Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

DESIGN OF COMPONENTS FOR A GENERIC MICROPROCESSOR ARCHITECTURE

by
Pradnesh R. Mohare

The objective of this thesis was to develop a generic microprocessor design that can be adapted to many of the existing 16 bit microprocessors. Common features of various microprocessors were used to develop the design of many generic components which can then be used to design the required microprocessors instead of custom-designing each one of them separately. The components were designed using a CISC based micro-programmed design approach as that was more suitable in terms of design and verification time for generic implementation. The generic parts designed include the Register File for temporary data storage, the Effective Address Calculator that generates the effective address for the operand, the Barrel Shifter for fast multiply/divide operations and the Priority Encoder for determining the processor state.

DESIGN OF COMPONENTS FOR A GENERIC MICROPROCESSOR ARCHITECTURE

by
Pradnesh R. Mohare

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

May 1999

APPROVAL PAGE

DESIGN OF COMPONENTS FOR A GENERIC MICROPROCESSOR ARCHITECTURE

Pradnesh R. Mohare

Dr. Durgamadhab Misra, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin S-H. Hou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Jin-Biao Huang, Committee Member — Date
Member of Technical Staff, Sarnoff Corporation, Princeton, NJ

BIOGRAPHICAL SKETCH

Author: Pradnesh R. Mohare

Degree: Master of Science

Date: May 1999

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1999

- Bachelor of Engineering in Computer Engineering,
  Bombay University, India, 1996

Major: Computer Engineering

Presentations and Publications:

J. Patel, P. Mohare, D. Pattnaik, M. Babladi, N. Patel, A. Patel, S. Sadeq, C. Feng, D. Misra and E. Hou, "Implementation of exp(x) and log(x) generator using Mentor Graphics and Autologic II", *Mid-Atlantic Region Local Users Group (MARLUG '98)*, Baltimore, MD, May 1998.

To my beloved family

ACKNOWLEDGEMENT

The author would like to express his appreciation and sincere gratitude to Dr. Durgamadhab Misra for his invaluable guidance and moral support throughout the research. The author has benefited significantly from the technical discussions with Dr. Misra during this research.

The author is gratefully acknowledging Sarnoff Corporation for the opportunity to work on such an interesting topic of research. He would also like to thank Dr. Edwin Hou and Dr. Jin-Biao Huang for serving as members of the committee.

The author would also like to thank his colleagues at Sarnoff, Mr. Koen Verhaege, Mr. Glenn Vinogradov, Mr. George Guo, Mr. Robert Ho and Mr. Nicola Fedele for their co-operation and help.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background

Recently, the semiconductor industry is making tremendous progress in the form of advanced digital signal processors, faster memories and ever increasingly powerful microprocessors. The constant endeavor of the manufacturers to improve the performance of their products while cutting down on costs using advanced technology has reduced the product life spans. In addition, adverse working conditions like temperature, humidity, voltage fluctuations and aging can further reduce the working life span of the chips.

An important decision has to be made when the procurement of replacements is needed or when the chips that fail and are not in current manufacturing cycle. This has tremendous impact on systems design and implementation. A feasible solution is to reverse-engineer these chips.

1.2 Objective

The focus of this thesis is to design a generic 16-bit microprocessor that can be readily used as a replacement part for most systems and could be adapted to many of the existing 16-bit microprocessors. This requires the availability of stringent chip specifications. These can be obtained from existing samples, performance data, technical specifications and available running code. After analyzing the existing approaches, a novel approach is proposed in this thesis towards the design of a generic microprocessor. This new

approach involves the design of generic components, which can be customized for final, generic microprocessor design.

A review of popular existing 16-bit microprocessors from Intel, Motorola, Zilog, Cypress, Advanced Micro Devices and SGS was carried out. Two possible approaches were evaluated - 1) RISC-based hardware emulation architectural approach and 2) CISC based micro-programmed design approach. The CISC based architecture was found more suitable in terms of design and verification time for generic implementation.

The investigation of different microprocessor architectures and the design development of a generic microprocessor architecture have been described in this thesis. Chapter two discusses the fundamentals of computer architectures and the design approach to be followed. Chapter three describes the architecture of the parts designed. Chapter four contains the simulation results for the parts designed. Chapter five presents a summary of the entire work done and throws some light on the possible future work.

CHAPTER 2

STANDARD MICROPROCESSORS

2.1 Background

Two approaches to implementing a generic microprocessor architecture were considered. To implement a generic microprocessor Vinogradov [1] has followed a top-down approach in which the architecture functionality is a superset of the functions of the existing microprocessor architectures. The architecture described consists of a RISC core controlled by a lookup micro-code ROM. An external interface controller to emulate the exact behavior of all external interfaces was proposed. The RISC core would execute one instruction per clock cycle and to maintain compatibility would generate no-operation instructions for timing purposes. In another paper, Smith et al. [2] describe a method for automated composition of hardware components. The work describes the automated design of interfaces between two hardware components. Given the component model that describes bus functionality or a superset of the bus functionality, conditions for transferring data to or from the component are determined. A sequence of assignments to component ports is determined that would make those conditions come true. A Finite State Machine is then generated that executes the required assignments and monitors the necessary control ports.

The top down approach suggested by Vinogradov [1] however would have a considerably higher design time and would be very complex for verification purposes as well. It is important to note that most of the target microprocessors are CISC based. While the design assumes execution of one instruction per clock cycle memory

references would take more cycles and thus hamper the pipelined execution flow. To counter this Vinogradov proposes a delayed store approach. However, this approach leads to further problems like concurrency control thereby complicating the design even further. Another problem would be the different instruction formats for each processor. While the top-down approach suggested by Smith et al. does seem feasible for automated interface design the approach is not feasible for microprocessors considering design complexity. A bottom-up approach involving the design of generic components that could be used to design an existing microprocessor has been proposed in this thesis. A thorough investigation of some of the microprocessors was carried out to understand the architectures, instruction sets and addressing modes of the microprocessors. Most of the microprocessors currently used for different applications are considered. A list of the microprocessors studied is given in table 2.1

Table 2.1 List of studied microprocessors

Part number	Manufacturer	Device type
8085	Intel	Microprocessor
6800	Motorola	Microprocessor
8048	Intel	Microprocessor
8031	Intel	Microprocessor
8002	Zilog, AMD	Microprocessor
8086	Intel	Microprocessor
8748	Intel	Microprocessor
80	Zilog	Microprocessor
8051	Intel	Microprocessor
8035	Intel	Microprocessor
2901	AMD	Bit-slice
8000	Zilog, AMD	Microprocessor
8751	Intel	Microprocessor
8080	Intel	Microprocessor
49c402	IDT	Bit-slice

2.2 Systems

To design a generic 16-bit microprocessor we investigated some of the system fundamentals such as system classification and instruction set architectures

2.2.1 System Classification

System architectures where microprocessors are integrated are divided into three classes as proposed by Flynn [5] based on the number of instruction and data streams that they have. They are

1) Single Instruction stream over Single Data stream (*SISD*)

2) Single Instruction stream over Multiple Data stream (*SIMD*)

3) Multiple Instruction stream over Multiple Data stream (*MIMD*)

The traditional *von Neumann* machine is SISD . It has one instruction stream executed by one CPU and one memory containing its data. The first instruction is fetched from memory and then executed. Then the next instruction is fetched, executed, and so on.

SIMD machines operate on multiple data sets in parallel. The data sets can be arranged in the form of a vector ALU with multiple inputs or as an array processor in which a single control unit broadcasts instructions which are carried out by each processor using its own memory.

In an MIMD machine different CPUs carry out different programs and share information by using common memory or by passing messages amongst themselves. The machine has multiple instruction streams and multiple data streams.

All the microprocessors studied as a part of this work and mentioned in table 2.1, are SISD machines.

2.2.2 Types of Architectures

There are two types of computer architectures [6]

- *Harvard architecture*
- *Von Neumann architecture*

2.2.2.1 Harvard Architecture: In Harvard type architecture, instructions and data are stored in separate memory modules. This architecture provides a significant advantage such that the designer can provide greater flexibility with word-size selection. Increase in bandwidth allows an overlapping of instruction and data access and consequently increase performance. The instruction size need not be equal to or a factor of the data word size.

2.2.2.2 Von Neumann Architecture: The von Neumann architecture consists of the CPU, instruction/data memory and the input/output. It has been used in many systems for a very long time. However, sequential and frequent access of memory creates bottlenecking and limits the flexibility of operation. The control flow as opposed to data flow leads to performance limitation. Also the bandwidth is reduced as memory and data must reside in the same memory.

A classical von Neumann-type computer is divided into two main modules, the control unit and the data-path unit. The control unit synchronizes the entire operation of

the computer by generating control signals that activate the particular components involved in the execution of an instruction.

The data unit contains the components that are used to manipulate the operands involved in the instruction. The data unit consists of registers and the Arithmetic Logic Unit (ALU). The registers hold the ALU input while the ALU is computing

The ALU performs addition, subtraction and other simple operations on its inputs, yielding a result in the output register. The output register can be stored back into a register and from there, back into memory, if needed.

2.2.3 Instruction Set Architectures

The collection of instructions available to the programmer at a level is called the instruction set of that level. There are two main types of instruction set architectures

- *Reduced Instruction Set Computing(RISC)*
- *Complex Instruction Set Computing(CISC)*

A machine is classified as a RISC or a CISC depending upon the following factors as described in table 2.2.

Table 2.2 Comparison of RISC and CISC Architectures

	CISC	RISC
1	Complex instructions taking multiple cycles	Simple instructions taking 1 cycle
2	Any instruction may reference memory	Only Load/Stores reference memory
3	Less pipelining, if present at all	Highly pipelined
4	Instructions interpreted by micro-program	Instructions executed by hardware
5	Variable format instructions	Fixed format instructions
6	Many instructions and modes	Few instructions and modes
7	Complex micro-program	Complex compiler
8	Single Register Set	Multiple Register Sets

All the microprocessors studied except the Z8000 family are micro-programmed. The Z8000 family may also be implemented using micro-programming.

2.2.4 Types of Instructions

Instructions can be divided into three categories.

- ***Register-Memory***
- ***Register- Register***
- ***Memory-Memory***

Register memory instructions allow memory words to be fetched into registers where they can be used as ALU inputs in subsequent instructions. Register-register instructions fetch two operands from the registers into the ALU input registers, perform operations on them and put the results back into a register. A memory-memory instruction fetches its operands from memory into the ALU registers, performs its operation and stores the results back into memory.

2.2.5 Addressing Modes

An effective address is an address that contains an operand and is part of the instruction word. This address consists of two sub-fields, the mode and the register. The mode bits define the addressing mode of the instruction and the register bits designate the register involved. To specify an operand completely an effective address may need additional information.

2.2.6 Pipelining

Performance of microprocessors has been increased by faster clock speeds. This does however have some limitations. Another technique that is used to improve the performance is pipelining. Pipelining is a form of parallel processing since several operations of the contents of the pipeline occur simultaneously. The advantage of an instruction pipeline is in staging the activity associated with the instruction execution in such a way that the time needed to decode the instruction and the time needed to provide the control for the execution is not visible outside the chip. This is done as a sequence of the following steps

1) *Instruction Fetch*
2) *Instruction Decode*
3) *Operand Fetch*
4) *Instruction Execution*
5) *Write back*

Instruction Fetch

The Program counter is loaded with the address of the next instruction to be executed. The next instruction from memory is then fetched using the Bus Interface Unit and stored in the Instruction Register.

