16 - BIT RISC PROCESSOR DESIGN USING VHDL

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

MASTER OF TECHNOLOGY

in

ELECTRICAL ENGINEERING

(With Specialization in System Engineering & Operations Research)

By

PRAVIN SAKHARAM MANE

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2006

ID. NO. MT-313/2006-08/PSM-IG-MKV

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE-247667

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in this dissertation entitled **"16 – BIT RISC PROCESSOR DESIGN USING VHDL"** in the partial fulfillment of the requirements for the award of the degree of **Master of Technology in Electrical Engineering** with specialization in **System Engineering and Operation Research,** submitted in the **Department of Electrical Engineering,** Indian Institute of Technology Roorkee, Roorkee, is an authentic record of my own work carried out from July 2005 to June 2006 under the guidance of **Dr. Indra Gupta,** Assistant Professor and **Prof. M. K. Vasantha,** Professor, Department of Electrical Engineering, IIT Roorkee, Roorkee.

I have not submitted the matter embodied in this report for the award of any other degree or diploma.

[illegible]

Date: 22/06/2006 **(Mr. Pravin Sakharam Mane)**

CERTIFICATE

This is to certify that the above statement made by the candidate is true to the best of my knowledge and belief.

[illegible]

Prof. M. K. Vasantha
Professor
Department of Electrical Engineering
Indian Institute of Technology
Roorkee.

[illegible]

Dr. Indra Gupta
Asstt. Professor
Department of Electrical Engineering
Indian Institute of Technology
Roorkee.

ACKNOWLEDGEMENT

At the outset I express my heartfelt gratitude to Prof. M. K. Vasantha, Professor and Dr. Indra Gupta, Asstt. Professor, Department of Electrical Engineering, Indian Institute of Technology Roorkee, Roorkee for their valuable guidance, support, encouragement and immense help. I consider myself extremely fortunate for getting the opportunity to learn and work under their able supervision. I have deep sense of admiration for their innate goodness and inexhaustible enthusiasm. It helped me to work in right direction to attain desired objectives. Working under their guidance will always remain a cherished experience in my memory and I will adore it throughout my life.

I got the ideas on this topic from the course on online computer application techniques and I consider myself extremely lucky for learning the subject from Prof. M. K. Vasantha. His presentation in the classroom, work and discipline in laboratory, method of documentation and motivation will be what I would like to adhere in my life. His enthusiasm in the topic and encouragement helped me to carry out the work to current extent.

I am extremely thankful to Dr. Indra Gupta for her course on data structures. Before this C was only one of the software languages for me. Through her guidance, I came to know the powerful support the language has got for different data structures used in applications. Most of the companies want professionals expert in this area. Her guidance in the topic of this dissertation I would like to appreciate most.

I also acknowledge Prof. H. O. Gupta for his course on operations research, one of the core subjects of my specialization and course on data management. His practical knowledge and approach towards the subjects helped me to understand the implementation of these topics in real life.

I would like to acknowledge Dr. Rajendra Prasad for his guidance on another core subjects on System Engineering. Optimization techniques and system performance parameter evaluation, improvements were knowledgeful to me. I would also like to acknowledge Prof. A. K. Pant, Dr. G. N. Pillai and Dr. N. P. Padhy for course on modeling, simulation and evolutionary technique for getting familiar with topics like artificial neural network and genetic algorithm which were new to me.

I am thankful to Mr. Vishal Saxena, Mr. Vijander Singh and Ms. Nidhi Singh, research scholars in my department for their constant encouragement throughout the year. I am grateful to Mr. Arun Mohite, Mr. Sanjay Prabhu, Mr. Rajesh Vasnik and Mr. Vaibhav Jain, my colleagues of SEOR group for being excellent peers and creating a congenial environment for work. The facilities offered by the department and its lab technicians Mr. Kalyan Singh and Mr. C. M. Joshi offered flexibilities that deserve praises.

I am extremely thankful to my friends Mr. N. M. Mohite, Mr. Rajesh Shah, Mr. Sayyad M. A., Mr. Sudhir Chopade and Mr. Mahesh Aurade for their moral support throughout my post graduation.

The pleasure of nearing completion of the course requirements is immense, but with it carries the pain of leaving behind these wonderful two years of life in the sprawling green campus of this great historical institute. I proud myself for being the student of this reputed institute.

I thank my wife Mrs. Swati for her help and encouragement in preparing this report. Her support was immense because most of the time I was either in laboratory or in front of computer to work on this topic.

I dedicate this work to my parents for their support and encouragement through out my life.

[illegible]

Pravin S. Mane.

M. Tech. (SEOR)

ABSTRACT

Study over the years showed that simple instruction are used most of the time in CISC processors and many complex instructions can be replaced by group of simple instructions. In that sense RISC processor are designed to execute very few simple instructions. They operate on data, which is mostly present in internal registers. Most of the RISC processors use hardwired control approach, which simplifies design process. External memory is accessed by LOAD and STORE instructions. RISC processor supports only few addressing modes and most of them are register based.

Pipelining is used to improve the throughput of the processor by dividing the instruction execution in stages. Although single instruction takes same time for execution as in sequential execution, parallel operations on instructions in different stages reduces the overall time of execution. The balance of work between different stages of pipelining is important as the slowest stage of the pipeline decides the throughput of the processor. Four-stage pipelining is implemented in this design. The consequences of pipelining are the structural hazards, data hazards and control hazards. They can be handled using the methods of forwarding, stalling and flushing. Stalling degrades the performance by delaying the instruction execution. Prefetching unit is designed which works as a small cache. It is used to prefetch the instructions from memory and store them inside the buffer.

Developed RISC processor handles the hardware interrupts and exceptions. RESET has been assigned the highest priority. Six external hardware interrupts are available and are vectored. Overflow and undefined instruction exceptions are also dealt with.

VHDL is used as software synthesis tool for designing the processor. Xilinx ISE 7.1i is used for this purpose. Hierarchical approach is used for modeling the RISC processor. Basic units are described using behavioral programming and they are interconnected using structural programming to form complete RISC processor. To simulate the different stages of the processor, Xilinx ISE simulator is used. Simulation is used to check the correctness of the design before placing the design for implementation.

Spartan-II FPGA is used to implement the proposed design.

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuits
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CPI	Clock Cycles Per Instruction
Dest.	Destination
DLL	Delay Locked Loop
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
EX	Execution
FPGA	Field Programmable Gate Arrays
Func.	Function
HDL	Hardware Description Language
ID	Instruction Decode
IF	Instruction Fetch
IOB	Input/Output Block
ISA	Instruction Set Architecture
ISE	Integrated Software Environment
LC	Logic Cell
LUT	Look Up Tables
NRE	Non-recurring Engineering Cost
Opcode	Operation Code
PC	Program Counter
PCI	Peripheral Component Interconnect
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computers
ROM	Read Only Memory
RTL	Register Transfer Level

SB	Switch Box
Sorc.	Source
UCF	User Constraints File
VHDL	Very High speed Integrated Circuit Hardware Description Language
WB	Write Back

CONTENTS

Chapter 1
INTRODUCTION

There are two design philosophies in market of microprocessor: Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC). In this introductory chapter we will explain these two trends in brief. Then objectives of work carried out in this dissertation are stated and then the organization of rest of the thesis is presented.

1.1 Complex Instruction Set Computer (CISC)

CISC systems use complex instructions. For example, adding two integers is considered a simple instruction. But, an instruction that copies an element from one array to another and automatically updates both array subscripts is considered a complex instruction. These systems access external memory frequently for data and support various addressing modes. The main purpose was to restrict the size of program so that memory space can be saved. But, they require complex hardware to support complex instruction [1].

The implementation of CISC processor includes microprogrammed control. The conceptual diagram is shown in fig. 1.1. A microprogram is a small run-time interpreter that takes the complex instruction and generates a sequence of simple instructions that can be executed by hardware. This was used to eliminate the semantic gap between high-level language statements and the instructions of processor. Hence most CISC designs use microprogrammed control [1].

Complex instructions are generally variable in length and time if execution depends on specific instruction. The programs written using CISC instructions tend to be smaller in size. The CISC processors were designed to simplify compilers and to improve performance under constraints such as small and slow memories. CISC tends to support a variety of data structures from simple data types such as integers and characters to complex data structures such as records and structures.

CISC designs provide a large number of addressing modes. The main motivations are (i) to support complex data structures and (ii) to provide flexibility to access operands. Some CISC processors like Pentium allow one of the source operands to be in memory. Although this allows flexibility, it also introduces problems. First, it causes variable instruction execution times, depending on location of operands. Second, it leads to variable length instructions. Variable instruction lengths lead to inefficient instruction decoding and scheduling [1].

No doubt, CISC (Complex Instruction Set Computers) has gained the marketplace over the years. The concept of RISC processor is the result of accumulation of knowledge from CISC designs and of course, changing technology.

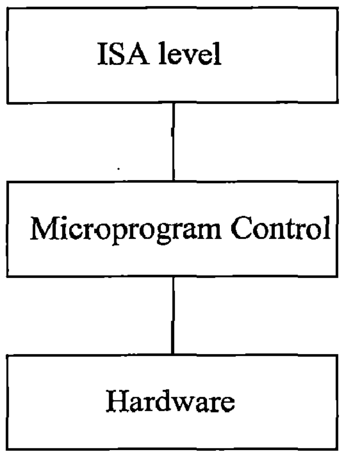


Fig. 1.1 CISC Implementation

1.2 Reduced Instruction Set Computer (RISC)

RISC (Reduced Instruction Set Computer) processors have gained significant attention of designers from last few years because of many features of it. Survey by program analysts over the years has shown that most of the time simple instructions are used and complex instructions are used occasionally. It is also found that many complex instructions can be replaced by group of simple instructions. Thus it is beneficial to design a system that supports a few simple data types efficiently and from which the missing complex data types can be synthesized.

RISC processors have larger register set than CISC processors to avoid the frequent use of external memory. Most of the instructions operate on operand present in internal registers. This aspect improves the performance of the processor.

RISC designs eliminate the microprogram layer and use the hardware to directly execute instructions. That's why they give improved performance [1].

The RISC terminology details are explained in the following chapters.

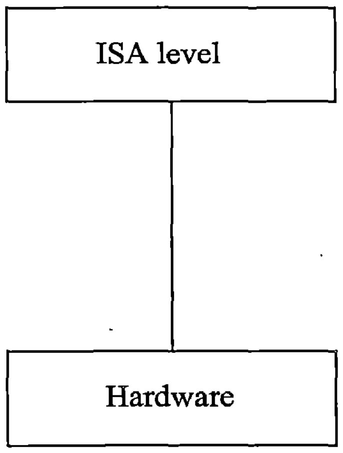


Fig. 1.2 RISC Implementation

1.3 Objectives of the Dissertation

An attempt has been made to design truly 16 – bit RISC processor i.e. instruction size, operand size as well as all data path are 16 bit. The address bus as well as data bus all is 16 bit in length. Effort has been made to implement following features of RISC processor in the proposed design.

- ❖ **Simple Instructions.** The objective is to design simple instructions so that each can execute in one clock cycle. This is possible through pipelined architecture. The advantage of simple instruction is that all operations can be hardwired.
- ❖ **Register-to-Register Operations.** Most of the operations are carried out on the data present in internal registers rather than external memory. This simplifies instruction set design and the structure of control unit.
- ❖ **Simple Addressing Modes.** Simple addressing modes allow fast address computation of operands. Most instructions use register based addressing. Only load and store instructions need a memory-addressing mode.
- ❖ **Large Number of Registers.** For register based operations we need large number of register to optimize the design. But the fixed length of instruction put limit on the registers that can be designed for the processor.
- ❖ **Fixed-Length, Simple Instruction Format**. Variable length instructions can cause implementation and execution inefficiencies. Hence fixed length format is used. The RISC processor also uses simple instruction format where boundaries of various fields in an instruction such as opcode, source

and destination operands are fixed. This allows an efficient decoding and scheduling of instructions.

- **Pipelining.** The method of pipelining is used to speed-up execution. The problems arising due to pipelining are also handled.

VHDL (Very high speed integrated circuit Hardware Description Language) is used as a programming language to implement the proposed processor. It is a special purpose programming language that deals with the design and modeling of digital systems. There are several reasons to choose VHDL to implement design :-

1. Through the use of structural modeling, VHDL can describe how a system is composed of smaller systems and the connections between them.

2. Behavioral modeling allows a system's functionality to be described using common programming language.

1.4 Organization of Thesis

Chapter 2 describes the Instruction Set Architecture (ISA) for proposed RISC processor. The issue of number of operands in instruction is emphasized and then the different instruction formats and datapaths designed for RISC processor are explained.

Chapter 3 is about the addressing modes supported by the RISC processor in this design. All the instructions designed are also explained in this chapter.

Chapter 4 is dealing with concepts of pipelining, its implementation and the handling of hazards arising due to pipelining.

Chapter 5 describes the RISC processor architecture in detail.

Chapter 6 is in support of the software I have used for designing RISC processor. An attempt has been made to implement the design on SPARTAN-II FPGA. Some data about SPARTAN-II FPGA is also given.

Chapter 7 includes the simulation results of designed processor.

Chapter 8 gives the idea about the futures scope in the present design.

Chapter 2

INSTRUCTION SET ARCHITECTURE (ISA) FOR RISC PROCESSOR

While designing any processor, the first thing to do is the decision of the instructions it can operate on, the addressing modes it support and the opcode formats of instructions. In the present design, an attempt has been made to implement true 16 – bit RISC processor. A true 16 – bit RISC processor has 16 – address lines, 16 – data lines, all internal operations are on 16 – bit data and opcode size for every instruction is 16 – bit.

In this chapter some of the important concepts related to instruction set architecture are considered. The issue of number of operands specified in the instruction explicitly, which directly affects the size of opcode of instruction is discussed. The instruction format for various instructions will be given.

2.1 Number of Operands

One of the characteristics that influences the instruction set architecture (ISA) is the number of operands specified explicitly in the instruction. Most operations can be divided into binary and unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, division operation produces two outputs: a quotient and remainder. Since most operations are binary, we need a total of three operands: two to specify input operands and one to specify where the result should go [1].

Most recent processors use three operands. However, it is possible to design systems with two, one, or even zero operands. These four types are discussed in brief here.

2.1.1 Three Operand Machines

In three operand machines, instructions carry all three operands explicitly. This approach is used in the proposed design. For example, if the following operation is to be carried out:

$$A = B + C$$

This can be achieved with single instruction:

ADD A, B, C

The advantage of this approach is that less number of instructions is required for carrying out any task. This saves space in memory. From design point of view it simplifies the task of removing the data hazards [1].

The disadvantage is that it restricts the number of registers designed in a processor.

2.1.2 Two Operand Machines

In two operand machines, one operand doubles as a source and destination. The Pentium is an example processor that uses two operands. To carry out the following operation:

$$A = B + C$$

We require two instructions:

ADD B, C

MOV A, B

The advantage of this approach is that more number of registers can be implemented in fixed instruction length processors as only two registers to be decoded in the instruction [1].

But this is at the expense of memory space. Handling hazards become complicated.

2.1.3 One Operand Machines

In one address machines, one operand is implicit and works as destination as well as one source. This implicit operand is called accumulator. Hence these machines are called accumulator machines. The following operation can be carried out with set of instructions.

$$A = B + C$$

The required instructions are:

LOAD B

ADD C

STORE A

The advantage is more registers can be designed in fixed instruction length format but memory space is wasted. Also handling hazards become more difficult [1].

2.1.4 Zero Operand Machines

In these machines, both source and destination are implicit. These machines make use of stack to carry out operation. For example, to carry out operation:

$$A = B + C$$

It is assumed that B and C are on the top of the stack and A is next to them. The instruction:

ADD

Will fetch B and C from top of stack and after addition result will be stored back on the top of the stack. But storing the values of B and C on the top of stack requires push operation [1].

Three operands method is used for most of the instructions in the proposed design. Some instructions use two operand and single operand method.

2.2 Instruction Formats

The instructions implemented for the proposed design can be categorized into following types because each instruction use different operands and it implement different operation.

2.2.1 Register Format (R - type)

The most common style of instructions is the R – type. It has two read registers and one write register. Fig. 2.1 shows typical R – type instruction format.

15 12	11 9	8 6	5 3	2 0
opcode	Dest.	Sorc.1	Sorc.2	Func.

Fig. 2.1 R – type Instruction Format

2.2.2 Register Immediate Format (RI - type)

The RI – type is similar to the R – type except that second read register and three function bits are replaced by a 6 – bit immediate value. Fig. 2.2 shows typical RI – type instruction format.

15 12	11 9	8 6	5 0
opcode	Dest.	Sorc.1	Immediate Data

Fig. 2.2 RI – type Instruction Format

2.2.3 Immediate Format (I - type)

The I – type instruction has one register and 8 – bit immediate field. The format of typical I – type instruction is shown in fig. 2.3.

15 12	11 9	8 1	0
opcode	Dest.	8 – bit immediate	func

Fig. 2.3 I – type Instruction Format

2.2.4 Shift Format (S - type)

The S – type instruction format has one source register specifying the number to be shifted, second source register to specify the number of bits by which the number is to be shifted and destination register to store the result [3].

15 12	11 9	8 6	5 3	2 0
opcode	Dest.	Sorc.1	Sorc.2	Func.

