a0 00001101 00000 ffff 0030 ffff ffff 1000 zzzz 002f
)480:011 >fbff 0000 00001000 00000 ffff ffff 0000 0030 0100 zzzz ffff
)485:011 >ffff 0000 00001000 00100 ffff 0030 ffff ffff 1000 zzzz 0030
)490:011 >ffff c503 00000010 10000 c503 ffff ffff ffff 0100 zzzz ffff
)495:011 >ffff c503 00000010 10100 ffff ffff ffff ffff 1000 zzzz 0031
)500:011 >f7ff 67b3 00000011 00000 67b3 ffff 003Q ffff 0100 zzzz ffff
)505:011 >ffff 67b3 00000011 00000 ffff 0030 ffff ffff 1000 zzzz 0032
)510:011 >dfff 0000 00001000 00000 ffff ffff 0003 0030 0100 zzzz ffff
)515:011 >ffff 0000 00001000 00100 ffff 0033 ffff ffff 1000 zzzz 0033
)520:011 >ffff c503 00000010 10000 c503 ffff ffff ffff 0100 zzzz ffff
)525:011 >ffff c503 00000010 10100 ffff ffff ffff ffff 1000 zzzz 0034
)530:011 >bfff 67b6 00001010 10000 67b6 ffff 0030 ffff 0100 zzzz ffff
)535:011 >ffff 67b6 01101010 10000 ffff 0030 ffff ffff 1000 zzzz 0035
)540:011 >ffff 67b6 01101010 10000 ffff ffff ffff ffff 0100 zzzz ffff
)545:011 >ffff 67b6 11101010 10010 ffff ffff ffff ffff 1000 zzzz ffff
)550:011 >ffff 67b6 11101010 10000 001c ffff ffff ffff 0100 zzzz ffff
)555:011 >ffff 67b6 00011010 10010 ffff ffff ffff ffff 1000 zzzz fffd
)560:011 >ffff 67b6 00011010 10000 ff80 ffff ffff ffff 0100 zzzz ffff
)565:011 >ffff 67b6 00000011 00010 ffff ffff ffff ffff 1000 zzzz fffb
)570:011 >ffff 67b6 00000011 00000 0010 ffff ffff ffff 0100 zzzz ffff
)575:011 >ffff 67b6 00000011 00000 ffff ffff ffff ffff 1000 zzzz ffff
)580:011 >ffef 0000 00001000 00000 ffff ffff 0006 0030 0100 zzzz ffff
)585:011 >ffff 0000 00001000 00100 ffff 0036 ffff ffff 1000 zzzz 0036
)590:011 >ffff 35a1 00000110 00000 35a1 ffff ffff ffff 0100 zzzz ffff
)595:011 >ffff 35a1 00000110 00100 ffff ffff ffff ffff 1000 zzzz 0037
)600:011 >ffdf 36b0 00001110 00000 36b0 ffff 0013 001c 0100 zzzz ffff
)605:011 >ffff 36b0 00001110 00100 ffff 0010 ffff ffff 1000 zzzz 0038
)610:011 >ffbf 7080 00001011 00000 7080 ffff 0000 0010 0100 zzzz ffff
)615:011 >ffff 7080 00001011 00000 ffff 0000 ffff ffff 1000 zzzz 0039
)620:011 >ffff 0000 00001000 00000 ffff ffff ffff 0035 0100 zzzz ffff
```

```
)625:011 >ffff 0000 00001000 00100 ffff 0035 ffff ffff 1000 ZZZZ 0035
)630:011 >ffff 7080 00001011 00000 7080 ffff ffff ffff 0100 ZZZZ ffff
)635:011 >ffff 7080 00001011 00000 ffff ffff ffff ffff 1000 ZZZZ 0036
)640:011 >ffff 0000 00001000 00000 ffff ffff ffff 0032 0100 ZZZZ ffff
)645:011 >ffff 0000 00001000 00100 ffff 0032 ffff ffff 1000 ZZZZ 0032
)650:011 >ffff 7080 00001011 00000 7080 ffff ffff ffff 0100 ZZZZ ffff
)655:011 >ffff 7080 00001011 00000 ffff ffff ffff ffff 1000 ZZZZ 0033
)660:011 >ffff 0000 00001000 00000 ffff ffff ffff 002f 0100 ZZZZ ffff
)665:011 >ffff 0000 00001000 00100 ffff 002f ffff ffff 1000 ZZZZ 002f
)670:011 >ffff 7080 00000110 10000 7080 ffff ffff ffff 0100 ZZZZ ffff
)675:011 >ffff 7080 11000110 10000 ffff ffff ffff ffff 1000 ZZZZ 0030
)680:011 >ffff 7080 11000110 10000 ffff ffff ffff ffff 0100 ZZZZ ffff
)685:011 >ffff 7080 00101011 00001 ffff ffff ffff ffff 1000 ZZZZ fffb
)690:011 >ff7f 7080 00101011 00000 0010 ffff ffff ffff 0100 ZZZZ ffff
)695:011 >ffff 7080 10101011 00001 ffff 0010 ffff ffff 1000 ZZZZ fffd