Op-code Decode

The instruction stored in the Instruction Register is then decoded. The operation code (op-code) in the instruction gives information on what the instruction is supposed to do.

This op-code is then used to point to a particular address in the micro-code ROM which then issues relevant signals for controlling various functions like the ALU, stack, Bus Interface Unit, Register file, Interrupt Mechanism.

Operand Fetch

The operand for the instruction may be available in the instruction itself or has to be calculated using the 5 stage Finite State Machine. The effective address of the operand is thus calculated. The operand may be fetched from the register file, program counter, stack pointer or from an external memory location.

Instruction Execution

The micro-program location contains the signals needed for the execution of the instruction for enabling/disabling certain multiplexers, providing timing information and so on. The Instruction Execution Control supervises all the activities and also takes care of activities like interrupt and trap handling, bus conditions and execution state control.

Write Back

The results of the execution may be placed in a register or a memory location. Upon execution the program counter is updated with the address of the next instruction to be executed, which may be one of the following

1. The previous value of the program counter plus the word length
2. A direct address specified as in the case of jump or subroutine instructions
3. A return address popped from the stack

4. A relative address formed by adding an offset to the current value
5. An address generated from a lookup table for interrupts
6. The starting address following a reset condition.

This calculated value is then put into the program counter as the address of the next instruction to be fetched. The Instruction Execution Control specifies when the Program counter is updated.

Thus, if t_1, t_2, t_3, t_4, t_5 are the times required for the respective steps, then the time required for a non-pipelined processor is $(t_1 + t_2 + t_3 + t_4 + t_5)$ and the throughput, T_{np} is $1/(t_1 + t_2 + t_3 + t_4 + t_5)$. However, if t_b is the speed of the slowest point in the pipeline, the throughput of the pipelined processor is $1/t_b$ and is the maximum throughput of the pipelined processor.

2.3 Important Considerations

The design of a generic architecture has to consider compatibility issues at various levels. They are as follows

- *Software compatibility*
- *Object-code compatibility*
- *Hardware compatibility*

Software compatibility refers to the portability of software programs across hardware platforms and architectures. It usually falls within the purview of system software design and hence beyond the scope of this thesis.

Object-code compatibility, also referred to as binary compatibility refers to the same binary or object code being able to run seamlessly on different chips which may be radically different internally but present the same external interface.

Hardware compatibility refers to the issues of physical, pin-out, structural, electrical and timing compatibility. Physical aspects include operating temperature range, power dissipation etc. Structural and pin compatibility usually refer to chip-replaceable design by taking into account the package size, pin count and arrangement. Electrical characteristics include signal levels, operating voltage specifications, noise margins and susceptibility, power calculations etc. Hardware compatibility usually falls within the purview of the physical level and package design and hence not much emphasis was given to it.

2.4 Design Approaches

Two possible approaches to design a generic architecture were studied. They were

- *CISC component-based architecture*
- *RISC based Hardware emulation*

The CISC component-based approach involved the design of generic components that could be used as-is or with a little modification into the design of the microprocessors mentioned in table 2.1. The RISC based approach involved design of a RISC core which would translate the target architecture's instructions into native format. This approach was found to be too complex and infeasible[4] as it would have to take into account vital timing issues thereby increasing the design complexity.

2.4.1 CISC Component-based Architecture

The architecture is composed of two units namely the Control Unit and the Datapath Unit. The Control Unit is responsible for coordinating and synchronizing the various activities inside the microprocessor while the Datapath Unit deals with manipulation of data. Internal data is passed between the two units as int_data for data and address purposes. Figure 2.1 shows the top structure of the microprocessor. Int_trap_ctrl handles interrupt and trap conditions. Int_ctrl is the internal control for the Bus interface unit. ALU_control is used for controlling the ALU sources, destination and operation, while Reg_ctrl is used for controlling the register file. Bus_ctrl is responsible for management of the data and address busses by generating requisite control signals. CPU_ctrl is used for controlling the state of the microprocessor like run state, wait state, reset etc. Address and Data are the contents to be put on the address and data busses respectively.

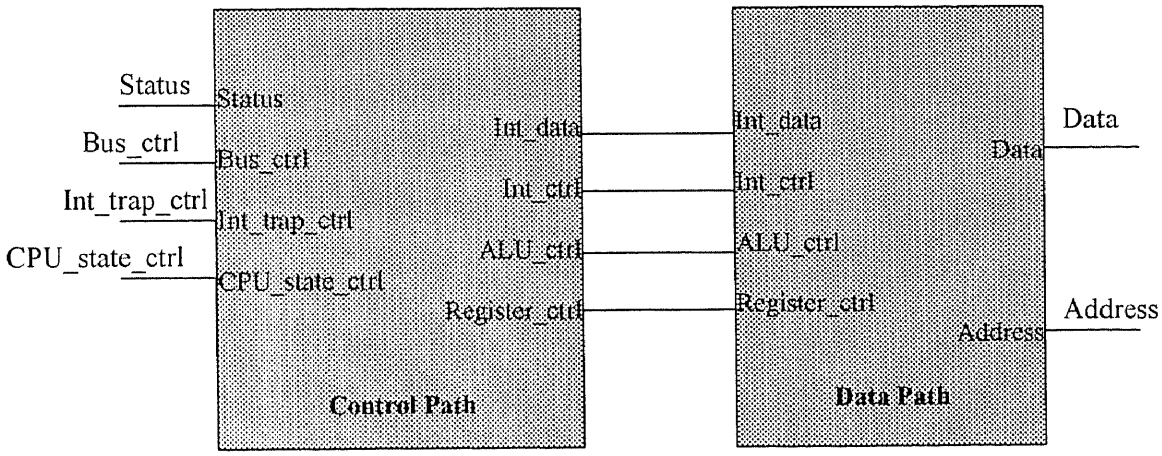

Figure 2.1 The top structure of the microprocessor

The Datapath Unit as shown in Figure 2.2 consists of the Arithmetic Logic Unit (ALU), the Register File and the Address/Data Interface, commonly known as the Bus Interface

Unit. Status signals the current status of the microprocessor like an interrupt acknowledge, memory refresh, internal operation etc. Data A and Data B are the output busses of the register file while int_data is the input. Addr and Data are the address and data bus interfaces respectively.

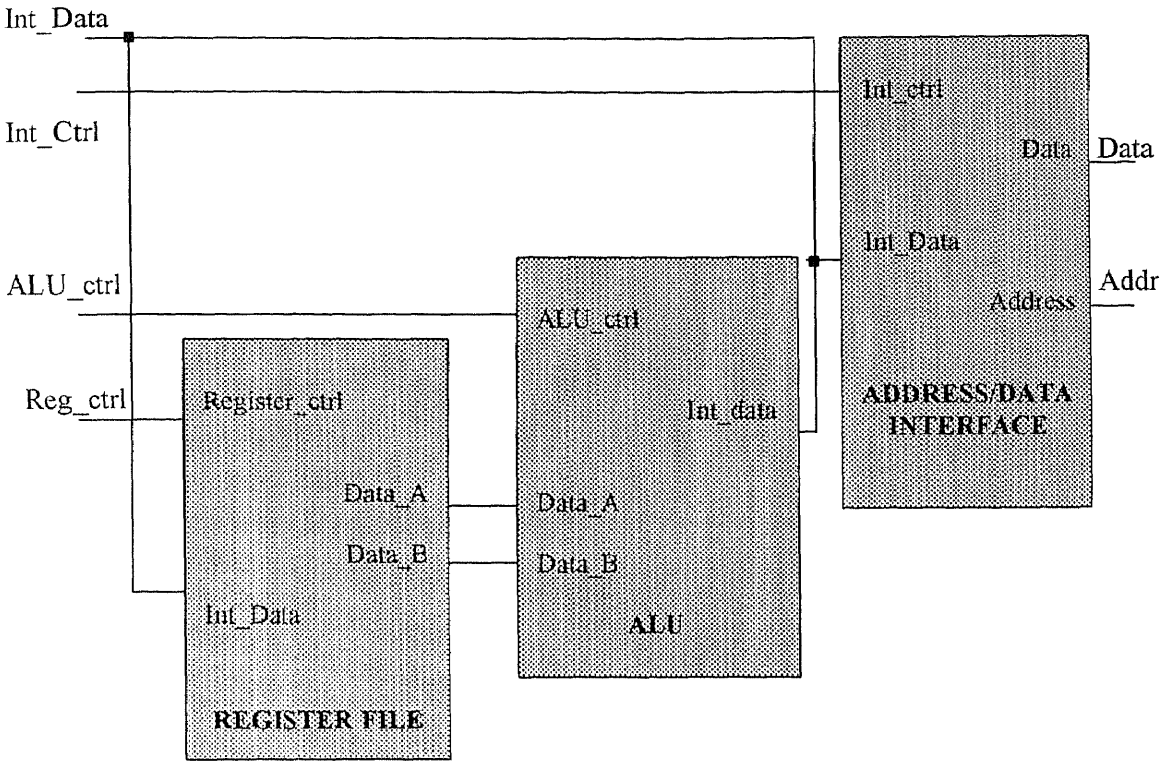

Figure 2.2 Data path structure of the microprocessor

Figure 2.3 shows the Control Path for the microprocessor. The input address to the microcode ROM is multiplexed from the output of the instruction decoder, interrupt/trap control, stack output and the Program counter. Ctrl_path_ctrl are the control signals for the various internal multiplexers. Branch provides a branch address to a ROM location.