Fig. 2.4 S – type Instruction Format

2.2.5 Shift Immediate Format (SI - type)

The SI – type format is used by shift instructions. It consists of one destination register, one source register and 5 – bit immediate field. The format of typical SI – type instruction is shown in fig 2.5.

15 12	11 9	8 6	5 1	0
opcode	Dest.	Sorc.1	5 – bit imm.	fun

Fig. 2.5 SI – Type Instruction Format

2.3 Datapath

Datapath show how data flow around processor. Each instruction follow different datapath as it has to access different operands. The paths for each instruction are combined to form overall datapath for processor. The most common datapaths are:

2.3.1 R – Type Datapath

In the R – type datapath the instruction is fetched from memory and broken up into its various parts. The two read registers from the instruction are fetched from the register file and ALU performs the operation given to it by the instruction. The result from ALU is then written back into the register file. The conceptual diagram is shown below.

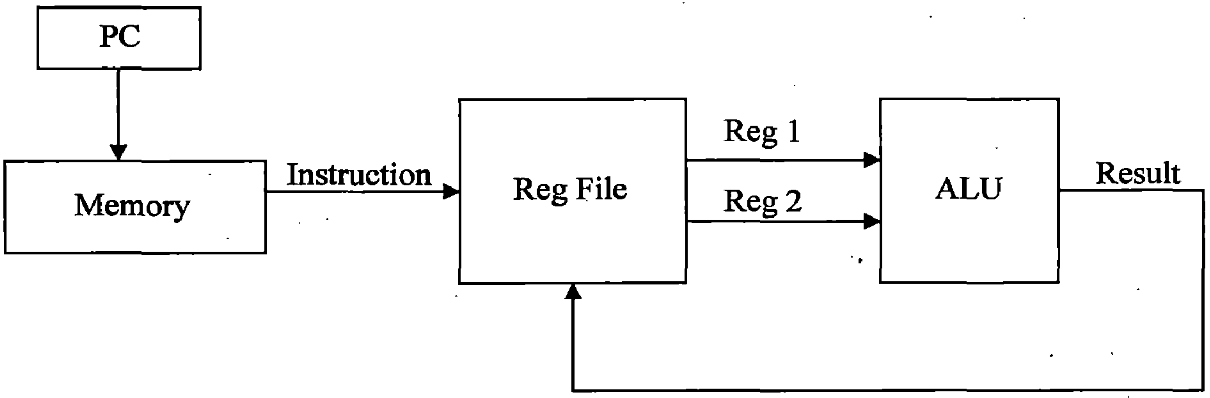


Fig. 2.6 R – Type Datapath

2.3.2 RI – Type Datapath

The RI – type is similar to the R – type except that the second register is replaced with a value that is actually inside the instruction. This immediate value is sign extended to 16 – bit and then use as the second input to the ALU. As with R – type, the result from ALU is then written back into the register file. The conceptual diagram for RI – type datapath is shown below.

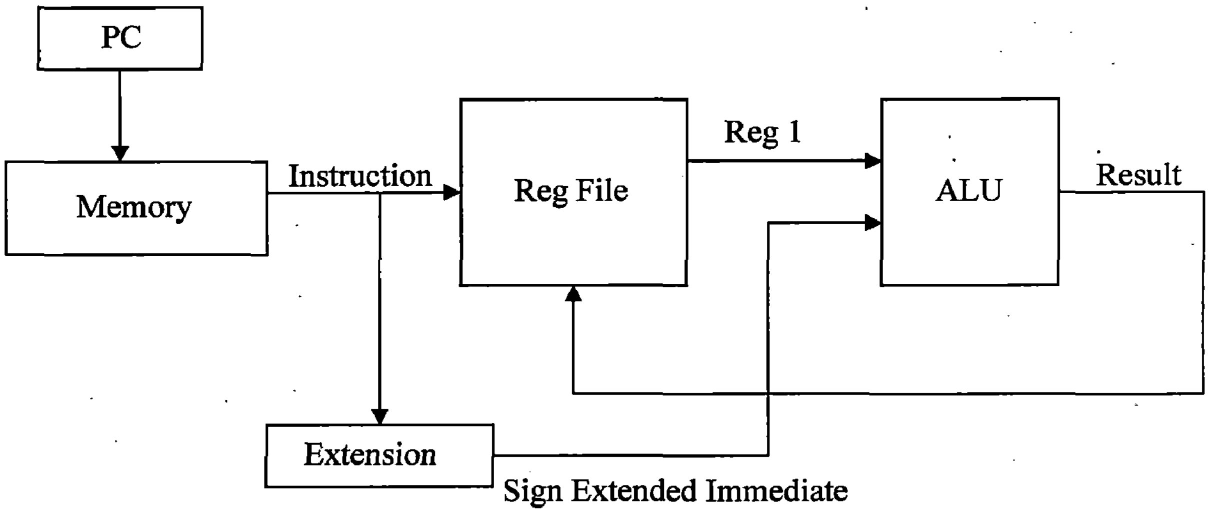


Fig. 2.7 RI – Type Datapath

2.3.3 Load Word Datapath

The datapath for a load word is similar to the RI – type datapath with the exception that result from the ALU is send to fetch a value from memory instead of being written to the register file. The value that is fetched from memory is then loaded into the register file. The conceptual diagram for load word datapath is shown below.

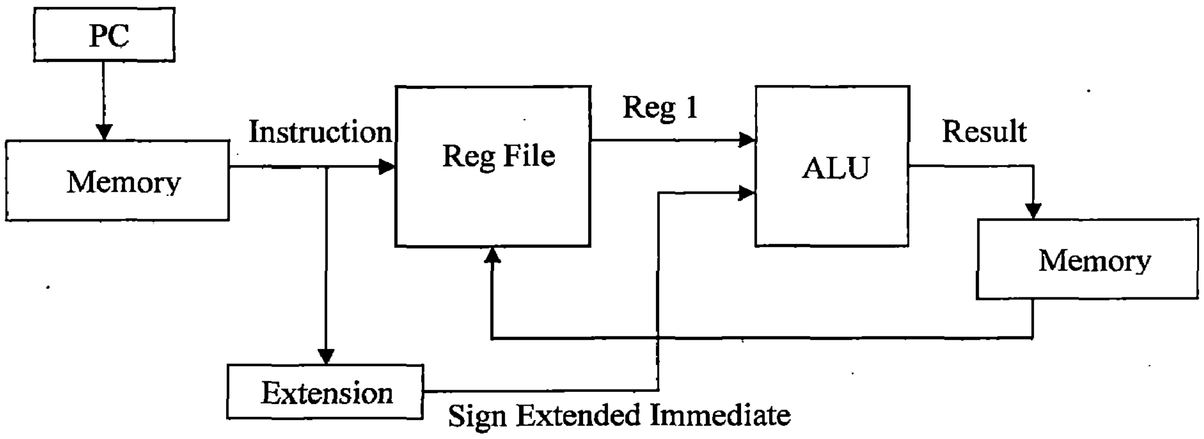


Fig. 2.8 Load Word Datapath

2.3.4 Store Word Datapath

The store word datapath is similar to the load word with the exception that the write register actually specifies which register to write to memory and not the register file. The conceptual diagram for this datapath is shown below.

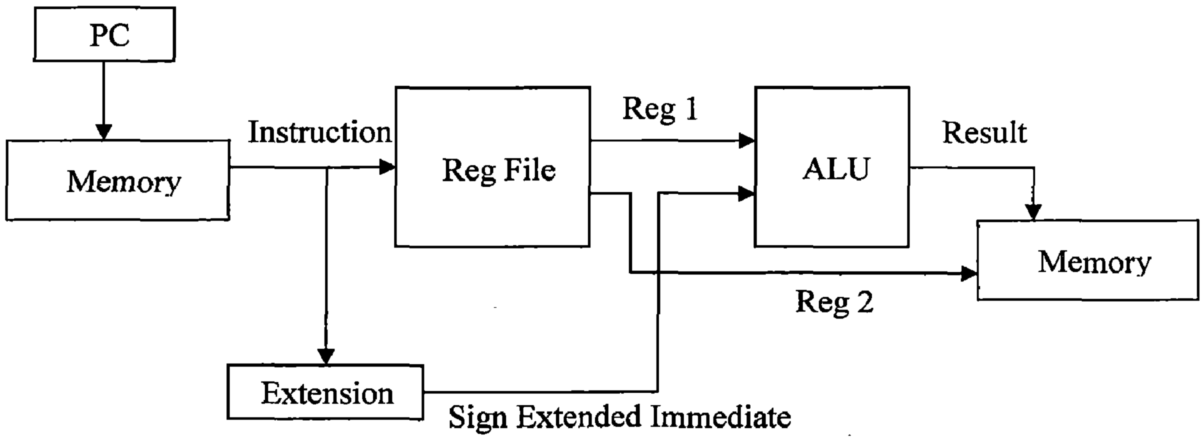


Fig. 2.9 Store Word Datapath

2.3.5 Register Branch Datapath

In the register branch datapath, one register from register file is compared to zero. If the branch type is branch on zero and register is zero then the second register is loaded into the program counter and execution flow continues. A similar think happens with the branch on not zero. The conceptual diagram for this datapath is shown below.

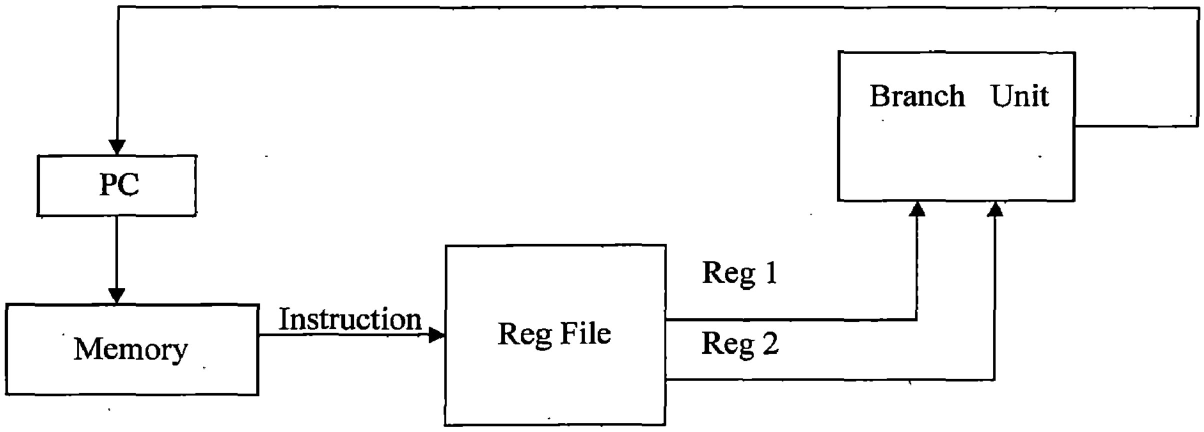


Fig. 2.10 Register Branch Datapath

The data paths explained above are implemented in the design of RISC processor designed in this dissertation.

Chapter 3
ADDRESSING MODES AND INSTRUCTION SET FOR RISC PROCESSOR

As most of the instructions operate on a data which is in internal register, RISC processor supports very few addressing modes. Most instructions use register based addressing. Only load and store instructions need a memory addressing mode. The supported addressing modes are explained in the first section of this chapter. The second section is about the instructions that I have designed for this 16-Bit RISC Processor.

3.1 Addressing Modes

The method of specifying data required for execution of an instruction is called addressing mode. RISC processor support few addressing modes. These are explained in the following section. RISC architecture is sometime called LOAD – STORE architecture since only LOAD and STORE instructions are used to access data from external memory.

3.1.1 Register Operand Addressing Mode

In this addressing mode both source and destination are registers. The instructions supporting register addressing mode are efficient in execution because registers are the part of processor [3]. The examples of this addressing mode are:

ADD R0, R1, R2

AND R6, R1, R5 etc.

3.1.2 Immediate Operand Addressing Mode

In this addressing mode one of the source operand is immediate that is it is specified in the instruction itself. Data is stored along with instruction opcode in program memory. This is generally used when data to be operated on is constant. The examples of this addressing mode are:

ADI R1, R2, 08H

XRI R3, R6, 13H etc.

3.1.3 Register Indirect Addressing Mode

This addressing mode accesses external memory. The address of a memory location to be accessed is specified in one of the register. For example:

LOAD R1, R2

This instruction load register R1 with data accessed from memory location whose address is specified in register R2.

3.1.4 Relative Addressing Mode

This addressing mode is specially used in branching instructions. It adds constant value to the current value of program counter so that branching will take place at a relative address [3]. For example the instruction:

BZI R1, 08

This instruction branches to new location calculated by adding current value of program counter with 08 if R1 contains zero.

All these addressing modes have been used while designing RISC processor.

3.2 Instruction Set

This section describes the instructions designed and implemented for the RISC processor. These instructions can be categorized into following types.

1. Arithmetic Instructions.
2. Logical Instructions.
3. Shift and Rotate Instructions.
4. Data Transfer Instructions.
5. Branching Instructions.
6. Interrupt Related Instructions.
7. Subroutine Related Instructions.
8. Other Instructions.

In the following discussion, each field of opcode is 1-bit in length. RD specifies 1-bit of destination register, RS1 specifies 1-bit of source register2 and so on.

1. Arithmetic Instructions

A. ADD RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 0 1.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type signed 16 – bit addition instruction which adds the content of RS1 and RS2 and stores the result in RD. If the result exceeds 16 – bit then overflow exception is invoked.

B. ADDu RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 0.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type unsigned 16 – bit addition instruction which adds the content of RS1 and RS2 and stores the result in RD. If the result exceeds 16 – bit then overflow exception is invoked.

C. SUB RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 1

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type signed 16 – bit subtraction instruction which subtracts the content of RS2 from RS1 and stores the result in RD. If the result exceeds 16 – bit then overflow exception is invoked.

D. SUBu RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 0 0

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type unsigned 16 – bit subtraction instruction which subtracts the content of RS2 from RS1 and stores the result in RD. If the result exceeds 16 – bit then overflow exception is invoked.

E. ADDI RD, RS1, D6

Type of the instruction : RI – type.

Opcode : 1 0 0 1 RD RD RD RS1 RS1 RS1 D6 D6 D6 D6 D6 D6 .

Where RD is destination register, RS1 is source register and D6 is 6 – bit immediate data.

Description : This is RI – type signed 16 – bit addition instruction which adds the content of RS1 with 6 – bit immediate data and stores the result in RD. If the result exceeds 16 – bit then overflow exception is invoked.

F. SUBI RD, RS1, D6

Type of the instruction : RI – type.

Opcode : 1 0 1 0 RD RD RD RS1 RS1 RS1 D6 D6 D6 D6 D6 D6 .

Where RD is destination register, RS1 is source register and D6 is 6 – bit immediate data.

Description : This is RI – type signed 16 – bit subtraction instruction which subtracts the 6 – bit immediate data from the content of RS1 and stores the result in RD. If the result exceeds 16 – bit then overflow exception is invoked.

2. Logical Instructions

A. NOT RD, RS

Type of the instruction : R – type.

Opcode : 0 0 0 0 RD RD RD RS RS RS X X X 1 1 1.

Where RD is destination register, RS is source register, X don't care.

Description : This is R – type 16 – bit logical NOT instruction which complements the content of RS and stores the result in RD.

B. AND RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 0 0.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type 16 – bit logical AND instruction which logically ANDs the content of RS1 and RS2 bit by bit and stores the result in RD.

C. OR RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 0 1.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type 16 – bit logical OR instruction which logically ORs the content of RS1 and RS2 bit by bit and stores the result in RD.

D. XOR RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 0.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type 16 – bit logical XOR instruction which logically XORs the content of RS1 and RS2 bit by bit and stores the result in RD.

E. NOR RD, RS1, RS2

Type of the instruction : R – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 1.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is R – type 16 – bit logical NOR instruction which logically NORs the content of RS1 and RS2 bit by bit and stores the result in RD.

3. Shift and Rotate Instructions

A. SLL RD, RS1, RS2

Type of the instruction : S – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 0 0.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is S – type 16 – bit logical shift instruction which shifts the content of RS1 left by a value present in RS2 and stores the result in RD. The vacated positions on the right are filled with zeros. The bits coming out of MSB are lost.

B. SRL RD, RS1, RS2

Type of the instruction : S – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 0 1.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is S – type 16 – bit logical shift instruction which shifts the content of RS1 right by a value present in RS2 and stores the result in RD. The vacated positions on the left are filled with zeros. The bits coming out of LSB are lost.

C. SRA RD, RS1, RS2

Type of the instruction : S – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 1 0.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is S – type 16 – bit arithmetic shift instruction which shifts the content of RS1 right by a value present in RS2 and stores the result in RD. The vacated positions on the left are filled with the value of MSB of original data. The bits coming out of LSB are lost.

D. ROR RD, RS1, RS2

Type of the instruction : S – type.

Opcode : 0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 1 1.

Where RD is destination register, RS1 and RS2 are source registers.

Description : This is S – type 16 – bit rotate instruction which rotates the content of RS1 right by a value present in RS2 and stores the result in RD. The bits coming out of LSB are feed back from MSB in the vacated positions.

E. SLLI RD, RS, D5

Type of the instruction : SI – type.