)700:011 >ffbf 7080 10101011 00000 ff80 ffff ffff ffff 0100 ZZZZ ffff
)705:011 >ffff 7080 00001011 00001 ffff ff80 ffff ffff 1000 ZZZZ ffff
)710:011 >ffdf 7080 00001011 00000 001c ffff ffff ffff 0100 ZZZZ ffff
)715:011 >ffff 7080 00001011 00000 ffff 001c ffff ffff 1000 ZZZZ ffff
)720:011 >ffff 0000 00001000 00000 ffff ffff ffff 0010 0100 ZZZZ ffff
)725:011 >ffff 0000 00001000 00100 ffff 0010 ffff ffff 1000 ZZZZ 0010
)730:011 >ffff b1d1 00001100 10000 b1d1 ffff ffff ffff 0100 ZZZZ ffff
)735:011 >ffff b1d1 01001100 10100 ffff ffff ffff ffff 1000 ZZZZ 0011
)740:011 >ffff b1d1 01001100 10000 b2d2 ffff 0001 ff80 0100 ZZZZ ffff
)745:011 >ffff b1d1 00001100 10010 ffff ffff ffff ffff 1000 ZZZZ ff81
)750:011 >ffff b2d2 00001100 10000 0013 ffff ffff ffff 0100 ZZZZ ffff
)755:011 >ffff b2d2 01001100 10100 ffff ffff ffff ffff 1000 ZZZZ 0012
)760:011 >ffff b2d2 01001100 10000 b3d3 ffff 0002 ff80 0100 ZZZZ ffff
)765:011 >ffff b2d2 00001100 10010 ffff ffff ffff ffff 1000 ZZZZ ff82
)770:011 >ffff b3d3 00001100 10000 0012 ffff ffff ffff 0100 ZZZZ ffff
)775:011 >ffff b3d3 01001100 10100 ffff ffff ffff ffff 1000 ZZZZ 0013
)780:011 >ffff b3d3 01001100 10000 b5d4 ffff 0003 ff80 0100 ZZZZ ffff
)785:011 >ffff b3d3 00001100 10010 ffff ffff ffff ffff 1000 ZZZZ ff83
)790:011 >ffff b5d4 00001100 10000 0011 ffff ffff ffff 0100 ZZZZ ffff
)795:011 >ffff b5d4 01001100 10100 ffff ffff ffff ffff 1000 ZZZZ 0014
)800:011 >ffff b5d4 01001100 10000 a6d4 ffff 0004 ff80 0100 ZZZZ ffff
)805:011 >ffff b5d4 00001100 10010 ffff ffff ffff ffff 1000 ZZZZ ff84
)810:011 >ffff a6d4 00001000 10000 001c ffff ffff ffff 0100 ZZZZ ffff
)815:011 >ffff a6d4 10001000 10100 ffff ffff ffff ffff 1000 ZZZZ 0015
)820:011 >ffff a6d4 10001000 10000 9606 ffff 0004 ff80 0100 ZZZZ ffff
)825:011 >ffff a6d4 00001000 10001 ffff ffff ffff ffff 1000 ZZZZ ff84
)830:011 >ffbf 9606 00001111 00000 001c ffff ffff ffff 0100 ZZZZ ffff
)835:011 >ffff 9606 00001111 00100 ffff 001c ffff ffff 1000 ZZZZ 0016
)840:011 >ffbf 5000 00001001 00000 5000 ffff 001c ffff 0100 ZZZZ ffff
)845:011 >ffff 5000 00001001 00000 ffff 000e ffff ffff 1000 ZZZZ 0017
)850:011 >ffff 0000 00001000 00000 ffff ffff 0000 0000 0100 ZZZZ ffff
)855:011 >ffff 0000 00001000 00100 ffff 0000 ffff ffff 1000 ZZZZ 0000
)860:011 >ffff c501 00000010 10000 c501 ffff ffff ffff 0100 ZZZZ ffff
)865:011 >ffff c501 00000010 10100 ffff ffff ffff ffff 1000 ZZZZ 0001
)870:011 >ffdf 11b1 00001100 00000 11b1 ffff 0010 ffff 0100 ZZZZ ffff
)875:011 >ffff 11b1 00001100 00100 ffff 0010 ffff ffff 1000 ZZZZ 0002
)880:011 >fffd 12b3 00001100 00000 12b3 ffff 0001 0010 0100 ZZZZ ffff
)vct:test vector file simul end detected
)885:011 >ffff 12b3 00001100 00100 ffff 0011 ffff ffff 1000 ZZZZ 0003
BACK
```

```
KEEP_LOG
) End of GENESIL session 'chiptest'
) ++++++++++++++++++
```

APPENDIX E

Chip Timing Analysis

```
*********************************************************************
***
                Genesil Version v7.0 -- Mon May  8 16:39:58 1989
Chip: ~genzhang/zhang/thesis                                    Timing
Analyzer