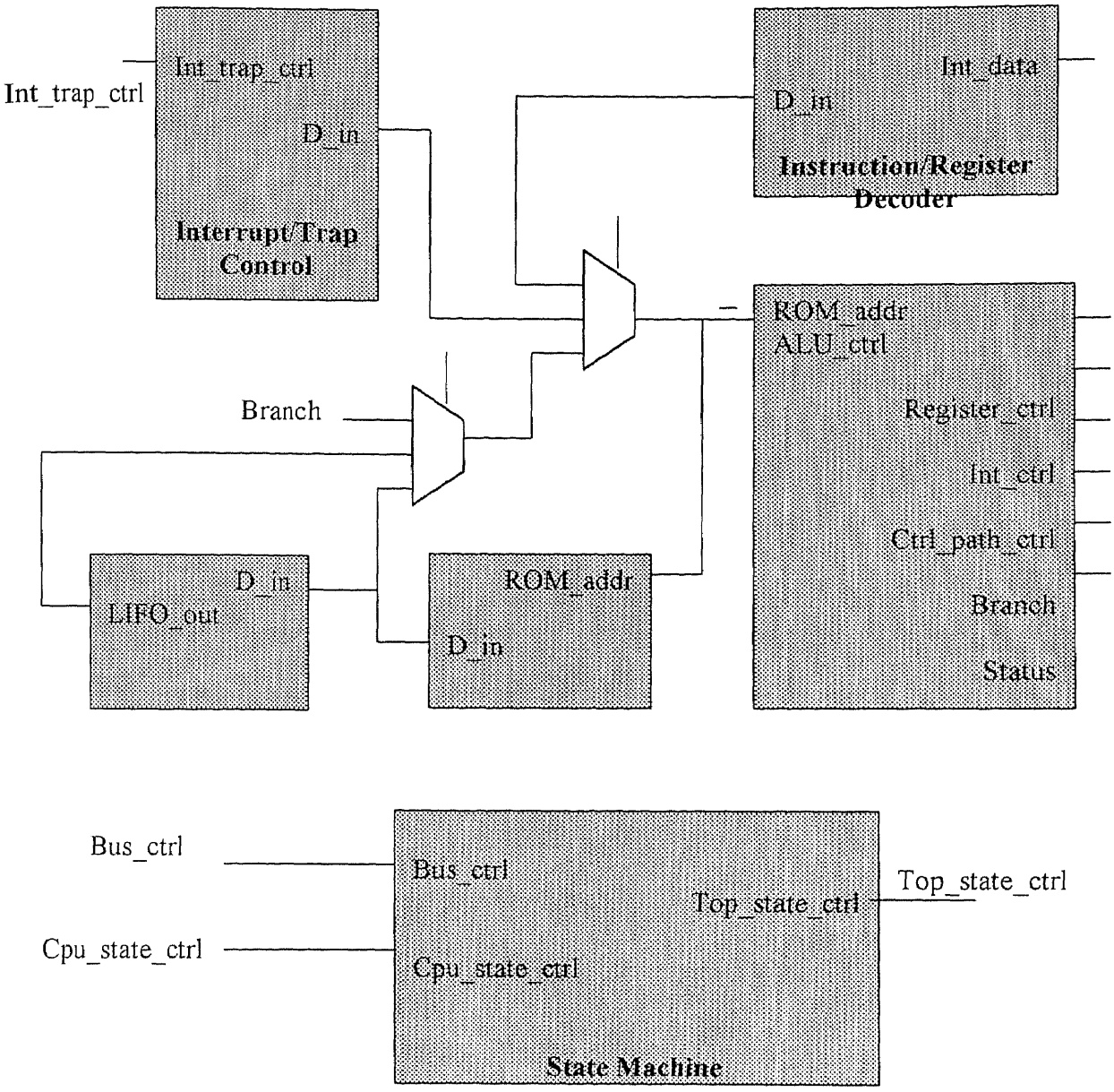

Figure 2.3 Control Path Structure of the Microprocessor

CHAPTER 3

DESIGN

3.1 Introduction

A generic microprocessor is designed using a CISC component-based architecture. Some of the fundamental components to process diverse instruction sets are required for a generic microprocessors. In this design we selected some of the important components normally present in most of the microprocessors. The generic components designed are

- ***Register File***
- ***Barrel Shifter***
- ***Effective Address Calculator***
- ***Priority Encoder***

The Register File is used for temporary data storage. Most of the instructions of the microprocessors studied are register-register or register-memory. The instruction word supplies information about the addressing mode and other related information that can then be used to calculate the effective address of the operand using the Effective Address Calculator.

The microprocessor can be in various states of operation like reset, run, interrupt etc. The states are accorded priorities using a Priority Encoder and the microprocessor can be put into the proper state. The Barrel Shifter is used at the ALU output or built into the ALU for shift/rotate as well as multiply/divide operations. The detailed designs of the individual components are described in the following section.

3.2 Register File

3.2.1 Introduction

The registers are used primarily to store data temporarily during the execution of a program. The instruction set is closely tied to the register file architecture.

3.2.2 Design

The register file has been designed as four eight-byte memory blocks as shown in figure 3.1. Each memory block is dual-ported with one input bus and two output busses. It has two separate addresses that access two register locations whose contents are then put on the two output buses. When the instruction is using only one register the two addresses may be the same but when the instruction uses two different registers the two addresses may be different.

3.2.3 Access Modes

The register file may be used in both byte as well as word modes. Byte operations use two of the four memory blocks. When the register file is in byte mode the lower eight registers are in the LSB memory having addresses 0 through 7 and the higher registers are located in the LSB memory having addresses 8 through 15 as shown in the input address of the register file in figure 3.1. When the address references byte locations 0 to 15, the addresses are passed onto the memory in the same way it was received, but the MSB block of memory is inactive. The 8-bit data from the memory is directed to the output port and may be positioned in the lower eight bits of the data bus with the upper eight bits being zeros or replicated into the upper eight bits.

Word operations use both the memory blocks. The addresses are passed to both the blocks as the memory is divided into a lower byte, D_{7-0} and an upper byte, D_{15-8} in each memory block that are combined together to form a word (Figure 3.1). So, word locations 7-0 would be in the lower block while locations 15-8 would be in the upper block. Double or Quadruple words require two and four accesses into the register file respectively. For double word operations there are eight registers available while for quadruple word operations there are four registers available.

3.2.4 Generality

The register file size can be increased to 32 or more words also. In case of processors from the Intel family, one of the addresses is fixed (usually 0) and is known as the ***accumulator***. This does not need any modifications to the register file design except hardwiring one of the addresses to a constant value.

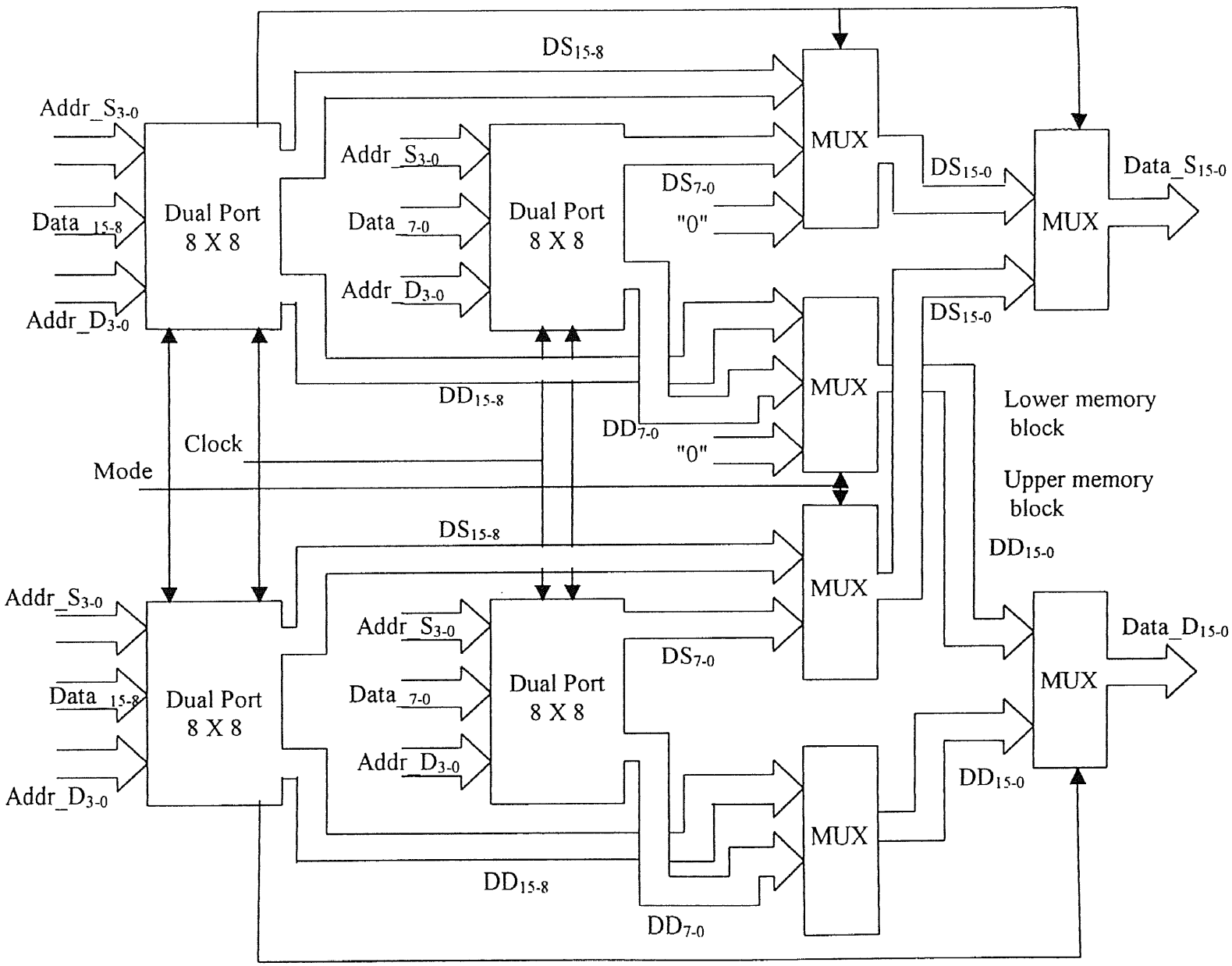

Figure 3.1 Block Diagram of the Register File

3.3 Barrel Shifter

3.3.1 Introduction

The Barrel shifter takes its input from the ALU. The output may then be fed back to the register file or to any other appropriate destination. Unlike conventional barrel shifters that manipulate input busses depending upon the control inputs, the barrel shifter designed as part of this work can implement fast shifts/rotates to facilitate speed-up of operations like multiplication/division. This is necessary for processors like those from the Z8000 family. The Barrel shifter can handle sixteen bits at a time. The shifter is capable of the following

- ***Shift Left***
- ***Shift Right***
- ***Rotate Left***
- ***Rotate Right***

The barrel shifter can shift or rotate 16 bits at a time as opposed to one or two bits per clock cycle. Thus it can be up to 16 times faster than a normal shifter.

3.3.2 Applications

The barrel shifter can be used in a dedicated 16-bit multiplication/division unit for fast computation. The Z8000 family has an in-built barrel shifter. The barrel shifter can be used as a normal shifter by hardwiring the number of shift/rotate positions to 1.

Multiplication is implemented as successive shifts and adds. For a 16-bit by 16-bit multiplication we require sixteen 16-bit shifts and sixteen add operations at the maximum. These 256 shifts would require 256 shift cycles. The barrel shifter on the other hand shifts

sixteen bits at a time and would hence require a maximum of only sixteen shift cycles to implement the 16-bit multiplication.

3.4 Effective Address Calculator

3.4.1 Introduction

The location of the operand for a particular instruction is known as the effective address. The effective address depends on the addressing mode. The microprocessors studied[9-19] have modes that are listed in table 3.1 e.g. the Z8002 has eight different addressing modes which is a superset of the addressing modes of all the other microprocessors. Table 3.2 shows the addressing modes and the corresponding effective addresses of the operand. EA is the effective address while R_n is the address of register n and D_{16} is the direct data needed.