Opcode : 0 1 1 1 RD RD RD RS RS RS D5 D5 D5 D5 D5 0.

Where RD is destination register, RS is source register, D5 is 5- bit immediate data.

Description : This is SI – type 16 – bit logical shift instruction which shifts the content of RS left by a value equal to immediate data and stores the result in RD. The vacated positions on the right are filled with zeros. The bits coming out of MSB are lost.

F. SRLI RD, RS, D5

Type of the instruction : SI – type.

Opcode : 0 1 1 1 RD RD RD RS RS RS D5 D5 D5 D5 D5 1.

Where RD is destination register, RS is source register, D5 is 5 – bit immediate data.

Description : This is SI – type 16 – bit logical shift instruction which shifts the content of RS right by a value equal to immediate data and stores the result in RD. The vacated positions on the left are filled with zeros. The bits coming out of LSB are lost.

G. SRAI RD, RS, D5

Type of the instruction : SI – type.

Opcode : 1 0 0 0 RD RD RD RS RS RS D5 D5 D5 D5 D5 0.

Where RD is destination register, RS is source register, D5 is 5 – bit immediate data.

Description : This is SI – type 16 – bit arithmetic shift instruction which shifts the content of RS right by a value equal to immediate data and stores the result in RD. The vacated positions on the left are filled with the value of MSB of original data. The bits coming out of LSB are lost.

H. RORI RD, RS, D5

Type of the instruction : SI – type.

Opcode : 1 0 0 0 RD RD RD RS RS RS D5 D5 D5 D5 D5 1.

Where RD is destination register, RS is source register, D5 is 5 – bit immediate data.

Description : This is SI – type 16 – bit rotate instruction which rotates the content of RS right by a value equal to immediate data and stores the result in RD. The bits coming out of LSB are feed back from MSB in the vacated positions.

4. Data Transfer Instructions

A. MVIL RD, D8

Type of the instruction : I – type.

Opcode : 0 1 0 0 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 0.

Where RD is destination register, D8 is 8 – bit immediate data.

Description : This is I – type data transfer instruction which loads the lower byte of the destination register with 8 – bit immediate data.

B. MVIH RD, D8

Type of the instruction : I – type.

Opcode : 0 1 0 0 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 1.

Where RD is destination register, D8 is 8 – bit immediate data.

Description : This is I – type data transfer instruction which loads the upper byte of the destination register with 8 – bit immediate data.

C. IN RD

Type of the instruction : R – type.

Opcode : 0 1 0 0 RD RD RD X X X X X X 0 0 0.

Where RD is destination register, X don't care.

Description : This is R – type data transfer instruction which loads the destination register with 16 – bit data available on input port.

D. OUT R

Type of the instruction : R – type.

Opcode : 0 1 0 0 R R R X X X X X X 0 0 1.

Where R is destination register, X don't care.

Description : This is R – type data transfer instruction which sends the content of register over the 16 – bit output port.

E. LW RD, RS, D6

Type of the instruction : RI – type.

Opcode : 0 1 0 0 RD RD RD RS RS RS D6 D6 D6 D6 D6 D6 .

Where RD is destination register, RS is source register, D6 is 6 – bit immediate data.

Description : This is RI – type data transfer instruction. This instruction loads the destination register with the data from memory location whose address is the addition of the contents of RS and 6 – bit immediate data. This is one of the instructions which access external memory.

F. SW RD, RS, D6

Type of the instruction : RI – type.

Opcode : 0 1 0 0 RD RD RD RS RS RS D6 D6 D6 D6 D6 D6 .

Where RD is destination register, RS is source register, D6 is 6 – bit immediate data.

Description : This is RI – type data transfer instruction. This instruction sends the content of destination register to external memory location whose address is the addition of the contents of RS and 6 – bit immediate data. This is the other instruction which accesses external memory.

5. Branching Instructions

A. BZ RD, RS

Type of the instruction : R – type.

Opcode : 0 0 1 0 RD RD RD RS RS RS X X X 0 1 0.

Where RD is destination register, RS is source register, X don't care.

Description : This is R – type conditional branch instruction. If the register RD contains zero value, the contents of RS are copied in program counter and the program execution switches to this new address. If RD doesn't contain zero value, normal execution continues that is next sequential instruction will be executed. This is absolute branching instruction.

B. BNZ RD, RS

Type of the instruction : R – type.

Opcode : 0 0 1 0 RD RD RD RS RS RS X X X 0 1 1.

Where RD is destination register, RS is source register, X don't care.

Description : This is R – type conditional branch instruction. If the register RD contains non-zero value, the contents of RS are copied in program counter and the program execution switches to this new address. If RD contains zero value, normal execution continues that is next sequential instruction will be executed. This is absolute branching instruction.

C. BZI RD, D8

Type of the instruction : I – type.

Opcode : 0 0 1 0 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 0.

Where RD is destination register, D8 is 8 – bit immediate data.

Description : This is I – type conditional branch instruction. If the register RD contains zero value, immediate data in the instruction is added to the current value of program counter and program execution switches to new relative address. If RD doesn't contain zero value, normal execution continues that is next sequential instruction will be executed. This is PC - relative branching instruction.

D. BNZI RD, D8

Type of the instruction : I – type.

Opcode : 0 0 1 0 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 1.

Where RD is destination register, D8 is 8 – bit immediate data.

Description : This is I – type conditional branch instruction. If the register RD contains non-zero value, immediate data in the instruction is added to the current value of program counter and program execution switches to new relative address. If RD contains zero value, normal execution continues that is next sequential instruction will be executed. This is PC - relative branching instruction..

6. Interrupt Related Instructions

A. EI D6

Type of the instruction : I – type.

Opcode : 0 0 1 0 D6 D6 D6 D6 D6 D6 1 0 0.

Where D6 is 6 – bit immediate data.

Description : This is I – type enable interrupt instruction. The RISC processor has six external interrupt lines. They can be enabled or disabled using this instruction. Each bit of D6 is for one of the interrupt line.

D6 (0) – Interrupt line 0; 0 – Disabled, 1 – Enabled.

D6 (1) – Interrupt line 1; 0 – Disabled, 1 – Enabled.

D6 (2) – Interrupt line 2; 0 – Disabled, 1 – Enabled.

D6 (3) – Interrupt line 3; 0 – Disabled, 1 – Enabled.

D6 (4) – Interrupt line 4; 0 – Disabled, 1 – Enabled.

D6 (5) – Interrupt line 5; 0 – Disabled, 1 – Enabled.

B. RETI

Type of the instruction : R – type.

Opcode : 0 0 1 1 X X X X X X X X X 0 1 0.

Where X is don't care.

Description : This is R – type return from interrupt instruction. When executed, it switches back the program execution to the address next to the address where interruption has occurred. This back link address is stored in internal temporary register while switching the execution to interrupt service routine. It is copied in program counter as part of execution of RETI instruction.

7. Subroutine Related Instructions

A. JAL RD, RS

Type of the instruction : R – type.

Opcode : 0 0 1 1 RD RD RD RS RS RS X X X 0 0 0.

Where RD is destination register, RS is source register, X is don't care.

Description : This is R – type jump and link instruction. When executed, it stores the current value of program counter in RD and switches the program execution to the location whose address is in RS. This is subroutine call instruction. The back link address is stored in register RD as part of instruction execution.

B. RJAL RD

Type of the instruction : R – type.

Opcode : 0 0 1 1 RD RD RD X X X X X X 0 0 1.

Where RD is destination register and X is don't care.

Description : This is R – type return from jump and link instruction. When executed, it copies the content of RD into program counter. This back link address in RD was stored as part of execution of jump and link instruction. This instruction is used at the end of subroutine.

8. Other Instructions

A. NOP

Type of the instruction : R – type.

Opcode : 0 0 0 0 X X X X X X X X X 0 0 0.

Where X is don't care.

Description : This is R – type no operation instruction. It does nothing. This instruction can be used to replace unnecessary instructions.

B. HLT

Type of the instruction : R – type.

Opcode : 1 1 0 1 X X X X X X X X X 0 0 0.

Where X is don't care.

Description : This instruction is used to stop the execution of the program.

Chapter 4
PIPELINING AND HAZARDS

To improve the execution speed of processor, instruction execution can be divided into different parts and parallel instruction execution can be done. Although each instruction individually takes equal time execution as in sequential execution, parallel execution improves the throughput of the processor.

First section describes the concept of pipelining and the issues related with it. As a consequence of pipelining, some conflicts arise which are known as hazards. These hazards and their remedies are explained in the next section.

4.1 Basic Concept of Pipelining

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. The key idea behind pipelining is to divide the work into smaller pieces and use assembly line processing to complete the work. In the present design instruction execution has been divided in four stages. In pipeline terminology, each step is called stage because it has a dedicated piece of hardware to perform each step. Different step are completing different parts of different instructions in parallel. The stages are connected one to the next to form a pipe – instructions enter at one end, progress through the stages, and exit at the other end [2].

Pipelining substantially reduces the execution time by overlapping execution of several instructions. In sequential execution, for example, five instructions take 20 clock ticks supposing that each instruction goes through four stages and each stage require one clock tick. On the other hand if same instructions go through pipelined execution, five instructions take only 8 clock ticks. This concept is explained in fig. 3.1 and fig. 3.2. However pipeline requires hardware support [1].

The throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time. The time required between moving an instruction one step down the pipeline is a machine cycle. Because all stages proceed at the same time, the length of the machine cycle is determined by the time required for the slowest pipe stage [4].

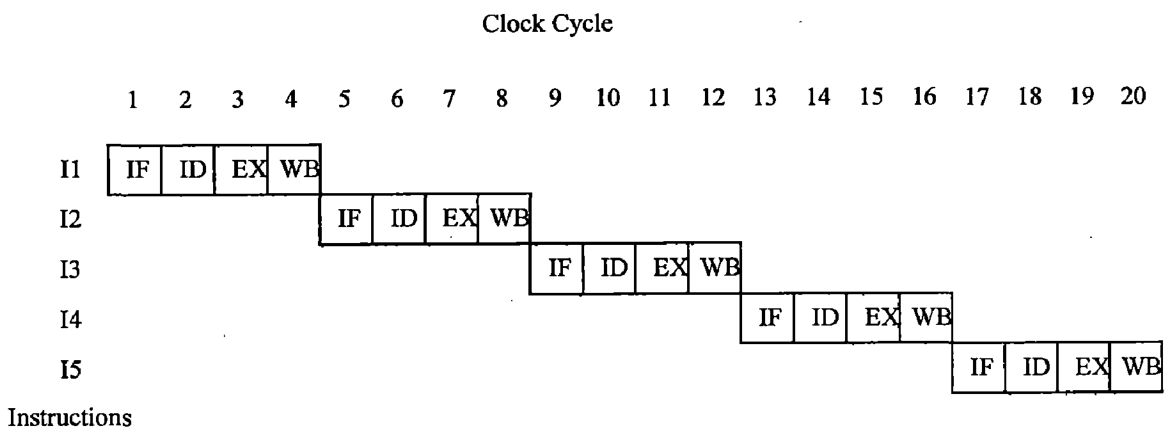


Fig. 4.1 Sequential Execution

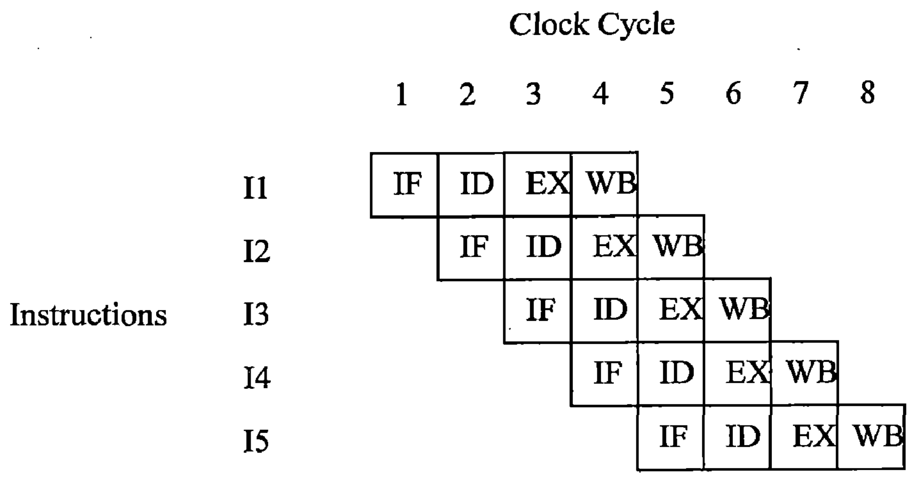


Fig. 4.2 Pipelined Executions

The designer's goal is to balance the length of each pipeline stage. If the stages are perfectly balanced, then the time per instruction on the pipelined machine assuming ideal conditions is equal to

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipelined stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipelined stages. Usually, however, the stages will not be perfectly balanced; furthermore, pipeline does involve some overhead. Thus, the time per instruction on the pipelined machine will not have its minimum possible value, yet it can be close [2].

Pipelining yields a reduction in the average execution time per instruction. Depending on what you consider as the base line, the reduction can be viewed as decreasing the number of clock cycles per instruction (CPI), as decreasing the clock cycle time, or as a combination. If the starting point is a machine that takes multiple clock cycles per instruction, then pipelining is usually viewed as reducing the CPI. This is the primary view we will take. If the starting point is a machine that takes on (long) clock cycle per instruction, then pipelining decreases the clock cycle time [2].

As said earlier, pipelining requires hardware support. For four stage instruction pipeline, we need three buffers. Each of these buffers holds only one value, the output produced by the previous stage. This is possible because pipeline follows just-in-time principle. Just-in-time arrival of input causes problems because any delay in one stage can seriously affect the entire pipeline flow [1].

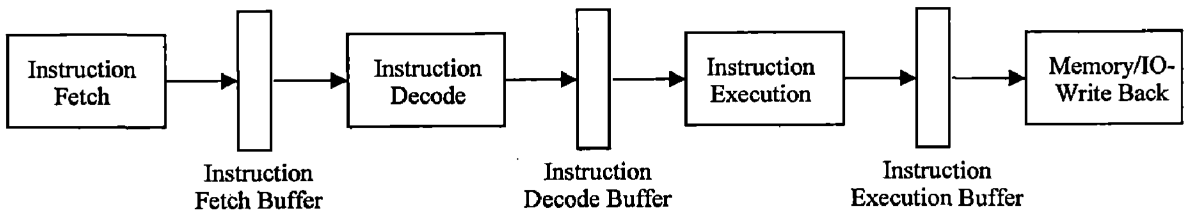


Fig. 4.3 Pipelining Requires Buffering

The fact that the execution time of each instruction does not decrease puts limits on the practical depth of a pipeline. In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead. Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage. Pipeline overhead arises from the combination of pipeline register delay and clock skew. The pipeline registers or latches add setup time plus propagation delay to the clock cycle. Once the clock cycle is as small as the some of the clock skew and latch overhead, no further pipelining is useful since there is no time left in the cycle for useful work [5].

Due to imbalance, one of the stages takes more time for its work to complete. Some stages take variable amount of time for its work. For example, execution stage may require taking data from external memory which may be slow. This causes pipeline stalls.

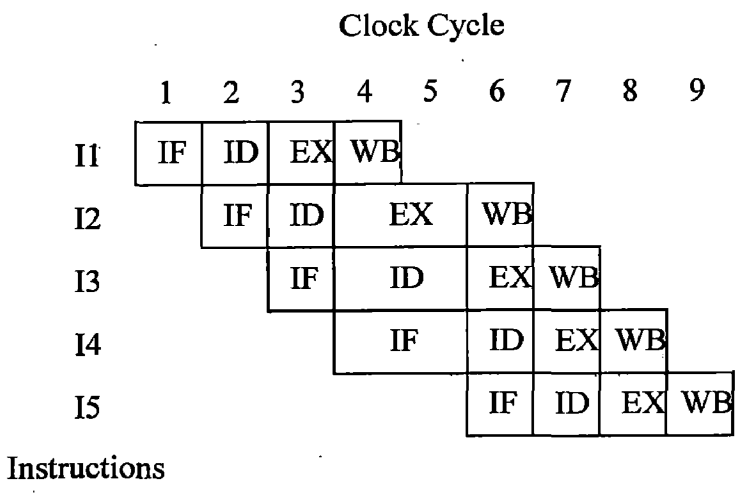


Fig. 4.4 Pipeline Stalls Due to Delay in Stage

4.2 Implementation of Pipelining

In the present design, execution of an instruction is divided into following four stages.

- **Stage 1-Instruction Fetch Stage.** This stage is responsible for fetching the instructions from memory. With the help of prefetcher, it makes use of idle time to fetch instructions ahead of time and store it in instruction queue.
- **Stage 2-Instruction Decode Stage.** This stage separates opcode part, function part and operand part of the instruction and sends it to control unit to generate the necessary signals for units inside the processor for execution of an instruction. This stage also consists of registers.
- **Stage 3-Execution Stage.** This stage consists of arithmetic logic unit, shifting unit necessary for carrying out operations on operand specified by instruction.
- **Stage 4/5-Memory/IO-Write Back Stage.** This stage writes the result back in to the destination register or memory. This is final stage of instruction execution.