*********************************************************************
***
---------------------------------------------------------------------
---
                              TIMING ANALYSIS REPORT
Object Type: CHIP                          Function Type: &CHIP
Technology: C1                             Fab Line:  NSC_CN20A
---------------------------------------------------------------------
---
Setup Files:
index file_name            comment
include
      ___________________ 0  >_________________

---------------------------------------------------------------------
---
Process Corner:
          GUARANTEED          TYPICAL
---------------------------------------------------------------------
---
Operating Conditions:
  Junction temperature:      ---       Supply voltage:      ---
---------------------------------------------------------------------
---
Current Clock Definition:
  Phase 1:      ---                      Phase 2:      ---
---------------------------------------------------------------------
---
```

```
*****************************************************************
***
                Genesil Version v7.0 -- Mon May  8 17:31:32 1989
Chip: ~genzhang/zhang/thesis                                  Timing
Analyzer
*****************************************************************
***
CLOCK REPORT MODE
-----------------------------------------------------------------
---
Fabline: NSC_CN20A                        Corner: TYPICAL
  Junction Temperature:75 degree C        Voltage:5.00v
  External Clock: CLOCK
 Included setup files: default setup file
-----------------------------------------------------------------
---
                              CLOCK TIMES (minimum)
Phase 1 High:    84.9   ns              Phase 2 High:    55.2   ns
            ----------                               ----------
Cycle (from Ph1):   166.9   ns         Cycle (from Ph2):   132.9   ns
                  ----------                               ----------
Minimum Cycle Time:   166.9   ns       Symmetric Cycle Time:  169.9
ns
                     ----------                                -------
---
-----------------------------------------------------------------
---
                              CLOCK WORST CASE PATHS
Minimum Phase 1 high time is    84.9   ns set by:
                               ----------
  ** Clock delay: 7.3ns (92.2-84.9)
   Node                              Cumulative Delay      Transition
   controller/finite/(internal)           92.2             rise
   controller/finite/O5                   90.2             fall
   controller/decode/O5                   90.0             fall
   controller/decode/O5'                  87.9             fall
   controller/decode/FALG[0]              61.5             rise
   controller/flag/FLAG[0]                61.5             rise
   controller/flag/FLAG[0]'               60.9             rise
   controller/flag/FLAG_VAL1[0]           59.2             rise
   controller/flag/FLAG_IN[0]             56.6             rise
   datapath/dp/LT                         55.8             rise
   datapath/dp/LT'                        49.8             rise