Table 3.1 Addressing Modes

Part number	Addressing modes
8085	Register, Immediate, Indexed, Direct, Implied
6800	Accumulator, Immediate, Direct Address, Extended, Indexed, Implied, Relative
8048	Register, Implied, Immediate
8031	Register, Direct Address, Immediate, Indirect
8002	Register, Direct Address, Immediate, Indirect, Indexed, Base Address, Base Indexed, Relative Address
8086	Immediate, Register, Direct Address, Implied, Indexed, Indirect, Base Indexed
8748	Register, Implied, Immediate
80	Immediate, Immediate Extended, Relative, Relative Extended, Register, Implied, Indirect, Bit
8051	Register, Direct Address, Immediate, Indirect
8035	Register, Implied, Immediate
2901	Register, Immediate
8000	Register, Direct Address, Immediate, Indirect, Indexed, Base Address, Base Indexed, Relative Address
8751	Register, Direct Address, Immediate, Indirect
8080	Register, Immediate, Indexed, Direct, Implied
49c402	Register, Immediate

Table 3.2 Effective addresses

Addressing Mode	Mod	Effective Address
Register	000	EA = An
Indirect	001	EA = $[R_n]$
Immediate	010	Data = D_{16}
Direct Address	011	EA = $[D_{16}]$
Indexed	100	EA = $[R_n] + D_{16}$
Base Address	101	EA = $D_{16} + [R_n]$
Relative Address	110	EA = PC + D_{16}
Base Indexed	111	EA = $[R_{n1}] + [R_{n2}]$

3.4.2 Implementation

These addresses can be calculated using a five-stage Mealy-type Finite State Machine. The Instruction Decoder decodes the instruction to give the op-code, addressing mode, registers to be used as well as the word/byte mode. The addressing mode and the register(s) specified in the instruction are used as inputs to the state machine. The output of the state machine is the effective address whose contents are to be used for the instruction as an operand. As seen from the table 3.3, the first state of the state machine is to fetch the instruction. The addressing mode and the register(s) specified in the instruction are used as inputs to the state machine to calculate the next state. The next state may actually correspond to an actual fetch or a no-op. The state is determined from the addressing mode of the instruction. If mode is not (00X), a fetch occurs. The finite state machine asserts a register read or a memory read signal depending upon the op-code.

Table 3.3 State Table for the Finite State Machine

Addressing Mode	State 1	State 2	State 3	State 4	State 5
Indexed	Fetch Instruction	Fetch Base Address	BA + Rs = Operand Address	Fetch contents @Operand Address	Execute
Direct Address	Fetch Instruction	Fetch Direct Address	SKIP	Fetch contents @ DA	Execute
Indirect	Fetch Instruction	SKIP	SKIP	Fetch contents @Rs	Execute
Base Address	Fetch Instruction	Fetch Displacement	Disp + Rs = Operand Address	Fetch contents @Operand Address	Execute
Register	Fetch Instruction	SKIP	SKIP	SKIP	Execute
Immediate	Fetch Instruction	Fetch Immediate Data	SKIP	SKIP	Execute
Relative Address	Fetch Instruction	Fetch Displacement	PC + Disp= Operand Address	Fetch contents @Operand Address	Execute
Base Indexed	Fetch Instruction	Fetch Index	BA + Index = Operand Address	Fetch contents @Operand Address	Execute

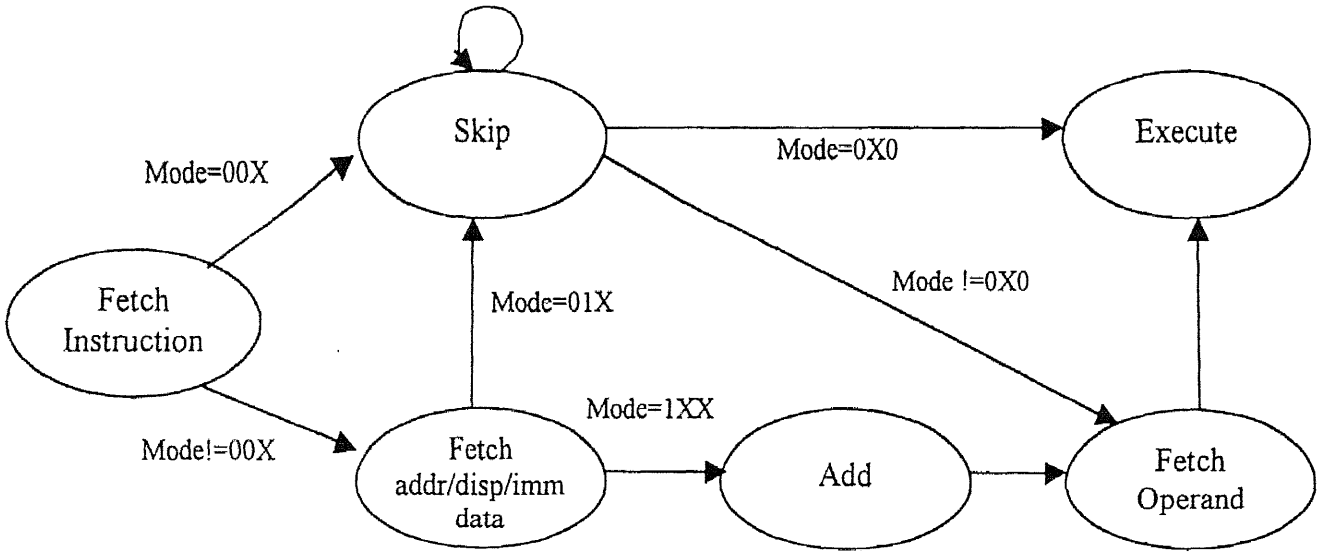

Figure 3.2 State Transition Diagram for the Finite State Machine

The effective address of the operand is available as the output of the fourth state. Register/Memory read is asserted during the fetch operand state. The ALU_ctrl signals are activated during the add stage as well as during the execute stage depending upon the instruction op-code. Register/memory writes are issued during the execute cycle. The algorithm for the working is presented below

1) Fetch Instruction

    1.1) Assert mem_addr=PC, mem_read

2) Decode Instruction

3) Get the addressing mode, mode

4) If mode != 000 goto 4.2.1

    4.1.1) Assert reg_read, reg_addr

    4.1.2) Perform operation

    4.1.3) Assert reg_write, reg_addr

    4.1.4) Write back

    4.1.5) Goto 5

    4.2.1) If mode != 001 goto 4.3.1

    4.2.2) Assert reg_write, reg_addr

    4.2.3) get mem_addr= reg_contents

    4.2.4) Assert mem_read, mem_addr

    4.2.5) Perform operation

    4.2.6) Write back

    4.2.7) Goto 5

    4.3.1) If mode != 010 goto 4.4.1

4.3.2) mem_addr = PC + word_size

4.3.3) Assert mem_read, mem_addr

4.3.4) Get mem_contents

4.3.5) Perform operation

4.3.6) Assert reg_addr, reg_write

4.3.7) Write back

4.3.8) Goto 5

4.4.1) If mode != 011 goto 4.5.1

4.4.2) mem_addr = PC + word_size

4.4.3) Assert mem_read, mem_addr

4.4.4) get mem_contents

4.4.5) mem_addr=mem_contents

4.4.6) Assert mem_read, mem_addr

4.4.7) Perform operation

4.4.8) Write back

4.4.8) Goto 5

4.5.1) If mode != 100 goto 4.6.1

4.5.2) mem_addr = PC + word_size; assert reg_addr, reg_read

4.5.3) Assert mem_read, mem_addr; get reg_contents

4.5.4) mem_addr = reg_contents + mem_contents

4.5.5) Assert mem_read, mem_addr

4.5.6) Perform operation

4.5.7) Write back

4.5.8) Goto 5

4.6.1) If mode != 101 goto 4.7.1

4.6.2) mem_addr = PC + word_size; assert reg_addr, reg_read

4.6.3) Assert mem_read, mem_addr; get reg_contents

4.6.4) mem_addr = reg_contents + mem_contents

4.6.5) Assert mem_read, mem_addr

4.6.6) Perform operation

4.6.7) Write back

4.6.8) Goto 5

4.7.1) If mode != 110 goto 4.8.1

4.7.2) mem_addr = PC + word_size

4.7.3) Assert mem_read, mem_addr

4.7.4) get mem_contents

4.7.5) mem_addr = PC + mem_contents

4.7.6) Assert mem_read, mem_addr

4.7.7) Perform operation

4.7.8) Write back

4.7.9) Goto 5

4.8.1) Assert reg_read, reg_addr

4.8.2) get reg_contents1, reg_contents2

4.8.3) mem_addr = reg_contents1 + reg_contents2

4.8.4) Assert mem_read, mem_addr

4.8.4) Assert reg_addr, reg_write

4.8.5) Write back

5) PC=PC + word_size

6) Goto 1

3.5 Priority Encoder

3.5.1 Introduction

The microprocessors studied have the following control pins and interrupt modes that affect the operation of the CPU. They are listed in table 3.4

Table 3.4 CPU operating modes

Part number	Interrupts	Other control pins
8085	NMI,INT,RST 5.5,6.5,7.5	READY,HOLD,RESET
6800	NMI,IRQ, RST	HALT,TSC,RESET
8048	INT	RESET
8031	INT0,INT1	RESET,PSEN,EA/VPP
8002	NMI,VI,NVI	BUS_REQ, STOP, WAIT, RESET
8086	INT, NMI	READY,HOLD,RESET
8748	INT	RESET
80	INT, NMI	HALT, WAIT, BUS-REQ, RESET
8051	INT0,INT1	RESET,PSEN,EA/VPP
8035	INT	RESET
2901	None	None
8000	NMI,VI,NVI	BUS_REQ, STOP, WAIT, RESET
8751	INT0,INT1	RESET,PSEN,EA/VPP
8080	INT	READY,HOLD,RESET
49c402	None	None

The microprocessor may be in a particular operation mode at any given time. The microprocessor can be in the run mode, reset mode, interrupt mode etc. The next instruction to be executed depends upon the mode in which the microprocessor is. The Program Counter is loaded with a particular value depending upon this mode.

The microprocessor state can be determined by using a Priority Encoder. The inputs to the encoder are the control pins as well as the external interrupt signals. The priority encoder encodes the priority of the signal available at that particular pin and makes it available at the output. A signal at a higher priority masks out signals with lower priorities so that only the highest priority signal is encoded at the output. These priorities can be configured as per specific needs. The priority encoder may be modified to store the signal requests into latches so that they may be implemented once all other higher priority requests are attended to. This could also be implemented as a modified programmable interrupt controller. The priority encoder can also be cascaded further to increase the number of priority inputs.

Table 3.5 Truth Table of 8 input Priority Encoder

INPUTS									OUTPUTS			
EI	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	A_2	A_1	A_0	EO
L	X	X	X	X	X	X	X	X	L	L	L	L
H	L	L	L	L	L	L	L	L	L	L	L	H
H	H	X	X	X	X	X	X	X	H	H	H	L
H	L	H	X	X	X	X	X	X	H	H	L	L
H	L	L	H	X	X	X	X	X	H	L	H	L
H	L	L	L	H	X	X	X	X	H	L	L	L
H	L	L	L	L	H	X	X	X	L	H	H	L
H	L	L	L	L	L	H	X	X	L	H	L	L
H	L	L	L	L	L	L	H	X	L	L	H	L
H	L	L	L	L	L	L	L	H	L	L	L	L