4.3 Pipelining Hazards

There are situations, called hazards that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. **Structural Hazards** arise from resource conflicts when the hardware cannot support all possible combinations if instructions in simultaneous overlapped execution.
2. **Data Hazards** arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. **Control Hazards** arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipeline can make it necessary to stall the pipeline. They are more complex to handle. Eliminating a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed [8].

4.3.1 Structural Hazards

When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combinations of instructions can not be accommodated because of resource conflicts, the machine is said to have a structural hazards. For example, the instruction fetch stage under normal conditions will be accessing the memory on every clock cycle. When a load or store instructions is used, the memory/IO-write back stage tries to access the memory. Because of single memory architecture a conflict occurs [9]. There are two ways for dealing with such conflict.

❖ **Stalling.** In this method instead of accessing memory by instruction fetch stage, the load/store instruction is allowed to use memory and the processor is simply stalled until the load/store instruction is finished. The problem with this method is that it can take a long time if there are multiple load/store instructions in a row.

❖ **Prefetching.** Prefetching involves fetching instructions from memory ahead of time and storing them in a queue in a processor. In present design queue has been implemented that can store four instructions. In such case instruction fetch stage will receive instructions from prefetch queue and load/store instruction will be allowed to access memory. Prefetching method is used to handle structural hazards.

4.3.2 Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. Data dependencies can deteriorate performance of a pipeline by causing stall. Data hazards occur when an instruction attempts to use a register whose value depends on the result of previous instruction that have not yet finished. For example consider the following case.

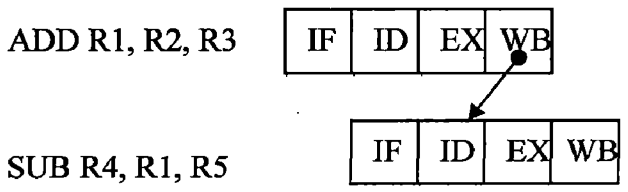


Fig. 4.5 Example of a Data Hazard

In this case result produced by ADD instruction is used as source for SUB instruction. Hence until result of first instruction is not written back in destination register, next instruction can not proceed and a stall will be introduced [7].

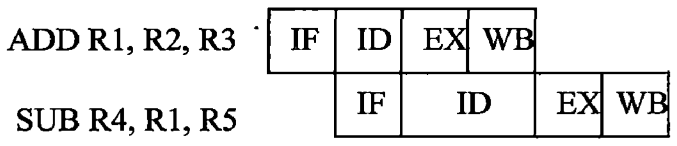


Fig. 4.6 A Stall Caused By Data Hazards

There are two techniques to handle data hazards.

- ❖ **Register Forwarding.** This technique, also called bypassing, works if the two instructions involved in the dependency are in the pipeline. The basic idea is to provide the output result as soon as it is available in the datapath. The forwarding method is best described through the use of an example. Fig. 4.7 shows two instructions in the pipeline, it can be observed that the SUB instruction needs the result of the ADD instruction in the SUB's EX stage but the ADD instruction does not write the result until the ADD's WB stage. However it can also be seen that the result for the ADD instruction is actually computed before the SUB instruction needs it so the result is forwarded from EX stage of ADD instruction to the EX stage of the SUB instruction [1].

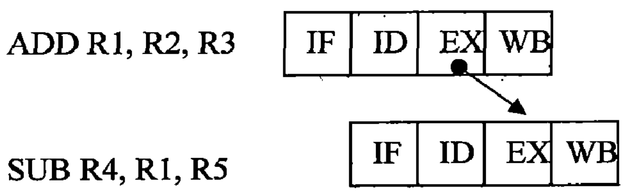


Fig. 4.7 Register Forwarding

❖ **Register Interlocking.** This is a general technique to solve the correctness problem associated with data dependencies. In this method, a bit is associated with each register to specify whether the contents are correct. If the bit is 0, the contents of the register can be used. Instruction should not read contents of register when this interlocking bit is 1, as the register is locked by another instruction. Fig. 4.8 shows how the register interlocking works for ADD SUB instructions given above. ADD instruction locks R1 until result is not written in it. Hence SUB instruction can not use it as far as it is locked by ADD instruction [1].

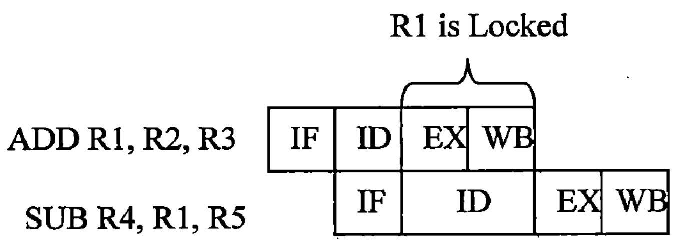


Fig. 4.8 Register Interlocking

4.3.3 Control Hazards

A control hazard occurs whenever there is a change in the normal execution flow of the program. Events such as branches, interrupts, executions and return from interrupts. A hazard occurs because branches, interrupts etc are not caught until the instruction is decoded in the second stage, by the time it is decoded the following instruction is already entered into the pipeline and left unchecked an unwanted instruction would remain in the pipeline. There is really only one solution to this type of hazard. That is, to implement a hardware stall. The hardware stall simply flushes the offending instruction from the pipeline [2].

Chapter 5
RISC PROCESSOR ARCHITECTURE

This chapter describes the architecture of developed 16-bit RISC processor. It is true 16-bit RISC processor as the address bus is 16-bit in length, data bus is 16-bit in length, all registers are 16-bit wide and all operations are carried out on 16-bit data. The first section gives the overview of RISC processor architecture. Second section gives the details of pipelined stages of processor. Then next sections describe control unit, branch forwarding unit, execution forwarding unit, prefetch unit, hazard detection unit and interrupt and exception unit in detail.

5.1 Overview of RISC Processor Architecture

To improve the throughput of processor, the pipelined architecture is used. Execution of the instruction is divided into four stages viz.

1. Stage1- Instruction Fetch Stage.
2. Stage2- Instruction Decode Stage.
3. Stage3- Instruction Execution Stage.
4. Stage4- Memory/IO-Write Back Stage.

Every instruction proceeds to the next stage in each clock cycle and new instruction enters the instruction fetch stage. Each individual instruction takes four clock cycles for complete execution after entering the instruction fetch stage provided that no stalls occur.

Control unit generate necessary signal at appropriate time for all the stages for instruction execution. Branch forwarding unit flushes the instructions behind branching instruction in the pipeline if branching is going to occur. Execution forwarding unit takes care of data hazard by forwarding the result of previous instruction to execution stage if data hazard occurs. Prefetching unit prefetches instruction from memory when processor is not utilizing external memory. It stores the prefetched instructions in prefetch queue, which is four words deep. Hazard detection unit is used to detect whether conflicts are going to occur and generate the necessary signals for other units [10]. Interrupt and exception unit handles the external interrupts and exceptions generated internal to the processor. The fig.5.1 shows the architecture of RISC processor implemented in the design.

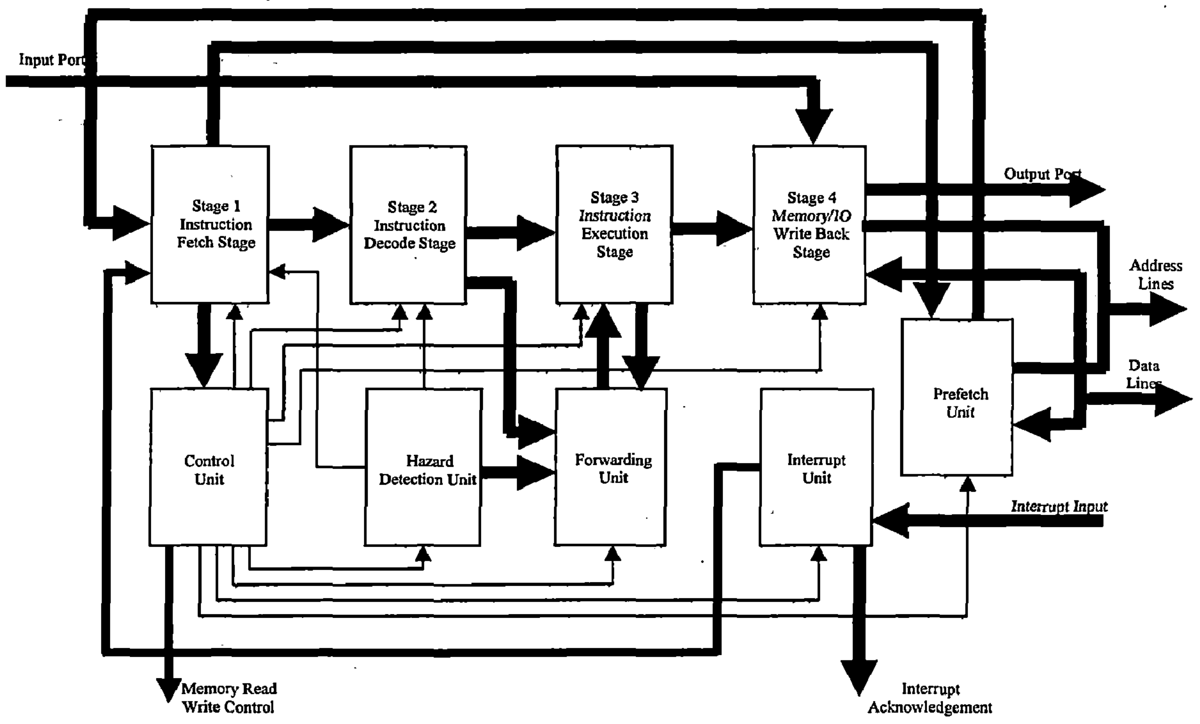

Input Port
Stage 1 Instruction Fetch Stage
Stage 2 Instruction Decode Stage
Stage 3 Instruction Execution Stage
Stage 4 Memory/IO Write Back Stage
Output Port
Address Lines
Data Lines
Prefetch Unit
Control Unit
Hazard Detection Unit
Forwarding Unit
Interrupt Unit
Interrupt Input
Memory Read Write Control
Interrupt Acknowledgement

Fig. 5.1 RISC Processor Block Diagram

5.2 Pipelined Stages of RISC Processor

Four stages of the RISC processor pipeline are described below.

5.2.1 Stage1-Instruction Fetch Stage

This stage consists of program counter, program counter incrementer and selector, which select the new value of program counter. This unit is responsible for obtaining the instruction from memory through prefetcher. The block diagram of instruction fetch stage is shown in fig. 5.2.

The various components of this stage and their functions are described in the following sections.

❖ **Program Counter Selector.** This selector is used to decide the execution sequence of program. Under normal sequential execution, the new program counter value is the previous program counter value incremented by one. For branch instruction the new program counter value will be the branch target address. RISC processor supports vectored hardware interrupts and RESET. The interrupt and their vectored address are given in table 5.1. Return from interrupt is the address next to location where interrupt had occurred. Return from subroutine is the address next to JAL instruction. Control unit generates the select input for program counter selector.

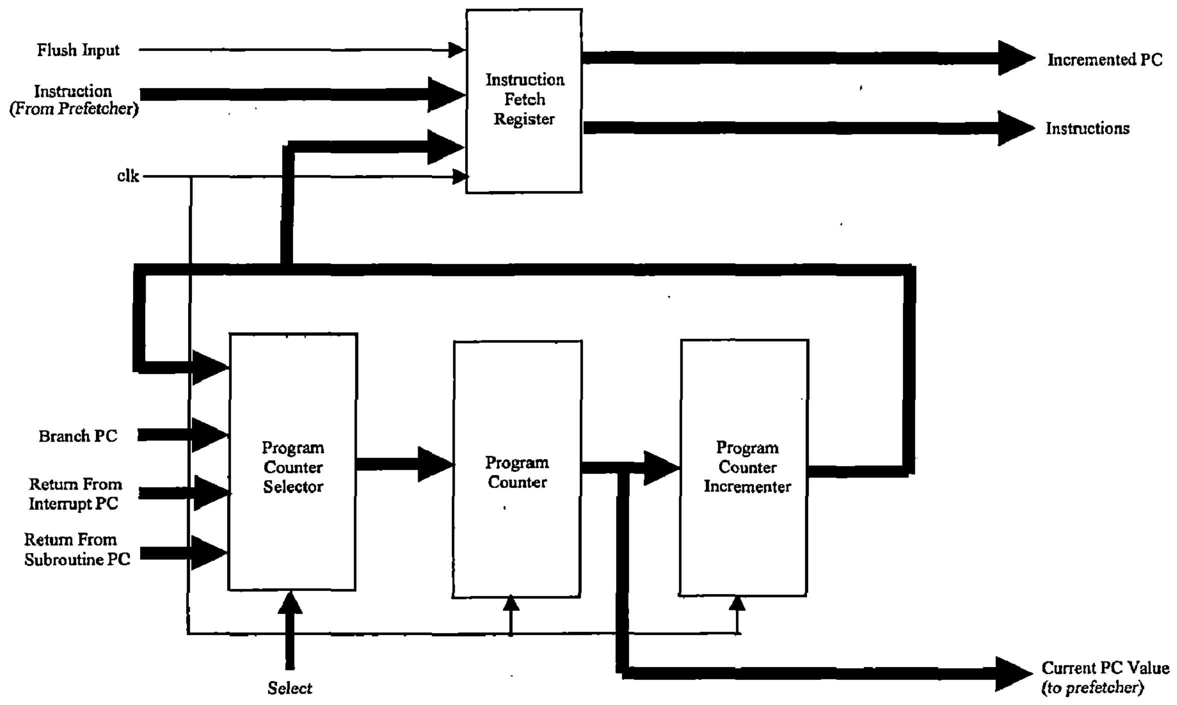

Flush Input
Instruction (From Prefetcher)
clk
Instruction Fetch Register
Incremented PC
Instructions
Branch PC
Return From Interrupt PC
Return From Subroutine PC
Program Counter Selector
Program Counter
Program Counter Incrementer
Select
Current PC Value (to prefetcher)

Fig. 5.2 Stage1-Instructions Fetch Stage

Select Input	New PC Value	Remark
0000B	Incremented Old PC.	Sequential Executions.
0001B	Branch Target Address.	Branch Instruction Execution.
0010B	Address Next to JAL Instruction.	Return From Subroutine.
0011B	Address Next to Interruption.	Return From Interrupt Routine.
0100B	FFFFH	Overflow.
0101B	FFF0H	Undefined Instruction.
0110B	0008H	Interrupt 0.
0111B	000AH	Interrupt 1.
1000B	000CH	Interrupt 2.
1001B	000EH	Interrupt 3.
1010B	0010H	Interrupt 4.
1011B	FFF8H	Interrupt 5.
Others	0012H	RESET.

Table 5.1 New PC Value Selections

- **Program Counter.** It is the register which holds the address of instruction which is to be executed. The content of it are send to prefetcher for fetching the instruction from memory.
- **Program Counter Incrementer.** This incrementer increments the program counter by one for normal sequential execution of the program.
- **Instruction Fetch Stage Register.** Every stage in the pipeline has its own buffer. This buffer is used to store the output of respective stage. Instruction fetch stage register stores the instruction coming from prefetcher and the incremented program counter value and supply it to the next stage. Flush input is used to flush this register for non sequential execution to handle the control hazards.

5.2.2 Stage2-Instruction Decode Stage

This stage consists of register file, sign extension unit; branch unit, multiplexers and instruction decode stage register. The block diagram of this stage is shown in fig.5.3.

- **Register File.** Register file consist of 8 registers. Each register is of 16-bit in length. They are designated as R0, R1... R7. These registers are decoded in the instruction as shown in following table. The source register codes are given as input to this file and it gives content of the respective registers as output. The write-back register code and write-back data are the other inputs from the write-back stage to this file. If write enable input is enabled, then data is written into the respective register. If write-back register and source register are same data is written provided that write enable input is high as well as same data is output as content of that register.
- **Sign Extension Unit.** All the operations carried out inside the processor are on 16-bit data. Hence 5-bit, 6-bit and 8-bit immediate data decoded in the instruction is sign extended to 16-bit by this unit. The output of this unit is used in all types of instructions having immediate data as one of the operand and in relative branch instructions.
- **Branch Unit.** This unit is used to calculate the branch target address. It is used to differentiate between relative branching and register indirect branching. This function is explained in the following table.

- **Multiplexers.** They are used to select the source registers for instruction. Some instructions have one source register and others have two source registers. Appropriate selection is done by set of multiplexers.
- **Instruction Decode Stage Register.** This is used to store the output produced by instruction decode stage. It stores mainly content of source register 1 and source register 2, sign extended immediate data.