   datapath/dp/OVF                        44.2             fall
   datapath/dp/OVF'                       39.7             fall
   datapath/dp/COUT                       36.2             fall
   datapath/dp/COUT'                      32.1             fall
   datapath/dp/BUS_A[6]                   16.0             fall
   datapath/dp/PHASE_A                     9.0             rise
   CLOCK/PHASE_A                           7.1             rise
   CLOCK                                   0.0             fall

Minimum Phase 2 high time is    55.2   ns set by:
                               ----------
  ** Clock delay: 7.2ns (62.5-55.2)
```

Node	Cumulative Delay	Transition
DATA15[0]/(internal)	62.5	rise
DATA15[0]/DATA_DIS	60.9	fall
memory/memcontr/DATA_DIS	60.5	fall
memory/memcontr/DATA_DIS'	41.8	fall
memory/memcontr/M_WR	41.2	rise
memory/memcontr/M_WR'	32.8	rise
memory/memcontr/nram	30.8	rise
memory/memcontr/ol	29.7	fall
memory/memcontr/ol'	29.6	fall
memory/memcontr/ADDR[9]	24.9	fall
memory/membus/ADDRB[15]	24.9	fall
memory/membus/ADDRB[15]'	24.6	fall
memory/membus/ADDR_BUS[15]	22.1	fall
controller/pointer/sp/SP[15]	20.5	fall
controller/pointer/sp/SP[15]'	16.2	fall
<roller/pointer/sp/SP_VAL1[15]	13.7	fall
controller/pointer/sp/PHASE_B	9.3	rise
CLOCK/PHASE_B	7.0	rise
CLOCK	0.0	fall

Minimum cycle time (from Ph1) is 166.9 ns set by:

** Clock delay: 10.6ns (177.5-166.9)

Node	Cumulative Delay	Transition
controller/flag/(internal)	177.5	rise
controller/flag/FLAG_IN[2]	176.7	fall
datapath/dp/GT	176.0	fall
datapath/dp/GT'	171.6	fall
datapath/dp/EQ	168.2	rise
datapath/dp/EQ'	156.8	rise
datapath/dp/ALU_OUT[6]	151.9	fall
datapath/dp/OPCODE[1]	125.9	rise
controller/finite/OPCODE[1]	125.2	rise
controller/finite/OPCODE[1]'	120.5	rise
*controller/finite/(internal)	118.5	fall
controller/finite/O5	90.2	fall
controller/decode/O5	90.0	fall
controller/decode/O5'	87.9	fall
controller/decode/FALG[0]	61.5	rise
controller/flag/FLAG[0]	61.5	rise
controller/flag/FLAG[0]'	60.9	rise
controller/flag/FLAG_VAL1[0]	59.2	rise
controller/flag/FLAG_IN[0]	56.6	rise
datapath/dp/LT	55.8	rise
datapath/dp/LT'	49.8	rise
datapath/dp/OVF	44.2	fall
datapath/dp/OVF'	39.7	fall
datapath/dp/COUT	36.2	fall
datapath/dp/COUT'	32.1	fall
datapath/dp/BUS_A[6]	16.0	fall
datapath/dp/PHASE_A	9.0	rise
CLOCK/PHASE_A	7.1	rise
CLOCK	0.0	fall

Minimum cycle time (from Ph2) is 132.9 ns set by:

** Clock delay: 0ns cycle_sharing disabled

Node	Cumulative Delay	Transition
DATA15[8]	79.1	fall
*DATA15[8]/(internal)	64.1	rise
DATA15[8]/DATA_DIS	61.1	fall
memory/memcontr/DATA_DIS	60.5	fall
memory/memcontr/DATA_DIS'	41.8	fall
memory/memcontr/M_WR	41.2	rise
memory/memcontr/M_WR'	32.8	rise
memory/memcontr/nram	30.8	rise
memory/memcontr/o1	29.7	fall
memory/memcontr/o1'	29.6	fall
memory/memcontr/ADDR[9]	24.9	fall
memory/membus/ADDRB[15]	24.9	fall
memory/membus/ADDRB[15]'	24.6	fall
memory/membus/ADDR_BUS[15]	22.1	fall
controller/pointer/sp/SP[15]	20.5	fall
controller/pointer/sp/SP[15]'	16.2	fall
<roller/pointer/sp/SP_VAL1[15]	13.7	fall
controller/pointer/sp/PHASE_B	9.3	rise
CLOCK/PHASE_B	7.0	rise
CLOCK	0.0	fall