3.5.2 Implementation

The circuit diagram for the encoder is shown below

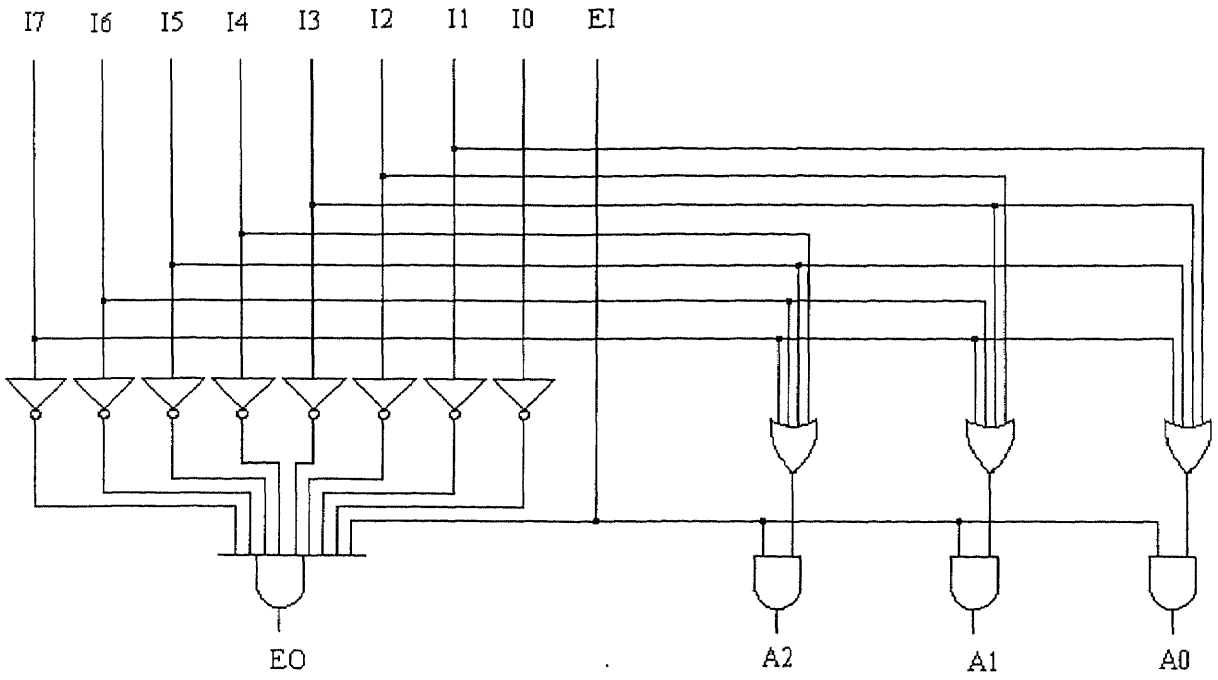

Figure 3.3 Circuit diagram for the Priority Encoder

The inputs to the priority encoder like the reset signal, interrupt, etc are given as inputs I7-0 while the encoded output is available as A2-0. These are used by the Instruction Execution Control to determine the state of the microprocessors and determine the value to be used as the Program Counter.

CHAPTER 4

SIMULATION RESULTS

4.1 Introduction

The following parts designed have been simulated.

- *Register File*
- *Barrel Shifter*
- *Effective Address Calculator*
- *Priority Encoder*

The results have been compared against expected operations and found to be satisfactory. The Barrel Shifter and Priority Encoder have been implemented in VHDL code. The simulations are as follows

4.2 Register File

Most of the common instructions used in the microprocessors have been tested against the register file design for compliance. Mode refers to the Byte/Word Mode to be used in the register file. Adr_S and Adr_D are the Source and Destination addresses for read access into the register file and Data_S & Data_D are the corresponding outputs from the register file onto the two data busses. Data is written into the register file from the Data signal using the D address as the destination. The register file thus has two read ports and one write port. Complex instructions can be split up as sequences of primitives using the instruction decoder and the micro-code ROM that can then be executed sequentially. Table 4.1 shows the working of the register file for common instructions of the microprocessors studied.

Table 4.1 Working of the Register File for different microprocessors

CPU	Instruction	Mode	Adr_S	Adr_D	Data_S	Data_D	Adr_D	Data
8085	Add C	Byte	C	A	[C]	[A]	A	[A] + [C]
6800	SBA	Byte	B	A	[B]	[A]	A	[A]-[B]
Z80	CP HL	Byte	[HL]	A	[[HL]]	[A]	---	----------
8051	XRL C	Byte	C	A	[C]	[A]	A	[C] X-Or [A]
8086	CMP BX,CX	Word	CX	BX	[CX]	[BX]	---	----------
Z8002	LD R2,R5	Word	R5	---	[R5]	---	R2	[R5]
8048	ANL A,C	Byte	C	A	[C]	[A]	A	[A] And [C]
8085	LDAX B	Byte	BC	---	[BC]	---	A	[BC]*
6800	CLRA	Byte	---	---	---	---	A	"0"
Z80	XOR B	Byte	B	A	[B]	[A]	A	[B] X-Or [A]
8031	MOV A,@R3	Byte	---	R3	---	[R3]	A	[[R3]]
8086	INC CX	Word	---	CX	---	[CX]	CX	[CX] + 1
Z8000	INB RH0, @R1	Byte	R1	---	[R1]	---	RH0	[[R1]] **
8748	MOV A, #4F	Byte	---	---	---	---	A	4F
8085	ANA B	Byte	B	A	[B]	[A]	A	[A] AND [B]
6800	LDAA Z,X	Byte	X	Z	[X]	[Z]	A	[[X] + Z]
Z80	RLCA	Byte	---	A	---	[A]	A	Rot(A)
8751	MUL A,B	Byte	A	B	[A]	[B]	A	[A] * [B]
8086	SUB CX, BX	Word	BX	CX	[BX]	[CX]	CX	[CX]-[BX]
Z8000	LD R0, R2(R5)	Word	R2	R5	[R2]	[R5]	R0	[[R2]+[R5]]
8035	INC A	Byte	---	A	---	[A]	A	[A] + 1

The Register File would have inputs from the ALU, Instruction Execution Control Unit, Bus Interface Unit and its own output looped back, all controlled by an input source multiplexer.

4.3 Barrel Shifter

The barrel shifter has been designed and implemented using VHDL code. The simulation was done using the QuickSim VHDL simulator from Mentor Graphics. The results of the simulation are as under. The input to the barrel shifter is the 16-bit data signal, "d". Shift/rotate operation is determined using the "s_r" signal with s_r = 0 being a shift operation and s_r = 1 being a rotate operation. Direction control is done using the "l_r" signal with l_r = 0 for left and l_r = 1 for right. The number of bit positions to be shifted is specified using the 4-bit "n_shift" signal. The barrel shifter may take input from the ALU or be integrated

into the ALU itself. Figures 4.1 throsugh 4.5 show the operation of the Barrel Shifter for different input conditions.