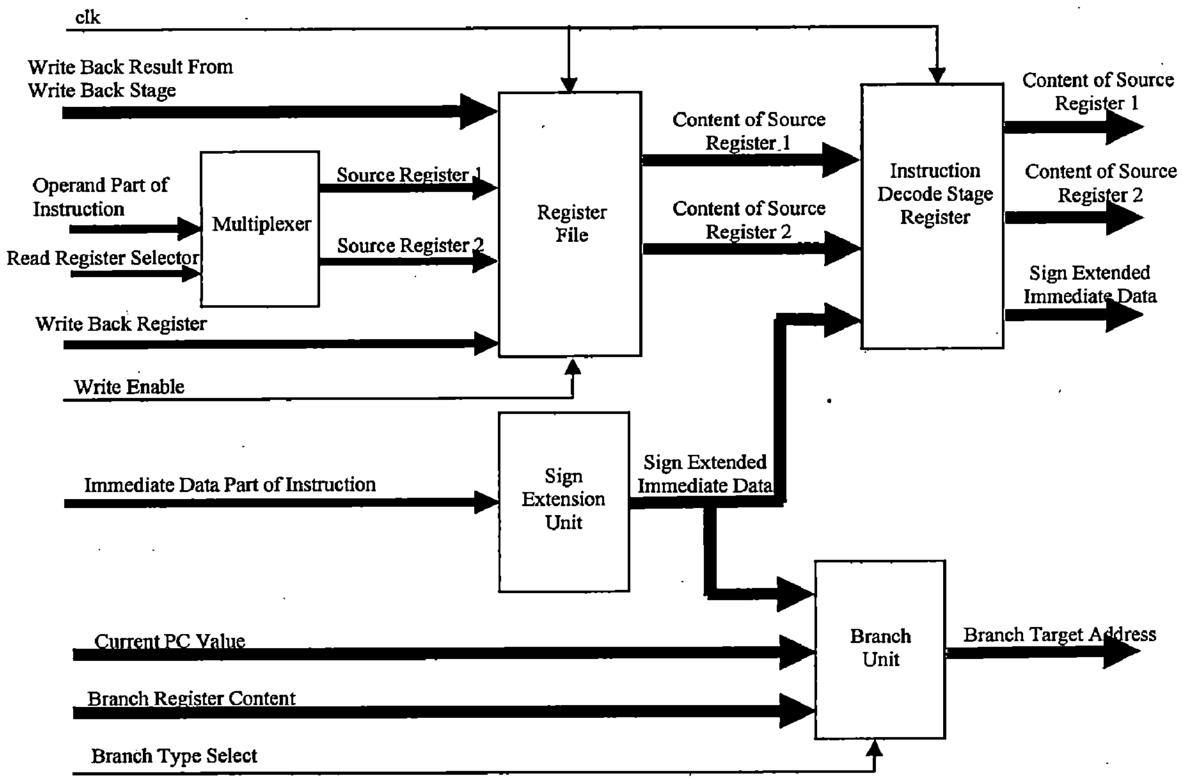

clk
Write Back Result From Write Back Stage
Operand Part of Instruction
Read Register Selector
Multiplexer
Source Register 1
Source Register 2
Write Back Register
Write Enable
Register File
Content of Source Register 1
Content of Source Register 2
Instruction Decode Stage Register
Content of Source Register 1
Content of Source Register 2
Sign Extended Immediate Data
Immediate Data Part of Instruction
Sign Extension Unit
Sign Extended Immediate Data
Current PC Value
Branch Register Content
Branch Type Select
Branch Unit
Branch Target Address

Fig. 5.3 Stage2-Instruction Decode Stage

Register	Code
R0	000B
R1	001B
R2	010B
R3	011B
R4	100B
R5	101B
R6	110B
R7	111B

Table 5.2 Register Codes to be Used in Instruction

Branch Type Select Input	Target Branch Address	Branch Type
0B	Contents of register in branch instruction.	Register Indirect Branching.
1B	Current PC + sign extended offset.	Relative Branching.

Table 5.3 Branch Type Differentiation

5.2.3 Stgage3-Instruction Execution Stage

This stage carries out the operation specified by the instruction on the operands and produces the required result. This stage consist of arithmetic logic unit (ALU), instruction execution stage register and multiplexer.

Arithmetic logic unit carries out arithmetical operations like addition and subtraction, logical operations like AND, OR, XOR, NOR and NOT, arithmetical and logical shifts.

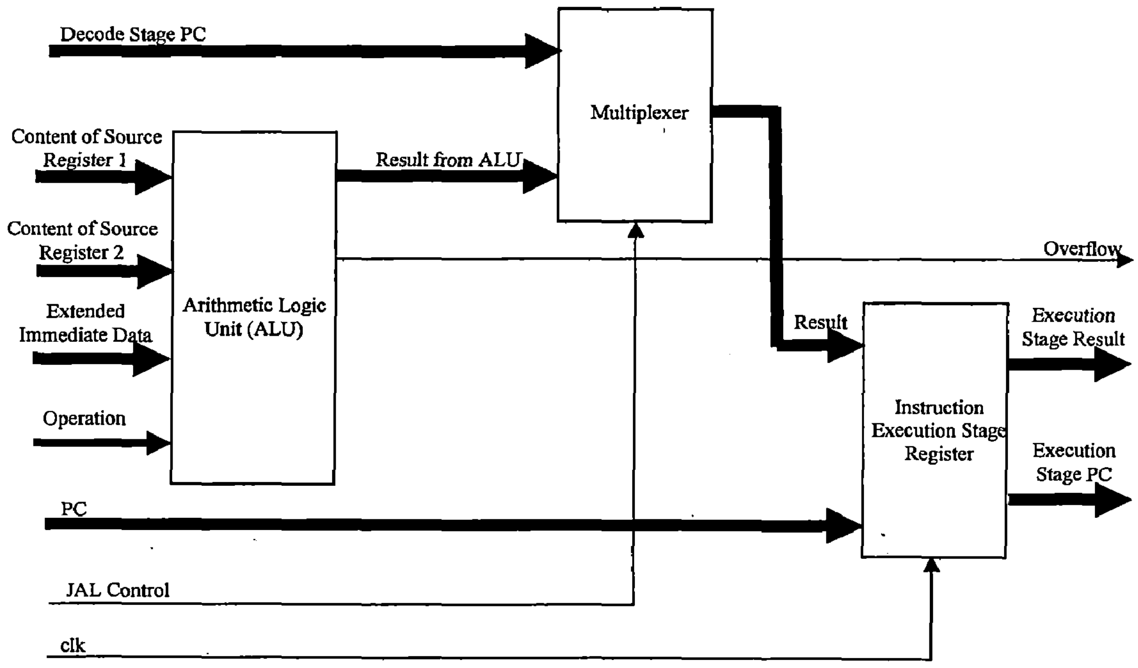


Fig. 5.4 Stage3-Instruction Execution Stage

❖ **Arithmetic Logic Unit.** This unit is composed of basic ALU, shift unit, move immediate unit, ALU multiplexer and selector. The block diagram of arithmetic logic unit is shown in the fig. 5.5.

- **Basic ALU.** This unit does the arithmetic operations like addition and subtraction, logical operations like AND, OR, NAND, NOR, XOR and NOT. In such case both source operand may be registers or one is register and other is immediate data. In case of arithmetical operations, overflow flag is appropriately set.
- **Shift Unit.** This unit shifts the data present in source1 by number of bits equal to contents of source2. Shift operation may be arithmetical or logical and it may be in right or left direction. Source2 may be register or an immediate value [9]. Shift operation specifies the direction as well as type of shift as in table 5.4. Shift by 8, shift by 4, shift by 2 and shift by 1 unit are connected in series and does the specified operation only if they are enabled. Otherwise they pass the input data as it is at the output.
- **Move Immediate Unit.** This unit loads the upper or lower byte of the specified register with 8-bit immediate data keeping the other byte as it is.
- **ALU Multiplexer.** This is used to select between basic ALU result, shift unit result and move immediate unit result as final result of arithmetic logic unit.
- **Selector.** This is used to select the source2 from either source register or immediate data.

Operation Input	Type and Direction of Shift
00B	Logical Shift Left.
01B	Logical Shift Right.
10B	Arithmetic Shift Right.
11B	Arithmetic Shift Left.

Table 5.4 Type and Direction of Shift Combinations

- **Instruction Execution Stage Register.** This register is used to store the result of instruction execution stage. It mainly stores the result of arithmetic logic unit and instruction execution stage PC value [11]. The output from this stage is fed to stage4-Memory/IO-write back stage.

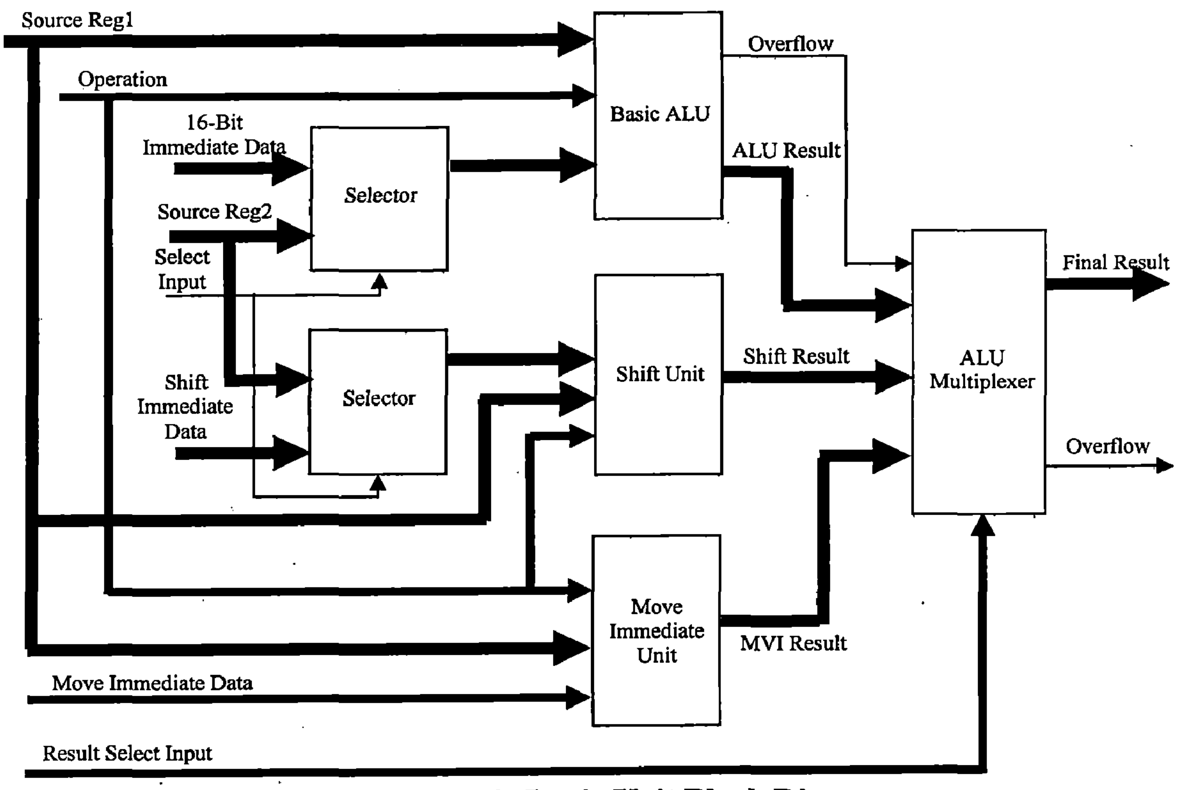


Fig. 5.5 Arithmetic Logic Unit Block Diagram

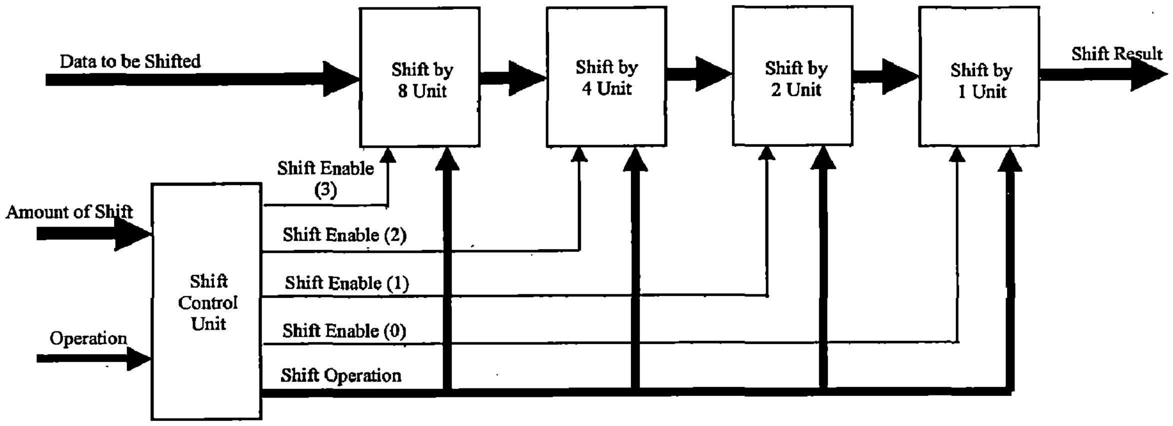


Fig. 5.6 Block Diagram of Shift Unit

- **Multiplexer.** It is used to select between arithmetic logic unit result and instruction decode stage PC value required for JAL instruction as final result.

5.2.4 Stage4-Memory/IO-Write Back Stage

This stage consists of output port register, memory/IO-write back pipeline register and result selector. The block diagram of this stage is shown in fig. 5.7.

- **Output Port Register.** This register comes into action for OUT instruction. The content of the register specified in the instruction is placed on output port.
- **Memory/IO-Write Back Pipeline Register.** This register stores the data coming from input port, the data coming from memory for store instruction and the result produced by stage3. This register also store code for write back register.
- **Result Selector.** Based on the instruction, this selector assigns one of the data from memory/IO-write back pipeline register as final write back data.

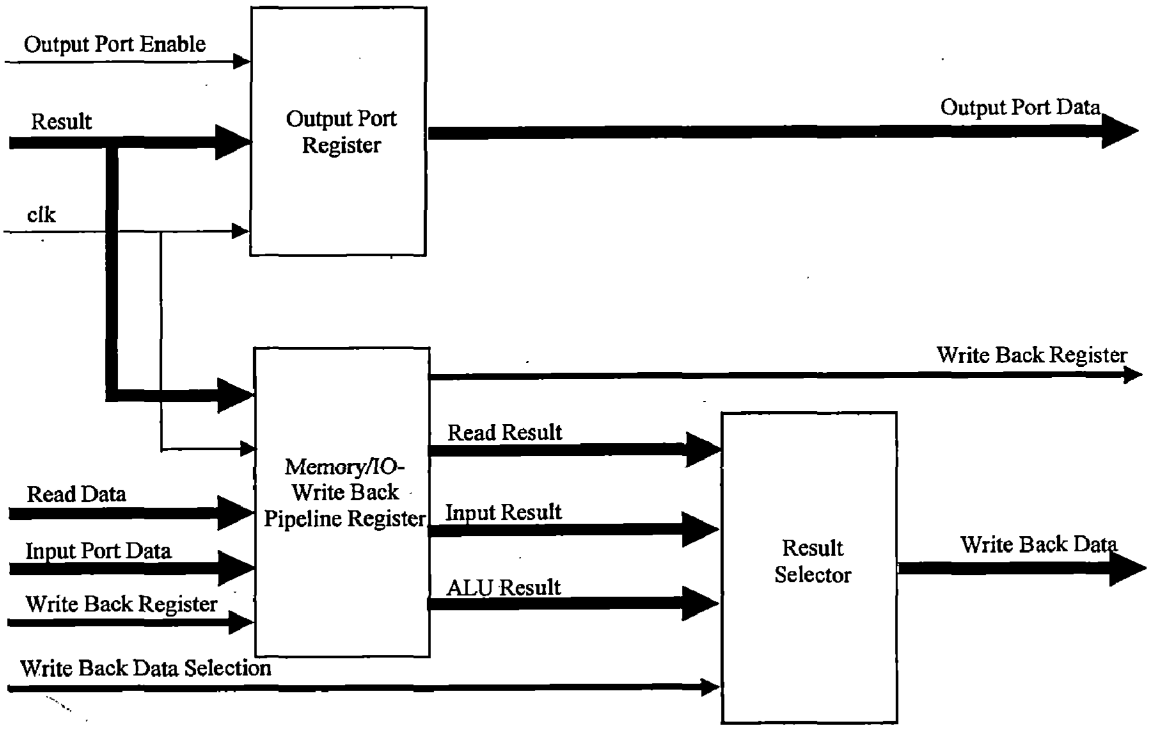


Fig. 5.7 Stage4-Memory/IO-Write Back Stage

5.3 Control Unit

This unit receives its input from stage1 and produce necessary control signals for all units of the processor for execution of an instruction. It also maintains the necessary sequence in generating control signals for stages of the pipeline. Hardwired control is used in the control unit [13]. This unit is made up of basic control unit, decode control register, execution control register and write back control register. Block diagram of it is shown in fig. 5.8.

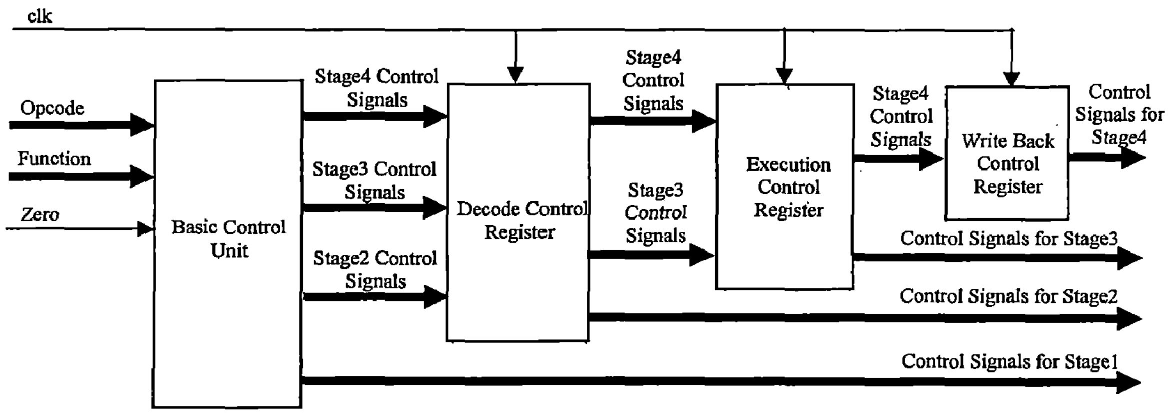


Fig. 5.8 Block Diagram of Control Unit

- **Basic Control Unit.** This unit generates all control signals for processor based on combinations of opcode, function and the value of zero line (Only Effective for Conditional Branch Instruction). These controls signals are passed out of control unit in sequence using three registers in the control unit. Some control signals from basic unit are used directly for stage1 of pipeline. Others are passed to decode control register.
- **Decode Control Register.** After receiving control signals from basic control unit, this register sends them on the next clock pulse. Some of them are used directly to control stage2 of pipeline. Others are passed to execution control register.
- **Execution Control Register.** Control signals received from decode control register are passed out on the next clock pulse. Some of them are used for controlling the stage3 and remaining is passed to write back control register.
- **Write Back Control Register.** Control signals received from execution control register are passed out on the next clock pulse to control stage4 of pipeline.

Control signals necessary for other unit of the processor are taken out from the above combination at appropriate time.

5.4 Branch Forwarding Unit

This unit is used to decide the branch target address and to create the condition necessary for conditional branch instruction. It also tries to resolve the conflict arising in pipeline due to branching [12]. It consists of branch forwarding detection unit, zero detector and selectors.

❖ **Branch Forwarding Detection Unit.** This unit decides the condition data and branch target address component for conditional branch instruction execution. It may be possible that the destination of previous instruction may be the condition data source or target address component source. In such case this unit selects the appropriate values for these two components.

❖ **Selectors.** One selector is used for selecting the condition data component and the other is used for selecting branch target address component from the source specified in instruction, execution stage result and write back stage result.

❖ **Zero Detector.** This unit checks whether the data at its input is zero in value or not. If zero the output line is set high otherwise low. This is used in instructions like branch on zero, branch on not zero etc.

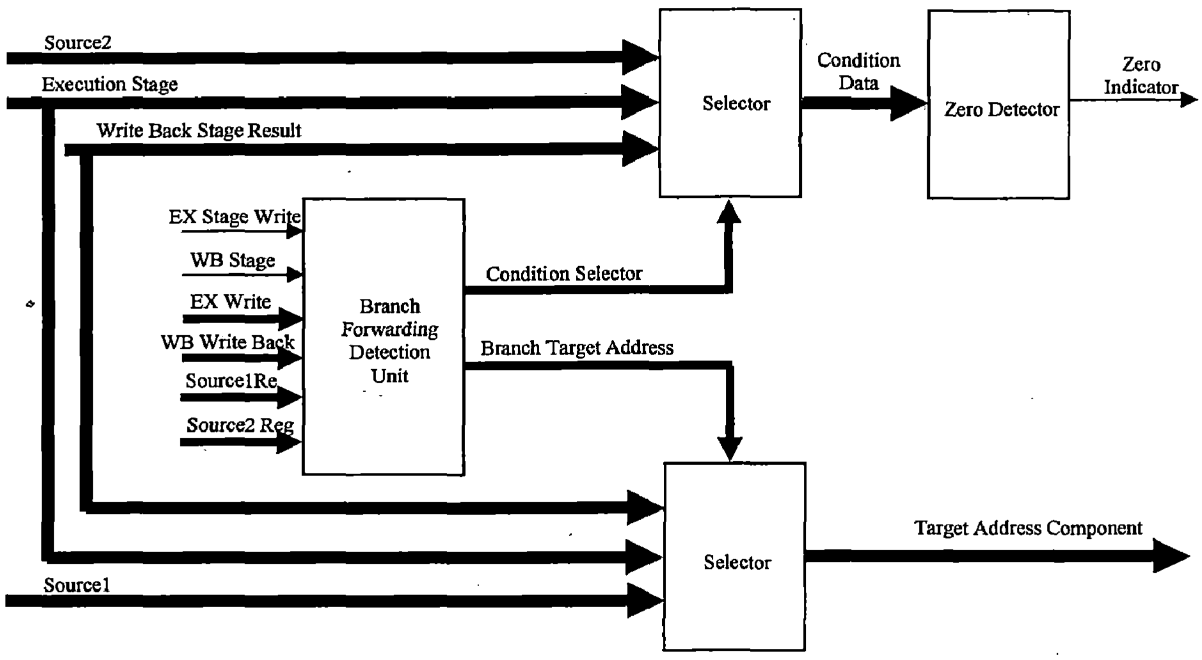


Fig. 5.9 Branch Forwarding Unit Block Diagram

5.5 Execution Forwarding Unit

This unit is used to eliminate the resource conflicts occurring in pipelining. There are certain conditions that occur in the pipeline cause resource conflicts. This unit consists of execution stage forwarding detection unit and selectors.

- **Execution Stage Forwarding Detection Unit.** This unit creates the select inputs for two selectors present in this stage. If the source register of current instruction is destination of previous instruction then the source for current instruction is not valid until the result of previous instruction is written back. This requires stalls [1]. Other method is to forward the execution stage result of previous instruction directly as the source for current instruction. This method is used here.
- **Selectors.** One selector is used for selection of source1 and other is used for selection of source2 from the source specified in the instruction, execution stage result and write back stage result. The select input for these selectors is generated by execution stage forwarding detection unit based on various conditions.

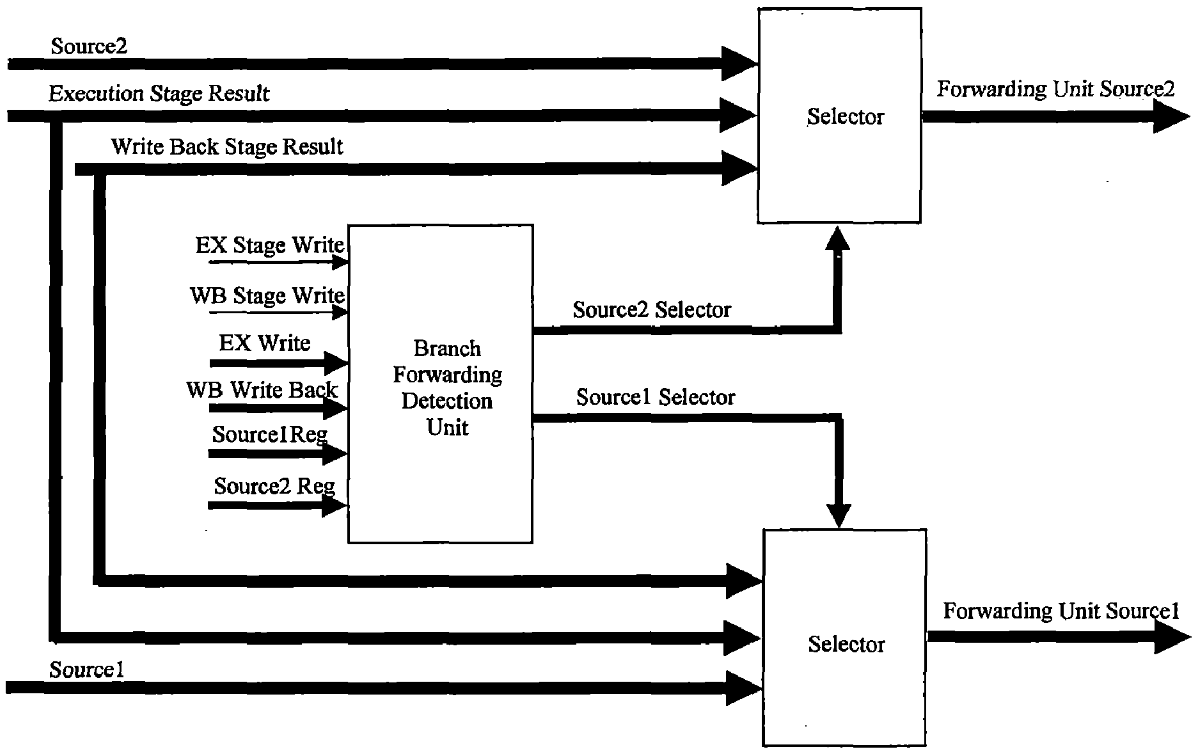


Fig. 5.10 Execution Forwarding Unit Block Diagram

5.6 Hazard Detection Unit

This unit takes care of different hazards occurring in the pipeline of processor. Based on various conditions of opcode, function, source register and control signals generated by control unit, it produces three control signals. One is used to decide whether program counter is to be updated or not and thus take the decision on whether to initiate stalls or not. For handling the control hazards it is necessary to flush the pipeline. Remaining two control signals from this unit are used for this purpose. One is used to flush instruction fetch stage register and the other is used to flush instruction decode stage register [2].

5.7 Interrupt and Exception Unit

This unit is responsible for handling hardware interrupts and exceptions. This processor supports six hardware interrupts, one RESET and overflow exception.

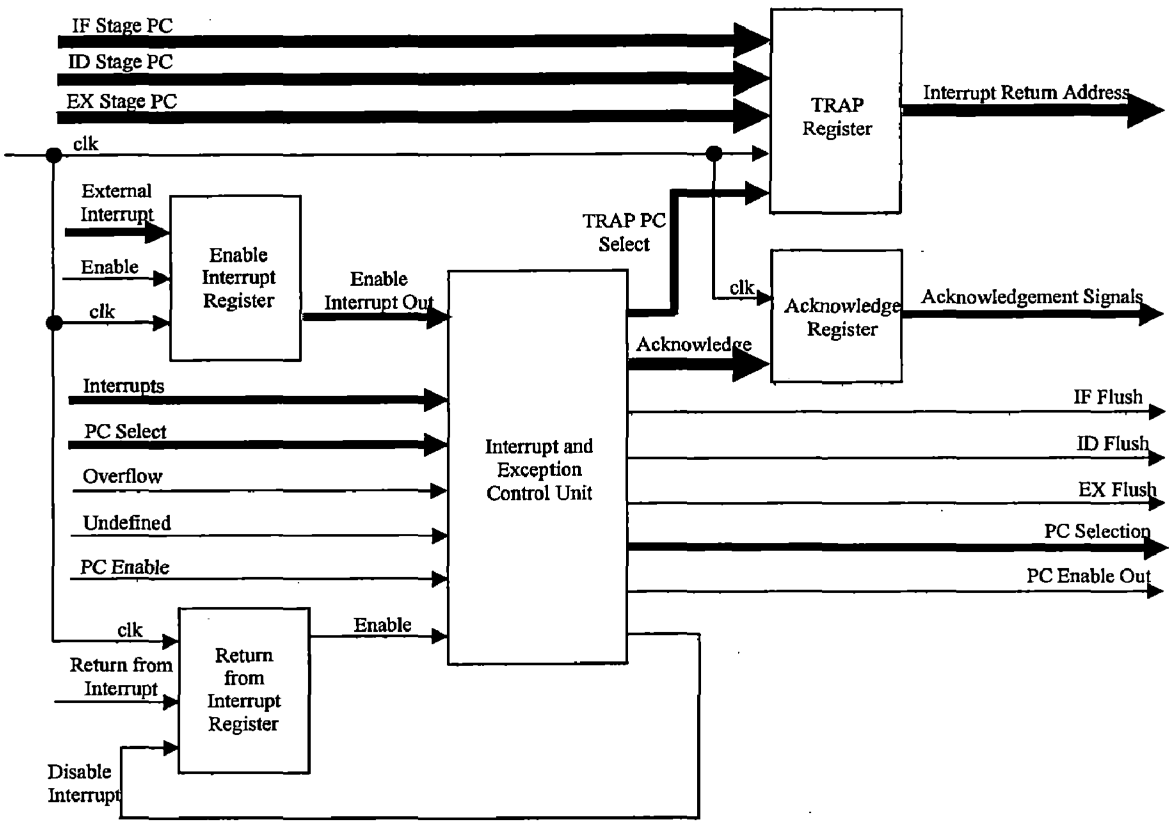


Fig. 5.11 Interrupt and Exception Unit Block Diagram

It consists of interrupt and exception control unit, return from interrupt register, acknowledgement register, enable interrupt register and TRAP register. Block diagram of interrupt and exception unit is shown in fig. 5.11.

❖ **Interrupt and Exception Control Unit.** It checks for arithmetic overflow, undefined instructions or external IO devices to make a request for processing their service routine. If this unit is enabled, each line is sampled in sequence of RESET, overflow, undefined and then hardware lines from interrupt5 to interrupt0. The first line to be true is serviced. The appropriate stages of pipeline are flushed and the unit is disabled. If an external IO device on interrupt lines was serviced, and acknowledgement is sent out and the current program counter value is stored via TRAP register.

- ❖ **Return from Interrupt Register.** When interrupt or exception service routine is being processed, unit is disabled and when return from interrupt or exception is executed unit is again enabled by this register.
- ❖ **Acknowledgement Register.** It is used to hold value of the acknowledgement signals when an interrupt is being processed. When clock goes high then whatever interrupt is being processed is set in the register. The acknowledgement signals are then sent out for device being serviced.
- ❖ **Enable Interrupt Register.** Hardware interrupts can be enabled or disabled using EI instruction. The data part of EI instruction is used for this purpose. The following table shows the details for interrupts and exception. The priority level is in descending order from top to bottom.

Interrupt/ Exception	**Type**	**Data Part of EI Instruction for Enabling**	**Priority Level**
RESET	Hardware, vectored	Can not be disabled	0-highest
Overflow	Exception, vectored	Can not be disabled	1
Undefined	Exception, vectored	Can not be disabled	2
Interrupt5	Hardware, vectored	1 0 0 0 0 0	3
Interrupt4	Hardware, vectored	0 1 0 0 0 0	4
Interrupt3	Hardware, vectored	0 0 1 0 0 0	5
Interrupt2	Hardware, vectored	0 0 0 1 0 0	6
Interrupt1	Hardware, vectored	0 0 0 0 1 0	7
Interrupt0	Hardware, vectored	0 0 0 0 0 1	8

Table 5.5 Interrupts and Exceptions

- **TRAP Register.** It consists of correct PC test unit and PC TRAP register. Block diagram of it is shown in fig. 5.12.
  - **Correct PC Test Unit.** It tests the program counter when an interrupt is being serviced. It is used in case a branch instruction has been issued before the interrupt occurred. This entity picks the correct program counter value to be trapped.
  - **PC TRAP Register.** It is used to store the correct program counter value from the overflow, undefined instruction and the user interrupt.

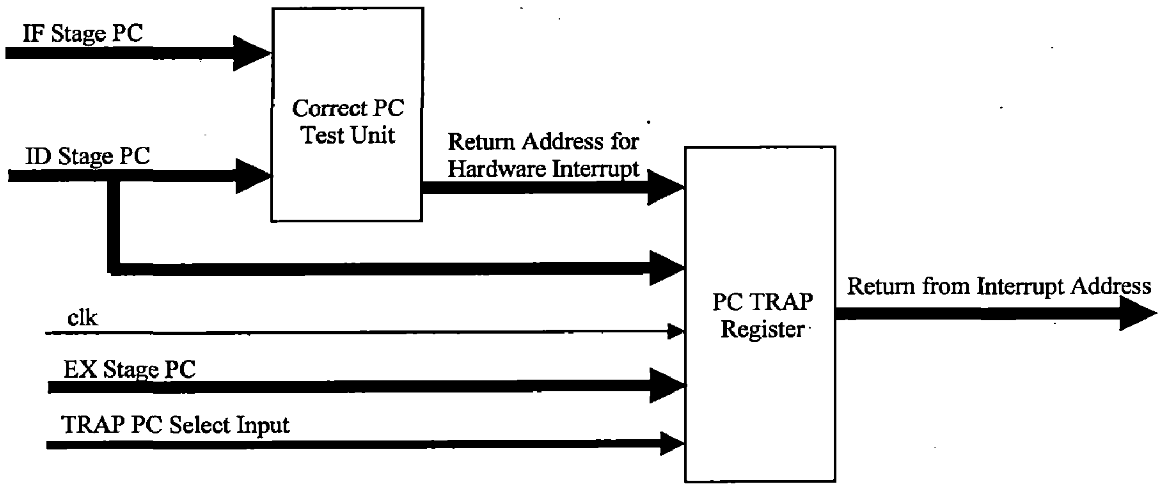


Fig. 5.12 TRAP Register Block Diagram

5.8 Prefetching Unit

When execution unit is not requesting any data from external memory or IO device, prefetcher unit brings the next sequential instructions from memory and store it in the prefetch buffer. This unit supplies the instructions to instruction fetch unit. It consists of prefetch buffer unit, prefetching control unit and clock divider [3]. The block diagram of prefetching unit is shown in fig. 5.13.

- **Prefetch Buffer Unit.** This unit is made up of prefetch buffer and Hit/Miss detection unit. Block diagram of it is shown in fig. 5.14.
  - **Prefetch Buffer.** It is used to store the prefetched instructions from memory. It can store four instructions along with the address from where the particular instruction is fetched and a bit indicating the validity of instruction. It works like small cache memory.