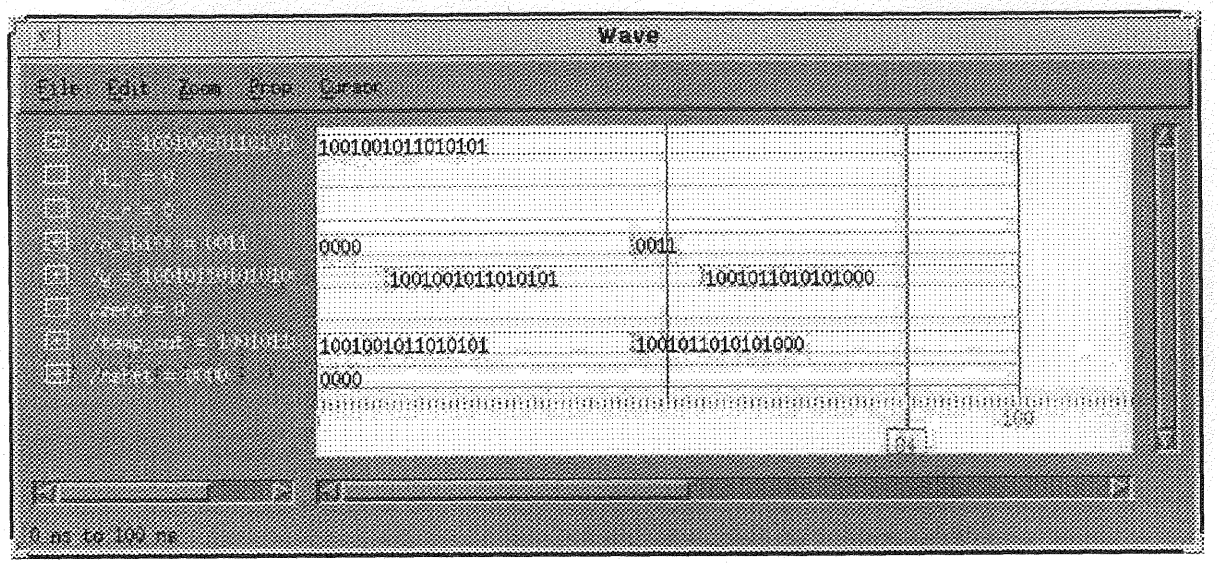

Figure 4.1 Barrel Shifter output for a 3 bit left-shift operation

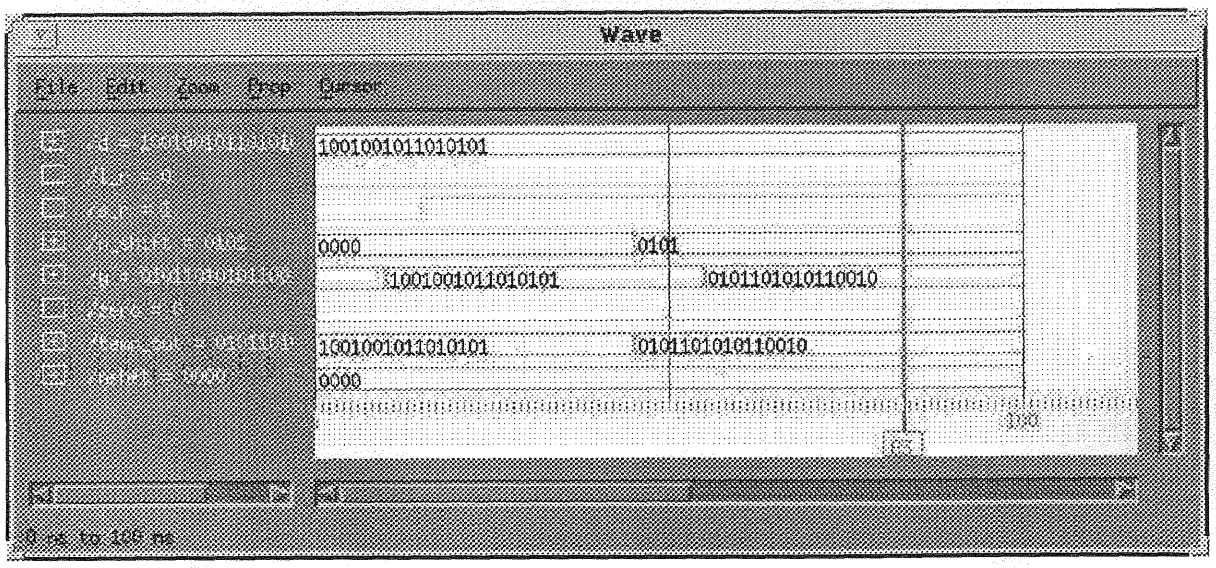

Figure 4.2 Barrel Shifter output for a 5 bit left-rotate operation

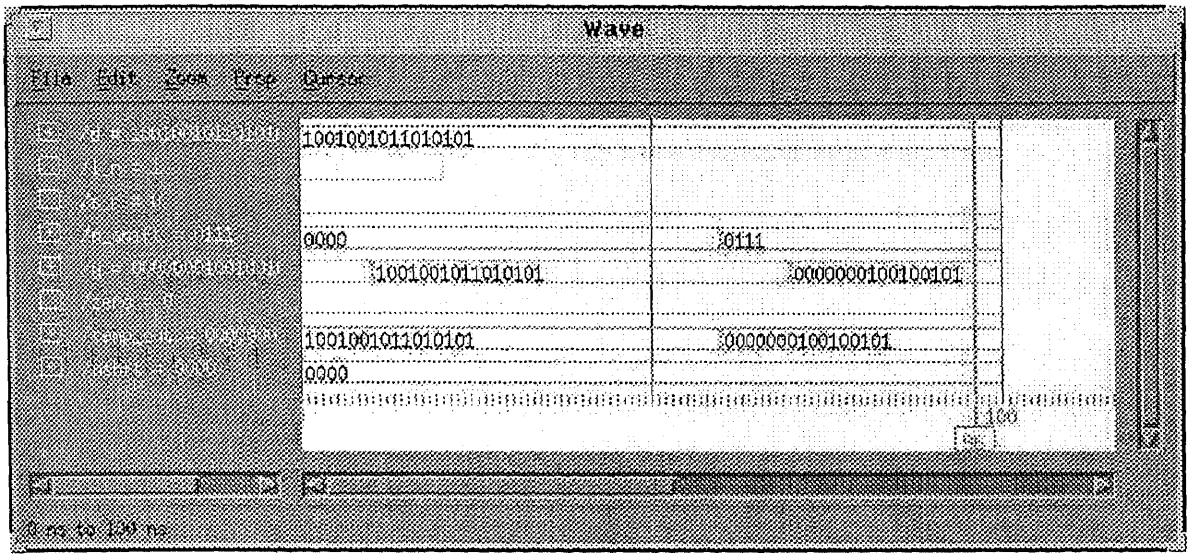

Figure 4.3 Barrel Shifter output for a 7 bit right-shift operation

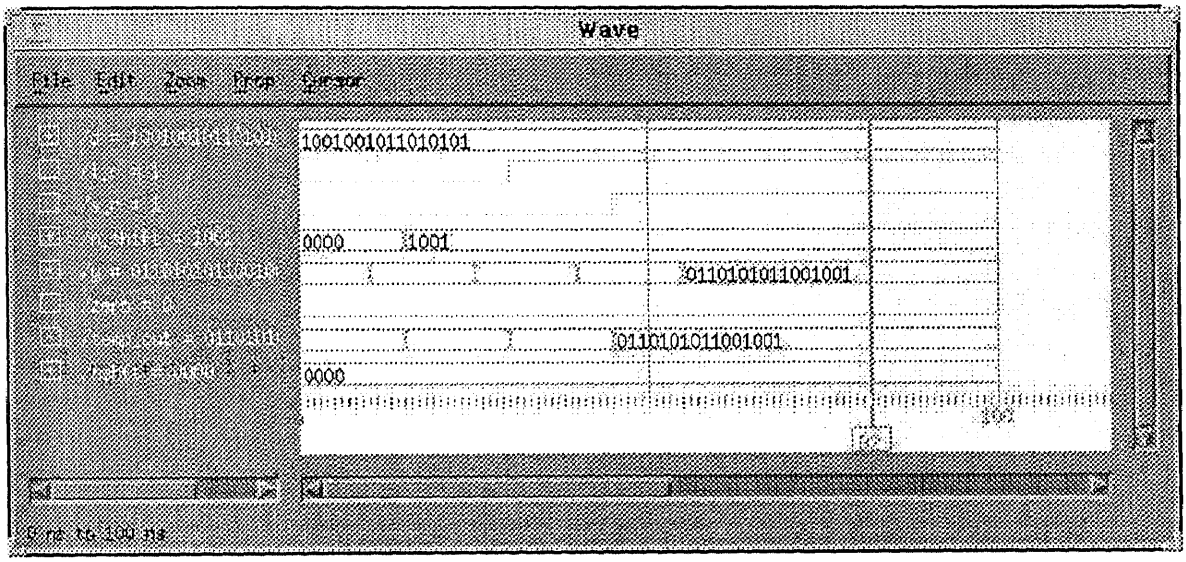

Figure 4.4 Barrel Shifter output for a 5 bit right-rotate operation

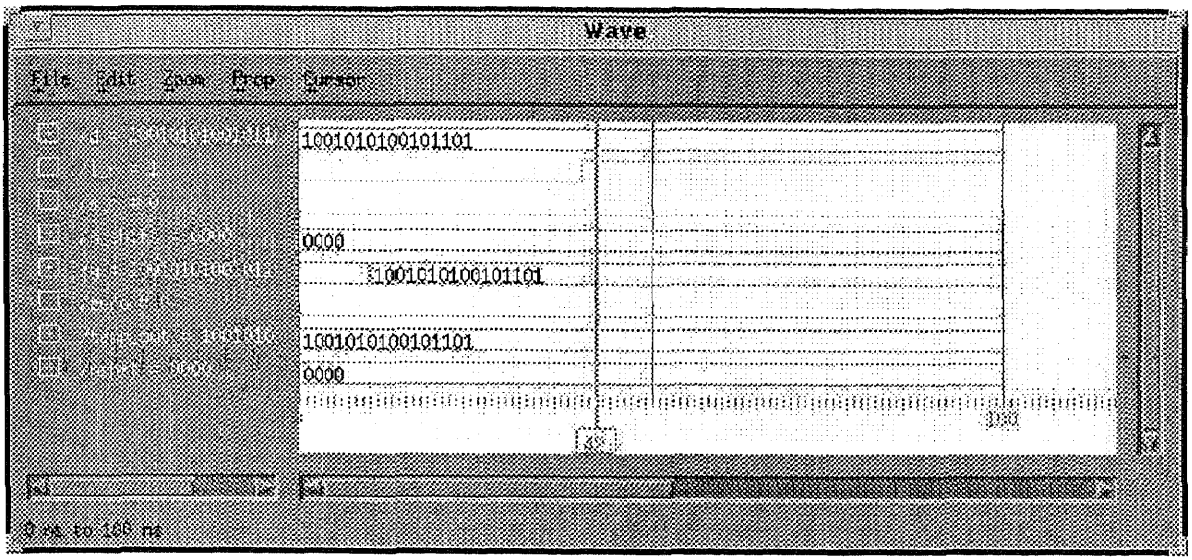

Figure 4.5 Barrel Shifter output for a 0 bit right-rotate operation

4.4 Effective Address Calculator

The addressing modes of the instructions used in the microprocessors have been tested against the Effective Address Calculator design for compliance. Table 4.2 shows the working of the Finite State Machine for the addressing modes of the microprocessors studied. All the possible addressing modes of the microprocessors studied have been considered and evaluated for conformance and the Finite State Machine has been found to work satisfactorily.

Table 4.2 Working of the Finite State Machine for different microprocessors

CPU	Instruction	Addr Mode	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
8085	Add C	Register	Fetch Inst	Skip	Skip	Skip	[A] = [C] + [A]
6800	SBA B	Accumulator	Fetch Inst	Skip	Skip	Skip	[A] = [A] - [B]
Z80	JR 14H	Relative	Fetch Inst	Fetch disp = 14H	EA=[PC] +14H	Fetch @EA	Jump to EA
8051	Mov A,B	Register	Fetch Inst	Skip	Skip	Skip	[A] = [B]
8086	Add CX, [BX]	Indirect Reg	Fetch Inst	Skip	EA=[BX]	Fetch @EA	[CX]=[CX]+[[BX]]
Z8002	LDB R0, %4A90(R1)	Base address	Fetch Inst	Fetch disp = 4A90H	EA=[R1]+4A90	Fetch @EA	[R0]=[[R1]+4A90]
8048	Mov A, #4F	Immediate	Fetch Inst	Fetch immed data=4FH	Skip	Skip	[A] = 4FH
8085	LHLD 3910H	Direct Addr	Fetch Inst	Fetch direct addr=3910H	Skip	Fetch @EA	[H,L]=[3910,1]
6800	LDAB #$78	Immediate	Fetch Inst	Fetch immed data=78H	Skip	Skip	[B] = 78H
Z80	JP 2080H	Extended	Fetch Inst	Fetch immed data=2080H	Skip	Skip	Jump to 2080H
8031	Mov A, @R3	Indirect Reg	Fetch Inst	Skip	EA = [R3]	Fetch @EA	[A] = [[R3]]
8086	Add BX, [437AH]	Direct Addr	Fetch Inst	Fetch direct addr=437AH	Skip	Fetch @EA	[BX]=[BX]+[437A]
Z8000	LD R0, R2(R5)	Base Index	Fetch Inst	Fetch index = [R5]	EA=[R2]+[R5]	Fetch @EA	[R0]=[[R2]+[R5]]
8748	ANL A,C	Register	Fetch Inst	Skip	Skip	Skip	[A]=[A]And[C]
8085	MVI A, 45H	Immediate	Fetch Inst	Fetch immed data=45H	Skip	Skip	[A] = 45H
6800	LDAA $2A, X	Indexed	Fetch Inst	Fetch base addr=2AH	EA=[X]+2AH	Fetch @EA	[A]=[[X]+2AH]
Z80	ADD B	Register	Fetch Inst	Skip	Skip	Skip	[A] = [A] + [B]
8751	Add A, 25H	Immediate	Fetch Inst	Fetch immed data=25H	Skip	Skip	[A] = [A] + 25H
8086	Mov AX, [BX]	Indirect Reg	Fetch Inst	Skip	EA = [BX]	Fetch @EA	[AX] = [[BX]]
Z8000	LD R5, %2A	Relative Addr	Fetch Inst	Fetch disp = 2AH	EA=[PC]+2AH	Fetch @EA	[R5]=[[PC]+2AH]
8035	INC A	Implied	Fetch Inst	Skip	Skip	Skip	[A] = [A] + 1

4.5 Priority Encoder

The 8-input Priority Encoder has been designed and implemented using VHDL code. The simulation was done using the QuickSim VHDL simulator. The inputs to the Priority Encoder are the 8 signals $I_{7\text{-}0}$. A is the encoded output signal. E_I is Enable input while E_O is the enable output that can be used for expanding the input size by cascading multiple units. The Priority Encoder has been simulated and found working satisfactorily. The results of the simulation are as shown in figures 4.6 through 4.9.

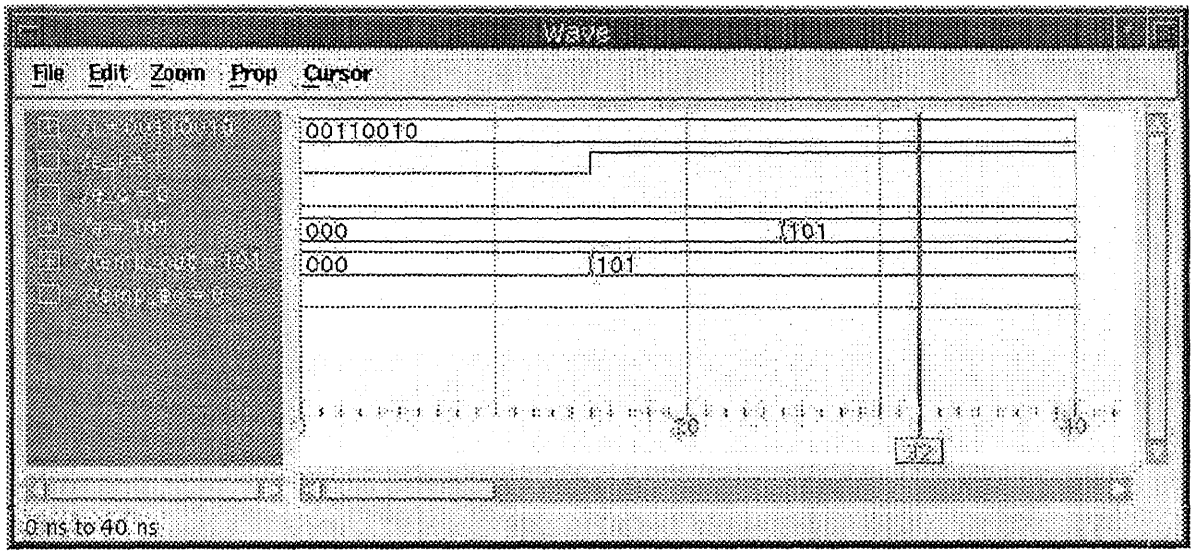

Figure 4.6 Priority Encoder output for input = 00110010 and input enabled

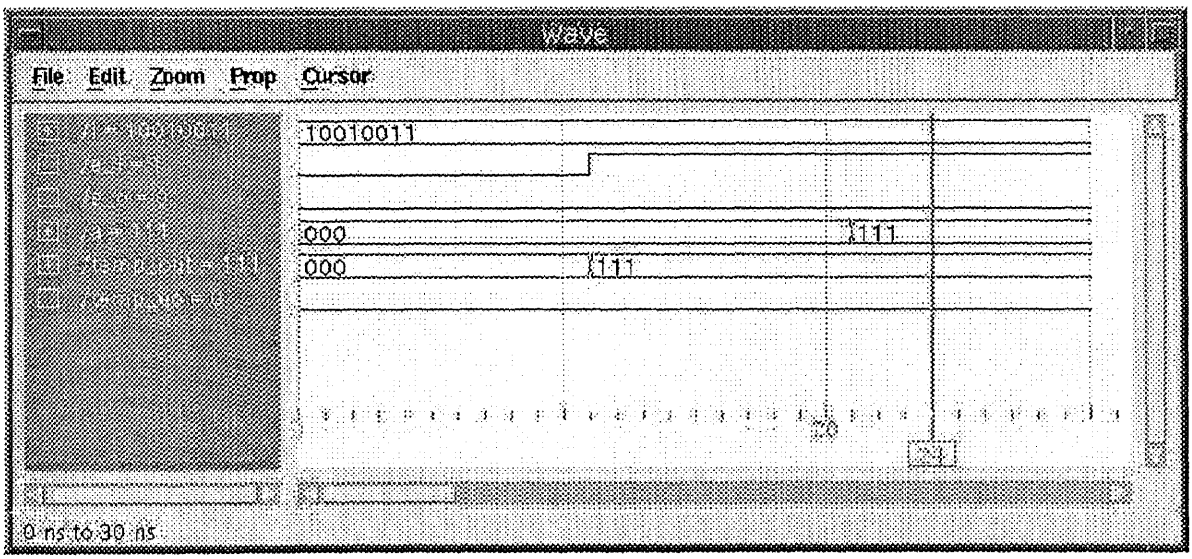

Figure 4.7 Priority Encoder output for input = 10010011 and input enabled

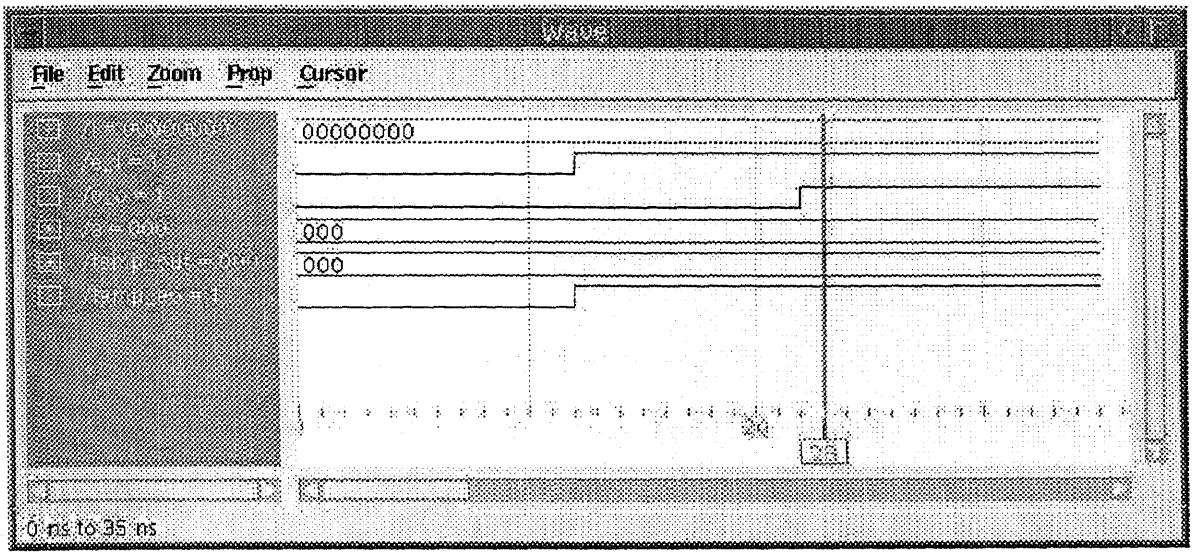

Figure 4.8 Priority Encoder output for input = 00000000 and input enabled

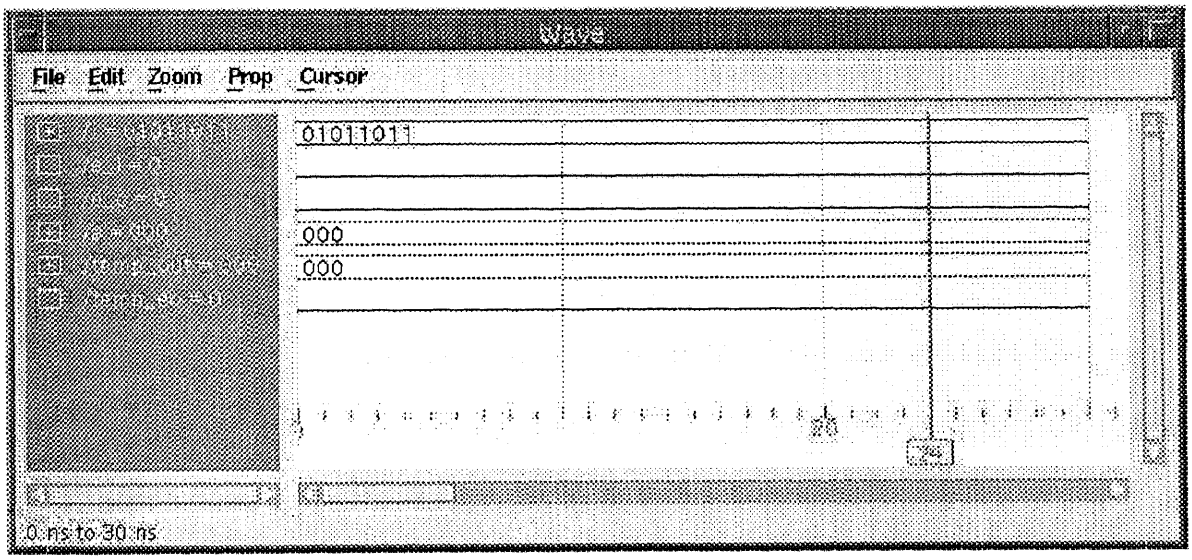

Figure 4.9 Priority Encoder output for input = 01011011 and input disabled

4.6 Event Sequence

The following sequence of instructions/events exercise the parts designed

1. Add CX, [BX]
2. ROL CX,1
3. A reset signal received during the execution of the rotate instruction.

The following events occur internally

- The Instruction Execution Control orders the BIU to fetch the first instruction Add CX, [BX].
- The instruction is then decoded by the instruction decoder and the effective address calculator determines the addressing mode to be indirect register.
- The decoder then points to the proper micro-code ROM which issues signals to the register file with Address S = BX and Address D = CX, mode = 1 and asserting the read signal.
- The register file outputs this data on the S and D data busses.
- The Instruction Execution Control then puts this data as an address on the BIU that fetches data from that memory location.
- The ALU then adds this data to the contents on the D Data bus and the Micro-code ROM under the control of the Instruction Execution Control puts this result into the register file with Address D = CX and the result as data. The write signal is also asserted
- The Instruction Execution Control then Instructs the BIU to fetch the next instruction ROL CX, 1

- The instruction is fetched into the Instruction Register and decoded by the instruction decoder. The effective address calculator determines the addressing mode as register. The instruction decoder then points to an appropriate location in the micro-code ROM. The micro-code ROM then generates control signals into the register file with address S = CX, mode = 1 and asserting the read signal.
- The register file then puts the contents of the CX register on the S Data Bus.
- The Instruction Execution Control now supplies this data to the Barrel Shifter, which then performs a left rotate operation by two bit positions.
- The Instruction Execution Control now puts this data into the register file with address D = CX, mode = 1 and asserting the Write Signal.
- A reset signal was received during the execution of the rotate instruction. The Priority encoder determines that there is no other signal that has higher priority and signals the Instruction Execution Control to jump to the starting address.
- The Instruction Execution Control then puts the value of the default starting address on the BIU and the instruction at that address is fetched.
- The implementation of this may be customized as to resetting to the starting address after completion of the current instruction or resetting by aborting the current instruction execution.

CHAPTER 5

SUMMARY

The CISC component-based approach was found feasible for the implementation of a generic microprocessor architecture to implement the studied microprocessors. The major components have been identified for generic implementation and the Register File, Effective Address Calculator, Barrel Shifter and Priority Encoder have been designed and simulated. VHDL models for the Barrel Shifter and Priority Encoder were developed and simulation results were in good agreement with the expected performance. A novel approach to design the Effective Address Calculator was followed in the form of a finite state machine such that all the existing microprocessors can use this generic component. The design of the Effective Address Calculator can handle eight different addressing modes. The Register File designed, as part of this thesis is capable of handling 8 to 32 bit-wide data. Some of the parts are processor-specific and hence beyond the scope of this work.

5.1 Conclusions

The CISC component-based approach was found feasible for the implementation of a generic microprocessor architecture as it has a lower design time and consequently a lower verification time. The approach is the most effective way to make a generic microprocessor to replace target microprocessors. There is a little penalty in the form of increased chip size that is offset by advances sub-micron in fabrication technology.

5.2 Future Work

The following is a summary of the future work that needs to be done and the improvements/modifications that can be done to the current design.

- The remaining components - the ALU, Instruction Execution Control, Bus Interface Unit and the micro-code ROM need to be designed.
- Sophisticated features like super-scalar architectures, branch target buffer, etc. could also be implemented in the generic architecture
- The design of a fully generic microprocessor architecture could be considered. This design would be completely compatible with the microprocessors studied and can also be custom configured. This may even be further modified into a chip-replaceable design with the customization being done at the hardware manufacturing stage itself.
- The final design could be optimized for speed, power and chip-area.