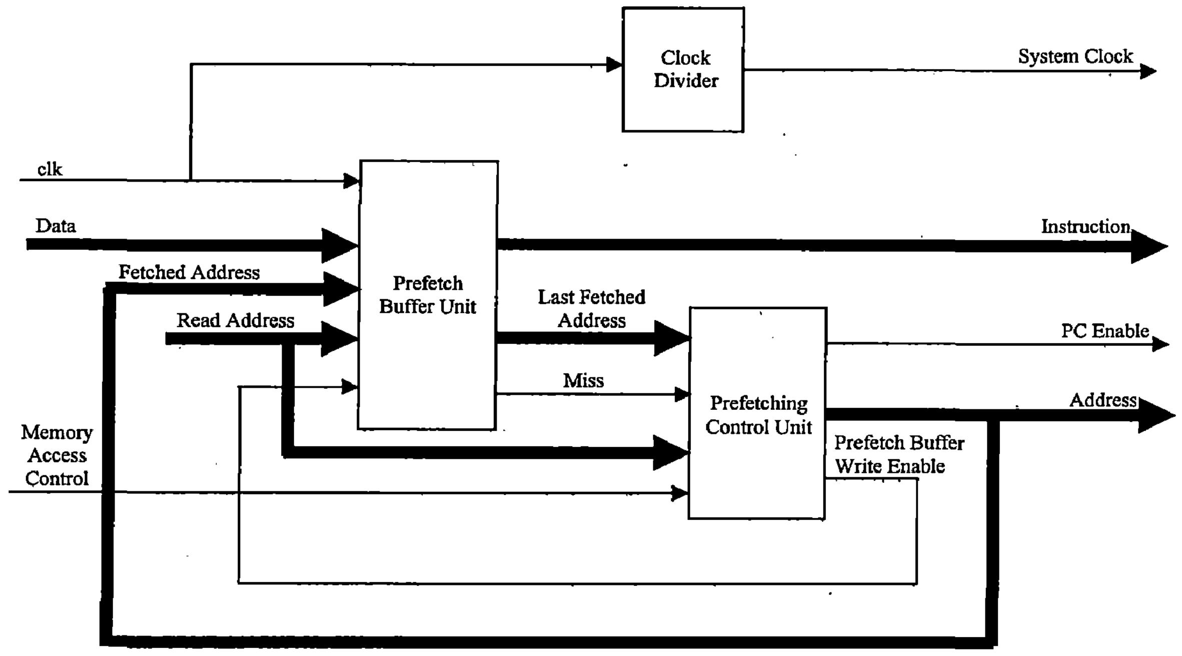


Fig. 5.13 Prefetching Unit Block Diagram

- ➢ **Hit/Miss Detection Unit.** It checks whether read address has a Hit or Miss in the prefetch buffer. It does this by comparing the upper 14 bits of read address along with the tag field of instruction stored in prefetch buffer. If the valid bit is low then instruction is not valid and Miss Output from this unit is set high. If valid bit of instruction is high and 14 bits of read address match with tag field then Miss is set low which indicate that instruction will be directly supplied by the prefetch unit. Otherwise Miss is high and instruction has to be taken from external memory.

- ❖ **Prefetching Control Unit.** Based on the read address received from the instruction fetch stage, the address from which last instruction is prefetched from external memory, the Hit or Miss in prefetch buffer and memory control line, this unit generate the address from which the instruction is to be fetched and also decides whether it is to be written in prefetch buffer.

- ❖ **Clock Divider.** Some components in the system may be slow and require slower clock. Clock divider is used to generate such slower clock signal.

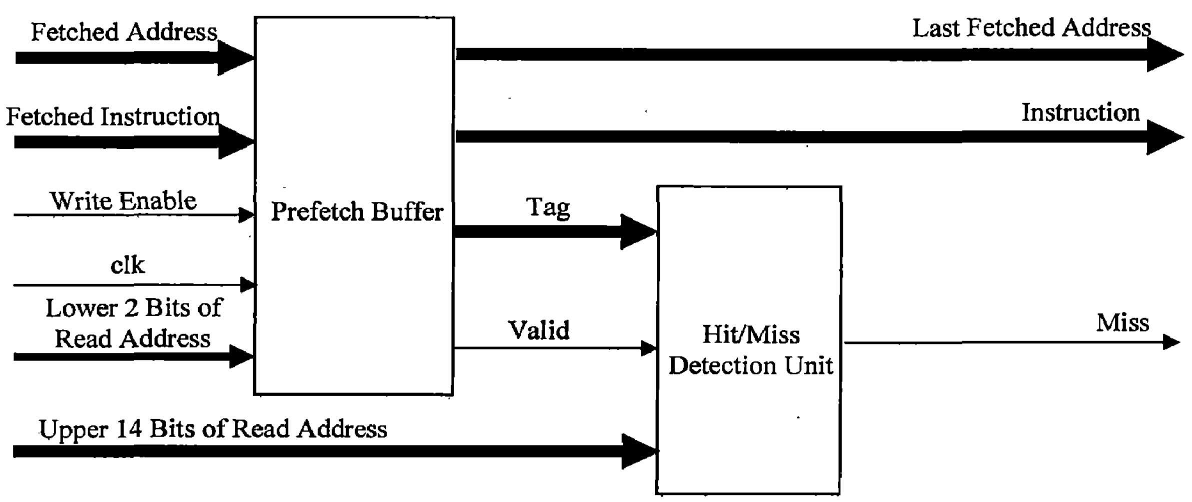


Fig. 5.14 Prefetch Buffer Unit Block Diagram

Chapter 6

DESIGN AND IMPLEMENTATION OF RISC PROCESSOR ON FPGA

VHDL is a language for describing digital electronics system which is used as design tool in this dissertation. Hierarchical approach has been used for designing the RISC processor. After designing the RISC processor and simulating the result, an attempt has been made to implement it on Spartan-II FPGA.

This chapter begins with brief introduction of VHDL language. Next to it basic programming technique are explained. Then emphasis is given on some terminologies in VHDL. Some basics on FPGA are given next. Chapter ends with data on Spartan-II FPGA.

6.1 VHDL

VHDL is an acronym which stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. VHDL is designed to fill a number of needs in the design process. First, it allows description of the structure of a system, that is, how it is decomposed into subsystems and how those subsystems are interconnected. Second, it allows the specification of the function of a system using familiar programming language forms. Third, as a result, it allows the design of a system to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping. Fourth, it allows the detailed structure of a design to be synthesized from a more abstract specification, allowing designers to concentrate on more strategic design decisions and reducing time to market [14].

VHDL is being used for documentation, verification and synthesis of large digital designs. This is actually one of the key features of VHDL, since the same VHDL code can theoretically achieve all three of these goals, thus saving a lot of effort. In addition to being used for each of these purposes, VHDL can be used to take three different approaches to describing hardware. These three different approaches are the structural, data flow and behavioral methods of hardware description. Most of the time a mixture of the three methods is employed. The following sections introduce you to the language by examining its use for each of these three methodologies. [15].

VHDL was established as the IEEE 1076 standard in 1087. In 1993, the IEEE 1076 standard was updated and an additional standard, IEEE 1164 was adopted. In 1996, IEEE 1076.3 became the VHDL synthesis standard [15].

6.2 VHDL Programming Techniques

Various methods can be used to write model for digital circuit using VHDL. Circuit in hand can be modeled using its functional description known as behavioral programming, by describing the system with the help of its component known as structural programming or by mixture of these two known as mixed mode programming [14].

6.2.1 Behavioral Programming

In VHDL, a description of the internal implementation of an entity is called an architecture body of the entity. There may be a number of different architecture bodies of the one interface to an entity, corresponding to alternative implementations that perform the same function. We can write a behavioral architecture body of an entity, which describes the function in an abstract way. Such an architecture body includes only process statements, which are collections of action to be executed in sequence. These actions are called sequential statements and are much like the kinds of statements we see in a conventional programming language. The types of actions that can be performed include evaluating expressions, assigning values to variables, conditional execution, repeated execution and subprogram calls. In addition, there is a sequential statement that is unique to hardware modeling languages, the signal assignment statement. This is similar to variable assignment, except that is causes the value on a signal to be updated at some future time [14].

6.2.2 Structural Programming

An alternative way of describing the implementation of an entity is to specify how it is composed of subsystems. We can give a structural description of the entity's implementation. An architecture body that is composed only of interconnected subsystems is called a structural architecture body. If we are to describe this in VHDL, we will need entity declarations and architecture bodies for the subsystems. Within the architecture body the ports of the entity are also treated as signals. In the second part of the architecture body, a number of component instances are created. Each component instance is a copy of the entity representing the subsystem, using the corresponding basic architecture body. The port map specifies the connection of the ports of each component instance to signals within the enclosing architecture body [14].

6.2.3 Mixed Mode Programming

Models need not be purely structural or purely behavioral. Often it is useful to specify a model with some parts composed of interconnected component instances and other parts described using processes. We use signals as the means of joining component instances and processes. A signal can be associated with a port of a component instance and can also be assigned to or read in a process.

We can write such a hybrid model by including both component instance and process statements in the body of an architecture. These statements are collectively called concurrent statements, since the corresponding processes all execute concurrently when the model is simulated [14].

6.3 Terminologies in VHDL

VHDL is a worldwide standard for the description and modeling of digital hardware. VHDL gives the designer many different ways to describe hardware. The language offers: familiar programming tools for complex and simple problems, sequential and concurrent modes of execution to meet a large variety of design needs, package and libraries to support design management and component reuse [16].

VHDL has ample features appropriate for describing the behavior of electronic components ranging from simple logic gates to complete Microprocessors, High Performance Digital Signal Processor and custom chips. Features of VHDL allow timing aspects of circuit behavior (such as rise and fall times of signals, delays through gates, functional operation) to be precisely described [18].

6.3.1 Entity

This is basic unit of description which gives the input and output ports of the digital circuit to be modeled and their types. For example, if entity with name ABC, X and Y as input ports of type bit, M and N are output ports of type bit will be expressed in VHDL as.

```
Entity ABC is
    Port (X: in std_logic;
          Y: in std_logic;
```

```
        M: out std_logic;
        N: out std_logic);
    End entity ABC;
```

6.3.2 Packages

Packages are intended to hold commonly used declarations such as constants, type declarations and global subprograms. Packages can be included within the same source file as other design units (such as entities and architectures) or may be placed in a separate source file and compiled into a named library. This latter method is useful in using the contents of a package throughout a large design or in multiple projects. The IEEE 1164 standard provides a standard package named std_logic_1164 that includes declarations for the type's std_logic, std_ulogic, std_logic_vector and std_ulogic_vector, as well as many useful functions related to those data types [17].

6.3.3 Design Libraries

A design library is an implementation-dependent storage facility for previously analyzed design units. This results in many different implementations in synthesis and simulation tools. In general, however, design libraries are used to collect commonly used design units (typically packages and package bodies) into uniquely-named areas that can be referenced from multiple source files in your design [15].

6.3.4 Components

Components are used to connect multiple VHDL design units (entity/architecture pairs) together to form a larger, hierarchical design. Using hierarchy can dramatically simplify the design description and can make it much easier to re-use portions of the design in other projects. Components are also useful while making the use of third-party design units, such as simulation models for standard parts, or synthesizable core models obtained from a company specializing in such models [16].

6.3.5 Configurations

Configurations are features of VHDL that allow large, complex design descriptions to be managed during simulation. (Configurations are not generally supported in synthesis). One example of how to use configurations is to construct two versions of a system-level design, one of which makes use of high-level behavioral descriptions of the system components, while a second version substitutes in a post-synthesis timing model of one or more components.

For large projects involving many engineers and many design revisions, configurations can be used to manage versions and specify how a design is to be configured for system simulation, detailed timing simulation and synthesis. Because simulation tools allow configurations to be modified and recompiled without the need to recompile other design units, it is easy to construct alternate configurations of a design very quickly without recompiling the entire design [17].

6.4 FPGA

A field programmable gate array (FPGA) is an inexpensive hardware component, which allows the user to program its functionality quickly and inexpensively. This allows for cheaper prototyping and shorter time to-market of hardware designs. FPGAs have a lower gate density than full custom (customized VLSI chips) and semi custom (mask programmed gate arrays) design methodologies FPGAs were first introduced in the mid-1980s to replace multi-chip glue logic circuits with a single reconfigurable solution [18]. FPGAs have far outgrown their sole use as a replacement for simple glue logic circuits. Presently, FPGA applications include signal and image processing, graphic accelerators, military target correlation/recognition, cryptograph, reconfigurable computing, and on-chip coprocessors. FPGAs are utilized in four major design areas: rapid prototyping emulation, pre-production and full-production [13]. FPGAs are the direct result of the convergence of two distinct technologies: Programmable Logic Devices (PLDs) and Application Specific Integrated Circuits (ASICs) [18]. A simple PLD consists of arrays of AND and OR gates that can be used to create basic circuit designs. ASICs are custom-made chips generally used in high volume applications because non-recurring engineering costs (NREs) are much higher than in an FPGA design cycle. FPGAs are sized from thousand of gates to tens-of-million gates and are available in a variety of sizes with different packaging, internal logic blocks and process technologies [18].

Internal FPGA architectures are commonly constructed using a symmetric tile structure containing a network of switchboxes, logic blocks, wire channels and input-output blocks. A switchbox (SB) is a location in the FPGA fabric that provides a method to connect internal wires together. The switchbox allows horizontal wire segments to switch to vertical wires. The size and contents within a logic block vary greatly depending on the manufacture and target market. For example, FPGAs targeted towards cost-effective solution typically contain simpler logic blocks than an FPGA targeted for high-performance applications. Although the contents within logic blocks can vary for different architectures, there are two basic building blocks found in a logic block: memory elements and function generators. Memory elements provide designers with the ability to temporarily store information until desired conditions are met. Function generators can be configured to produce any function up to the number of inputs into the function generator. Depending on the architecture, some function generators can operate in different modes such as random access memory (RAM), read only memory (ROM), or more complex modes like shift registers. FPGAs are configured through a bitstream that loaded into the device. A bitstream is a file created by the FPGA manufacturer that configures the switchboxes, logic blocks and other internal FPGA logic [19].

FPGAs have redefined the boundaries if digital electronics allowing designers to build systems piecewise. Multiple designers can rapidly test and verify the functionality of each individual piece of a system to ensure proper functionality prior to merging the entire system together. With increasing interest in reconfigurable computing, FPGAs are recognized as the most viable, cost effective solution. Whether a design is statically or dynamically reconfigurable, FPGAs provide rapid programmability and a short time to market design cycle. Many companies have marketed FPGAs, the major companies being Xilinx, Actel and Altera. Reprogrammable FPGAs use EPROM, EEPROM or static RAM technology. Xilinx FPGAs, which use static RAM technology, are the FPGAs used in this thesis [18].

6.5 Terminology in FPGA

Common terminology used in FPGA is explained with the help of specific FPGA device in this section. For this purpose, the Spartan-II FPGA used to implement 16-bit RISC processor is used. First we will explain some common features of FPGA and then explain the terminologies with respect to Spartan-II FPGA.

The Spartan-II 2.5V Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The six-member family offers densities ranging from 15,000 to 200,000 system gates. System performance is supported up to 200 MHz. Spartan-II devices deliver more gates, I/Os, and features per dollar than other FPGAs by combining advanced process technology with a streamlined Virtex-based architecture. Features include block RAM (to 56K bits), distributed RAM (to 75,264 bits), 16 selectable I/O standards, and four Delay-Locked Loops (DLLs). Fast, predictable interconnect means that successive design iterations continue to meet timing requirements. The Spartan-II family is a superior alternative to maskprogrammed ASICs. The FPGA avoids the initial cost, lengthy development cycles and inherent risk of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary (impossible with ASICs) [20].

6.5.1 Features of FPGA

1) Second generation ASIC replacement technology.
   - ❖ Densities as high as 5,292 logic cells with up to 200,000 systems gates.
   - ❖ Streamlined features based on Virtex architecture.
   - ❖ Unlimited reprogrammability.
   - ❖ Very low cost.
   - ❖ Advanced 0.18 micron process.
2) System level features
   - ❖ Select RAM hierarchical memory.
   - ❖ 16-bit/LUT distributed RAM.
   - ❖ Configurable 4K bit block RAM.
   - ❖ Fast interfaces to external RAM.
   - ❖ Fully PCI compliant.
   - ❖ Low-power segmented routing architecture.
   - ❖ Full read back ability for verification/observability.
   - ❖ Dedicated carry logic for high-speed arithmetic.
   - ❖ Efficient multiplier support.
   - ❖ Cascade chain for wide-input functions.
   - ❖ Abundant registers/latches with enable, set, reset.
   - ❖ Four dedicated DLLs for advanced clock control.

- ❖ Four primary low-skew global clock distribution nets.
- ❖ IEEE 1149.1 compatible boundary scans logic.