APPENDIX

VHDL SOURCE CODE

A.1 Barrel Shifter

```
-- Barrel_Shifter.vhd
-- This is to create a 16-bit barrel shifter
-- Works for any 4-bit value of L/R  Shift/Rotates including 0 and 15.

LIBRARY ieee, std;
LIBRARY work;

USE ieee.std_logic_1164.all;
USE work.common_pkg.all;
USE ieee.std_logic_arith.all;
USE std.textio.all;

ENTITY barrel_shifter IS
PORT(
            D :       IN std_data;
          L_R :       IN BIT;        -- L/R select,0=shift left,1=right
          S_R :       IN BIT;        -- Shift/Rotate 0 = Shift, 1= Rotate
      n_shift :       IN bit_vector(3 DOWNTO 0);
            Y :       OUT std_data
);
END barrel_shifter;

ARCHITECTURE behav OF barrel_shifter IS

CONSTANT  max_prop_delay : TIME := 10 ns;
SIGNAL              temp_out : std_data;
SIGNAL                  nshft : bit_vector(3 DOWNTO 0);

PROCEDURE conv_to_int (SIGNAL     vect : IN bit_vector( 3 DOWNTO 0);
                       variable int_out : OUT INTEGER ) IS
BEGIN
CASE vect(3) IS
  WHEN '0' => CASE vect(2) IS
                WHEN '0' => CASE vect(1) IS
                              WHEN '0' => CASE vect(0) IS
                                            WHEN '0' => int_out := 0;
                                            WHEN '1' => int_out := 1;
                                            WHEN  OTHERS => NULL;
```

```
                                    END CASE;
                                WHEN '1' => CASE vect(0) IS
                                            WHEN '0' => int_out := 2;
                                            WHEN '1' => int_out := 3;
                                            WHEN  OTHERS => NULL;
                                           END CASE;
                                WHEN  OTHERS => NULL;
                               END CASE;
                    WHEN '1' => CASE vect(1) IS
                                WHEN '0' => CASE vect(0) IS
                                            WHEN '0' => int_out := 4;
                                            WHEN '1' => int_out := 5;
                                            WHEN  OTHERS => NULL;
                                           END CASE;
                                WHEN '1' => CASE vect(0) IS
                                            WHEN '0' => int_out := 6;
                                            WHEN '1' => int_out := 7;
                                            WHEN  OTHERS => NULL;
                                           END CASE;
                                WHEN  OTHERS => NULL;
                               END CASE;
                    WHEN  OTHERS => NULL;
                   END CASE;
WHEN '1' => CASE vect(2) IS
                    WHEN '0' => CASE vect(1) IS
                                WHEN '0' => CASE vect(0) IS
                                            WHEN '0' => int_out := 8;
                                            WHEN '1' => int_out := 9;
                                            WHEN  OTHERS => NULL;
                                           END CASE;
                                WHEN '1' => CASE vect(0) IS
                                            WHEN '0' => int_out := 10;
                                            WHEN '1' => int_out := 11;
                                            WHEN  OTHERS => NULL;
                                           END CASE;
                                WHEN  OTHERS => NULL;
                               END CASE;
                    WHEN '1' => CASE vect(1) IS
                                WHEN '0' => CASE vect(0) IS
                                            WHEN '0' => int_out := 12;
                                            WHEN '1' => int_out := 13;
                                            WHEN  OTHERS => NULL;
                                           END CASE;
                                WHEN '1' => CASE vect(0) IS
                                            WHEN '0' => int_out := 14;
                                            WHEN '1' => int_out := 15;
```

```
                                        WHEN  OTHERS => NULL;
                                      END CASE;
                                WHEN  OTHERS => NULL;
                               END CASE;
                      WHEN  OTHERS => NULL;
                 END CASE;
          WHEN  OTHERS => NULL;
END CASE;
END;

PROCEDURE shifter (SIGNAL        D : IN std_data;
                   SIGNAL     nshft : IN bit_vector(3 DOWNTO 0);
                   SIGNAL      L_R : IN BIT;
                   SIGNAL tmp_out : OUT std_data) IS

CONSTANT filler : std_logic_vector(15 DOWNTO 0) := "0000000000000000";
VARIABLE  shift_pos : INTEGER ;

BEGIN
conv_to_int(nshft,shift_pos);
IF ( L_R <= '0') THEN
  FOR i in D'RANGE LOOP
    IF (i > shift_pos-1) THEN
        tmp_out(i) <= D(i-shift_pos);
    ELSE
      tmp_out(i) <= filler(i);
    END IF;
  END LOOP;
ELSIF ( L_R <= '1') THEN
  FOR i in D'RANGE LOOP
        IF (i > 15-shift_pos) THEN
         tmp_out(i) <= filler(i);
        ELSE
          tmp_out(i) <= D(i+shift_pos);
    END IF;
  END LOOP;
END IF;

END shifter;


PROCEDURE rotater (SIGNAL        D : IN std_data;
                   SIGNAL     nshft : IN bit_vector(3 DOWNTO 0);
                   SIGNAL      L_R : IN BIT;
                   SIGNAL tmp_out : OUT std_data) IS
```

```
CONSTANT filler : std_logic_vector(15 DOWNTO 0) := "0000000000000000";
VARIABLE  shift_pos : INTEGER ;

BEGIN
conv_to_int(nshft,shift_pos);
IF ( L_R <= '0') THEN
  FOR i in D'RANGE LOOP
        IF (i > shift_pos-1) THEN    -- Less than or equal to shift_pos
          tmp_out(i) <= D(i-shift_pos);
        ELSE
          tmp_out(i) <= D(16+i-shift_pos);
        END IF;
  END LOOP;
ELSIF ( L_R <= '1') THEN
  FOR i in D'RANGE LOOP
        IF (i < (16-shift_pos)) THEN
          tmp_out(i) <= D(i+shift_pos);
        ELSE
          tmp_out(i) <= D(i+shift_pos-16);  --  D(i-(16-shift_pos))
     END IF;
  END LOOP;
END IF;

END rotater;

BEGIN

barrel : PROCESS (D,L_R,S_R,n_shift)
BEGIN

CASE S_R IS
 WHEN '0' => shifter(D,n_shift,l_r,temp_out);
 WHEN '1' => rotater(D,n_shift,l_r,temp_out);
 WHEN OTHERS => NULL;
END CASE;

END PROCESS barrel;

Y <= temp_out AFTER max_prop_delay;
END behav;
```

A.2 Priority Encoder

```
-- Priority_Encoder.vhd
-- This is to create an 8 input Priority Encoder with cascade facility

LIBRARY ieee, std;

USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY pri_enc IS
PORT(
        I :    IN std_logic_vector(7 DOWNTO 0);
      E_I :    IN BIT;          -- Enable Input, for cascade purposes
      E_O :    OUT BIT;         -- Enable output, for cascade purposes
        A :    OUT bit_vector(2 DOWNTO 0)
);
END pri_enc;

ARCHITECTURE behavior OF pri_enc IS
CONSTANT   max_prop_delay : TIME := 10 ns;
SIGNAL                temp_out : BIT_VECTOR(2 DOWNTO 0);
SIGNAL                 temp_eo : BIT;

BEGIN

temp_out(2) <= '1' WHEN E_I = '1' AND (I(7) = '1' OR I(6) = '1' OR I(5) = '1' OR I(4) =
                    '1') ELSE
               '0';
temp_out(1) <= '1' WHEN E_I = '1' AND (I(7) = '1' OR I(6) = '1' OR I(3) = '1' OR I(2) =
                    '1') ELSE
               '0';
temp_out(0) <= '1' WHEN E_I = '1' AND (I(7) = '1' OR I(5) = '1' OR I(3) = '1' OR I(1) =
                    '1') ELSE
               '0';
temp_eo     <= '1' WHEN E_I = '1' AND I <= "00000000" ELSE
               '0';

A <= temp_out AFTER max_prop_delay;
E_O <= temp_eo AFTER max_prop_delay;
END behavior;
```

A.3 Common Package

```
-- Common_pkg.vhd
-- This is the common package for the barrel shifter

LIBRARY ieee;

USE ieee.std_logic_1164.all;

PACKAGE common_pkg IS

CONSTANT databus_width : NATURAL := 16;
CONSTANT addrbus_width : NATURAL := 16;
CONSTANT  instbus_width : NATURAL := 16;
CONSTANT        shft_size :  NATURAL :=  4;

SUBTYPE  std_data     IS std_logic_vector (databus_width-1 DOWNTO 0);
SUBTYPE  std_address IS std_logic_vector (addrbus_width-1 DOWNTO 0);
SUBTYPE  std_shift     IS std_logic_vector (3 DOWNTO 0);
SUBTYPE  std_inst       IS std_logic_vector (instbus_width-1 DOWNTO 0);

CONSTANT  logic_0 : std_data := (OTHERS => '0');

END common_pkg;
```

REFERENCES

1. Vinogradov, G., "Implementation of Legacy Microprocessors using RISC-based Hardware Emulation Architectural Approach", *Internal Paper for Sarnoff Corporation*, Princeton, NJ.

2. Mohare, P., Huang, J.B., " General Emulation of Microprocessors", *Internal Paper for Sarnoff Corporation*, Princeton, NJ.

3. Smith, J., De Micheli, G., "Automated Composition of Hardware Components", *Design Automation Conference 98,* San Francisco, CA.

4. Hepler, E., *Internal Paper for Sarnoff Corporation,* VLSI Concepts, Malvern, PA.

5. Stone, H.S., Chen, T.C., Flynn, M.J., Fuller, S.H., Lane, W.G., Loomis, H.H., Jr., McKeeman, W.M., Magelby, K.B., Matick, R.E., Whitney, T.M., *Introduction to Computer Architecture*, Science Research Associates Inc., Amherst, MA, 1975.

6. Veronis, A.M., *Survey of Advanced Microprocessors,* New York, Van Nostrand Reinhold, 1991.

7. Tannenbaum, A.S., *Structured Computer Organization*, Englewood Cliffs, NJ, Prentice Hall, Ed. 3, 1994.

8. Carter, J.W., *Microprocessor Architecture and Microprogramming: A State Machine Approach,* Englewood Cliffs, NJ, Prentice Hall, 1995.

9. Weste, N.H.E., Eshraghian, K., *Principles of CMOS VLSI Design,* Burlington, MA, Addison-Wesley, Ed. 2, 1992.

10. Gaonkar, R., *The Z80 Microprocessor: Architecture, Interfacing, Programming and Design,* New York, Macmillan, Ed. 2, 1992.

11. Gaonkar, R., *Microprocessor Architecture, Programming and Applications with the 8085/8080A,* New York, Merrill, Ed. 2, 1989.

12. Tocci, R.J., Laskowski, L.P., *Microprocessors and Microcomputers: The 6800 Family*, Englewood Cliffs, NJ, Prentice Hall, 1986.

13. Hall, D., *Microprocessors and Interfacing; Programming and Hardware*, New Delhi, Tata McGraw-Hill, 1994.

14. Stewart, J.W., *The 8051 Microcontroller*, Englewood Cliffs, NJ, Regents/Prentice Hall, 1993.

15. Lin, W.C., *Microprocessors: Fundamentals and Applications,* New York, IEEE Press, 1977.

16. Alexandridis, N.A., *Microprocessor System Design Concepts*, Rockvillle, MD, Computer Science Press, 1984.

17. Khambata, A.J., *Microprocessors/Microcomputers: Architecture, Software and Systems,* St. Paul, MN, John Wiley & Sons, Ed. 2, 1987.

18. Gibson, G.A., Liu, Y.C., *Microcomputers for Engineers and Scientists,* Englewood Cliffs, NJ, Regents/Prentice Hall, 1980.

19. Hwang, K., *Advanced Computer Architectures,* New York, NY, Mcgraw-Hill, Ed. 1, 1994.