3) Versatile I/O and packaging
   - ❖ Pb-free package options.
   - ❖ Low-cost packages available in all densities.
   - ❖ Family footprint compatibility in common packages.
   - ❖ 16 high-performance interface standards.
   - ❖ Hot swap Compact PCI friendly.
   - ❖ Zero hold time simplifies system timing.
4) Fully supported by powerful Xilinx development system
   - ❖ Foundation ISE Series: Fully integrated software.
   - ❖ Alliance Series: For use with third-party tools.
   - ❖ Fully automatic mapping, placement and routing.

6.5.2 General Overview of Xilinx Spartan-II FPGA Family

The Spartan-II family of FPGAs have a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable Input/Output Blocks (IOBs). There are four Delay-Locked Loops (DLLs), one at each corner of the die. Two columns of block RAM lie on opposite sides of the die, between the CLBs and the IOB columns. These functional elements are interconnected by a powerful hierarchy of versatile channels. This is shown in fig. 6.1 [20].

Spartan-II Device	**Logic Cells**	**System Gates (Logic and RAM)**	**CLB Array (R x C)**	**Total CLBs**	**Maximum Available User I/O**	**Total Distributed RAM Bits**	**Total Block RAM Bits**
XC2S15	432	15,000	8 x 12	96	86	6,144	16K
XC2S30	972	30,000	12 x 18	216	92	13,824	24K
XC2S50	1,728	50,000	16 x 24	384	176	24,576	32K
XC2S100	2,700	100,000	20 x 30	600	176	38,400	40K
XC2S150	3,880	150,000	24 x 36	864	260	55,296	48K
XC2S200	5,292	200,000	28 x 42	1,176	284	75,264	56K

Table 6.1 Spartan-II FPGA Family Members

Spartan-II FPGAs are customized by loading configurable data into internal static memory cells. Unlimited reprogramming cycles are possible with this approach. Stored values in these cells determine logic functions and interconnections implemented in the FPGA. Configuration data can be read from an external serial PROM (master serial mode), or written into the FPGA in slave serial, slave parallel, or Boundary Scan Modes [20].

Spartan-II FPGAs are typically used in high-volume applications where the versatility of a fast programmable solution adds benefits. Spartan-II devices provide system clock rates up to 200 MHz. Spartan-II FPGAs offer the most cost-effective solution while maintaining leading edge performance. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan-II FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and distributed from), DLL clock drivers, programmable set and reset on all flip-flops, fast carry logic and many other features [20].

6.5.3 Spartan-II Array Description

The Spartan-II user-programmable gate array, shown in figure 1, is composed of five major configurable elements.

- ❖ IOBs provide the interface between the package pins and the internal logic.
- ❖ CLBs provide the functional elements for constructing most logic.
- ❖ Clock DLLs for clock-distribution delay compensation and clock domain control.
- ❖ Versatile multi-level interconnects structure.

As can be seen in fig. 6.1, the CLBs from the central logic structure with easy access to all support and routing structures. The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip. Values stored in static memory cells control all the configurable logic elements and interconnect resources. These values load into the memory cells on power-up and can reload if necessary to change the function of the device. Each of these elements will be discussed in detail in the following sections [20].

- ❖ **Input/Output Block (IOBs).** The Spartan-II IOB, as seen in fig. 6.2, features inputs and outputs that support a wide variety of I/O signaling standards. These high-speed inputs and outputs are capable of supporting various state of the art memory and bus interfaces. The three IOB registers function either ad edge-triggered D-type flip-flops or as level-sensitive latches. Each IOB has a clock signal (CLK) shared by the three registers and independent Clock Enable (CE) signals for each register [19]. In addition to the CLK and CE control signals, the three registers share a Set/Reset (SR). For each register, this signal can be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear. A feature not shown in the block diagram, but controlled by the software, is polarity control. The input and output buffers and all of the IOB control signals have independent polarity controls [20]. Optional pull-up and pull-down resistors and an optional weak-keeper circuit are attached to each pad. Prior to configuration all outputs not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak=keeper circuits are inactive, but inputs may optionally be pulled up. The activation of pull-up resistors prior to configuration is controlled on a global basis by the configuration mode pins. If the pull-up resistors are not activated, all the pins will float. Consequently, external pull-up or pull-down resistors must be provided on pins required to be at a well-defined logic level prior to configuration [20]. All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients. Two forms of over-voltage protection are provided, one that permits 5V compliance, and one that does not. For 5V compliance, a zener-like structure connected to ground turns on when the output rises to approximately 6.5V to the output supply voltage, VCCO. The type of over-voltage protection can be selected independently for each pad [20].

- ❖ **Input Path.** A buffer in the Spartan-II IOB input path routes the input signal either directly to internal logic or through an optional input flip-flop. An optional delay element at the D-input of this flip-flop eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of the FPGA and when used, assures that the pad-to-pad hold time is zero. Each input buffer can be configured to conform to any of the low-voltage signaling standards supported. In some of these standards the input buffer utilizes a user-supplied threshold voltage, VREF. The need to supply VREF imposes constrains on which standards can used in close proximity to each other. There are optional pull-up and pull-down resistors at each input for use after configuration [19].

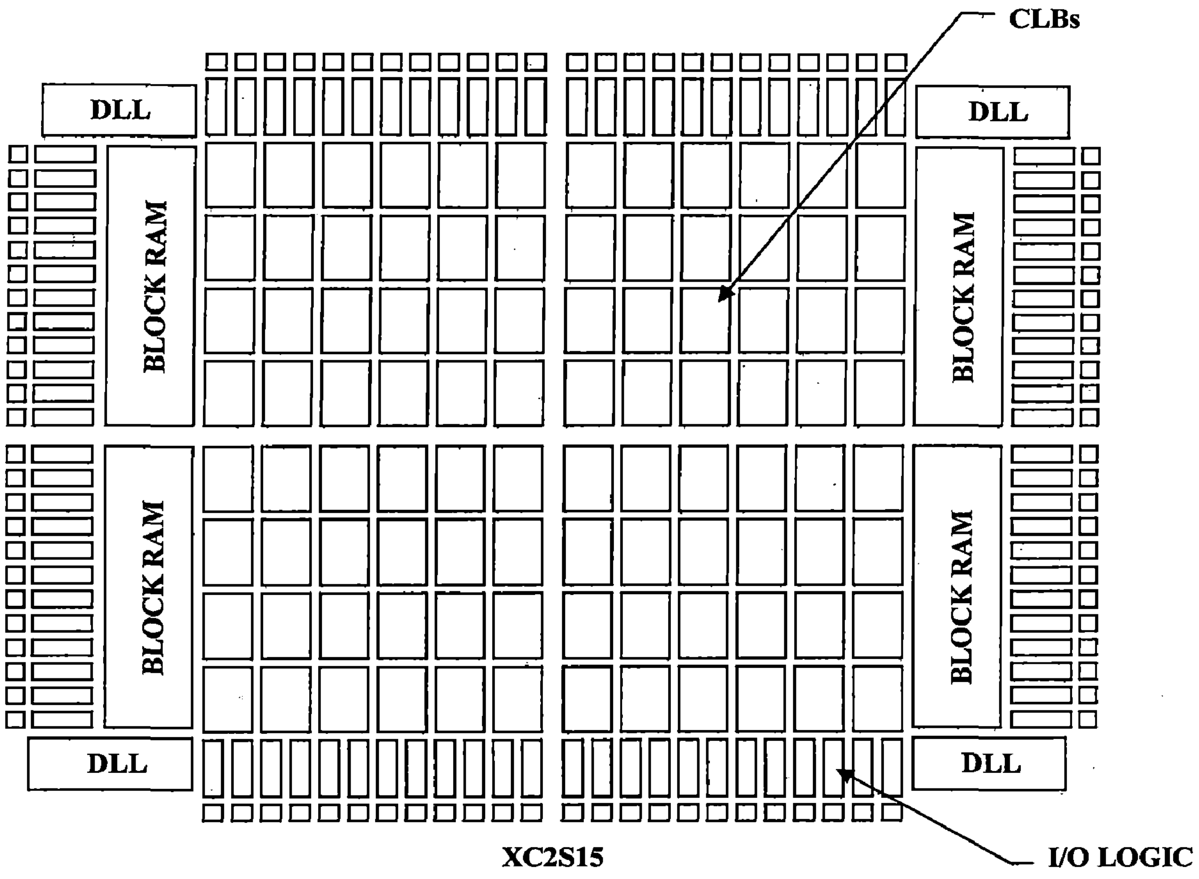


Fig. 6.1 Basic Spartan-II Family FPGA Block Diagram

❖ **Output Path.** The output path includes a 3-stzte output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop. The 3-stzte control of the output can also be routed directly from the internal logic or through a flip-flip that provides synchronous enable and disable. Each output driver can be individually programmed for a wide range of low-voltage signaling standards. Each output buffer can source up to 24 mA and sink up to 48 mA. Drive strength and slew rate controls minimize bus transients [20]. In most signaling standards, the output high voltage depends on an externally supplied VCCO voltage. The need to supply VCCO imposes constraints on which standards can be used in close proximity to each other. An optional weak-keeper circuit is connected to each output. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low to match the input signal. If the pin is a connected to a multiple-source signal, the weak keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way helps eliminate bus chatter. Because the weak-keeper circuit uses the IOB input buffer to monitor the input level, an appropriate VREF voltage must be provided if the signaling standards require one. The provision of this voltage must comply with the I/O banking rules [20].

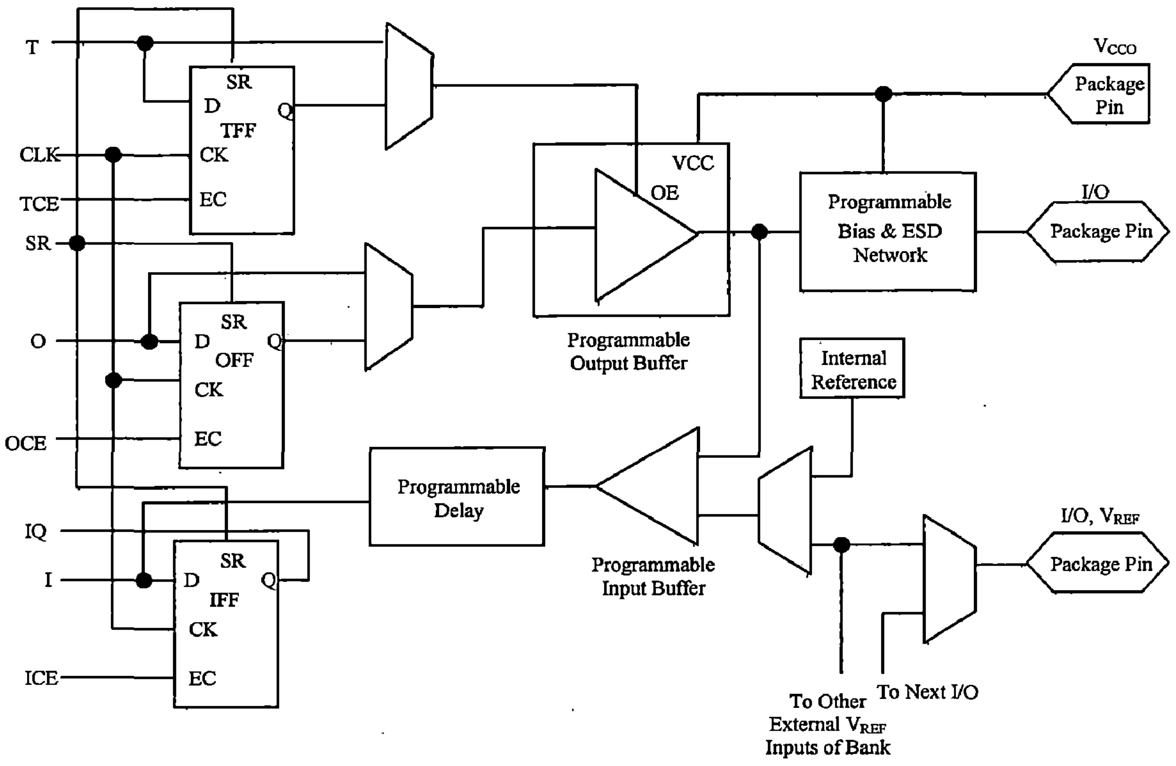


Fig. 6.2 Spartan-II Input/Output Block (IOB)

❖ **I/O Banking.** Some of the I/O standards described above require VCCO and/or VREF voltage. These voltages are externally connected to device pins that serve groups of IOBs, called banks. Consequently, restrictions exist about which I/O standards can be combined within a given bank. Eight I/O banks result from separating each edge of the FPGA into two banks as shown in figure 3. Each bank has multiple VCCO pins which must be connected to the same voltage. Voltage is determined by the output standards in use [20]. Some input standards require a user-supplied threshold voltage, V_{REF}. In this case, certain user-I/O pins are automatically configured as inputs for the V_{REF} voltage. About one in six of the I/O pins in the bank assume this role. V_{REF} pins within a bank are interconnected internally and consequently only one V_{REF} voltage can be used within each bank. All V_{REF} pins in the bank, however, must be connected to the external voltage source for correct operation. In a bank, inputs requiring V_{REF} can be mixed with those that do not but only one V_{REF} may be used within a bank. Input buffers that use V_{REF} are not 5V tolerant. The V_{CCO} and V_{REF} pins for each bank appear in the device pin-out tables. Within a given package, the number of V_{REF} and V_{CCO} pins can vary depending on the size of device. In larger devices, more I/O pins convert to V_{REF} pins. Since these are always a superset of the V_{REF} pins used for smaller devices, it is possible to design a PCB that permits migration to a larger device. All V_{REF} pins for the largest device anticipated must be connected to the V_{REF} voltage and not used for I/O [20].

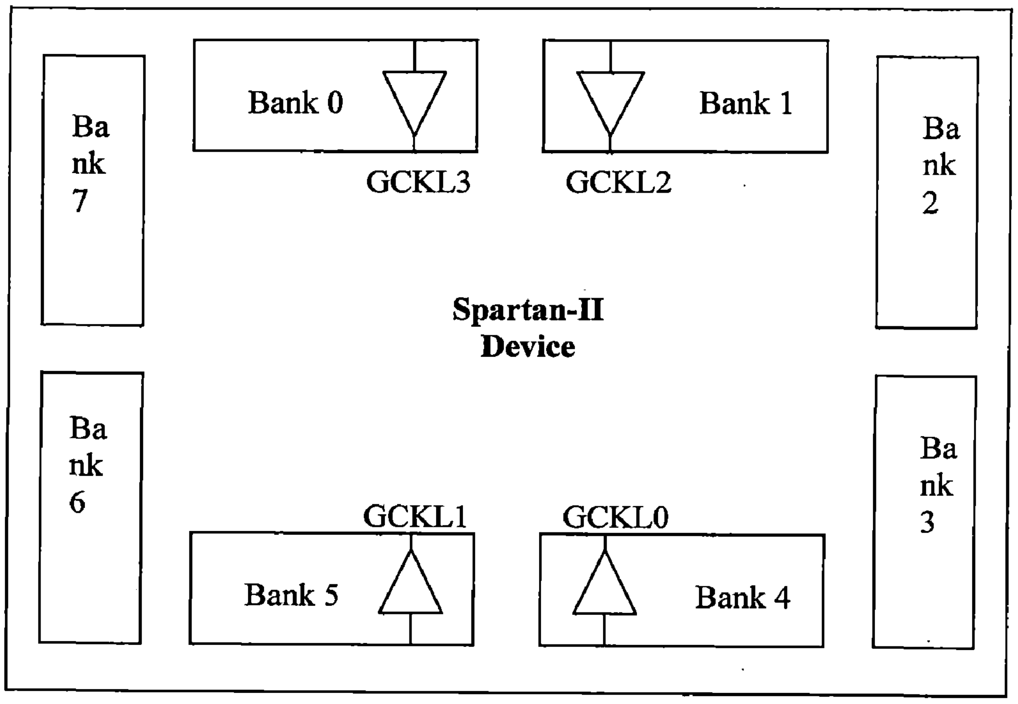


Fig. 6.3 Spartan-II I/O Banks

- **Configurable Logic Block (CLBs).** The basic building block of the Spartan-II CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic and storage element. Output from the function generator in each LC drives the CLB output and the D input of the flip-flop. Each Spartan-II CLB contains four LCs, organized in two similar slices; a single slice is shown in fig. 6.4. In addition to the four basic LCs, the Spartan-II CLB contains logic that combines function generators to provide functions of five or six inputs [20].
- **Look-Up Tables (LUTs).** Spartan-II function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16x1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16x2-bit or 32x1-bit synchronous RAM, or a 16x1-bit dual-port synchronous RAM. The Spartan-II LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-mode data. This mode can also be used to store data in application such as Digital Signal Processing [20].
- **Storage Elements.** Storage elements in the Spartan-II slice can be configured either as edge-triggered D-type flip-flop or as level-sensitive latches. The D inputs can be driven either by function generators within the slice or directly from slice inputs, bypassing the function generators. In addition to Clock and Clock Enable signals, each slice has synchronous set and reset signals (SR and BY). SR forces a storage element into the initialization state specified for it in the configuration. BY forces it into the opposite state. Alternatively, these signals may be configured to operate asynchronously. All control signals are independently invertible and are shared by the two flip-flops within the slice [20].
- **Additional Logic.** The multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up nine inputs. Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs. Each

CLB has four direct feed through paths; one per LC. These paths provide extra data input lines or additional local routing that does not consume logic resources [20].

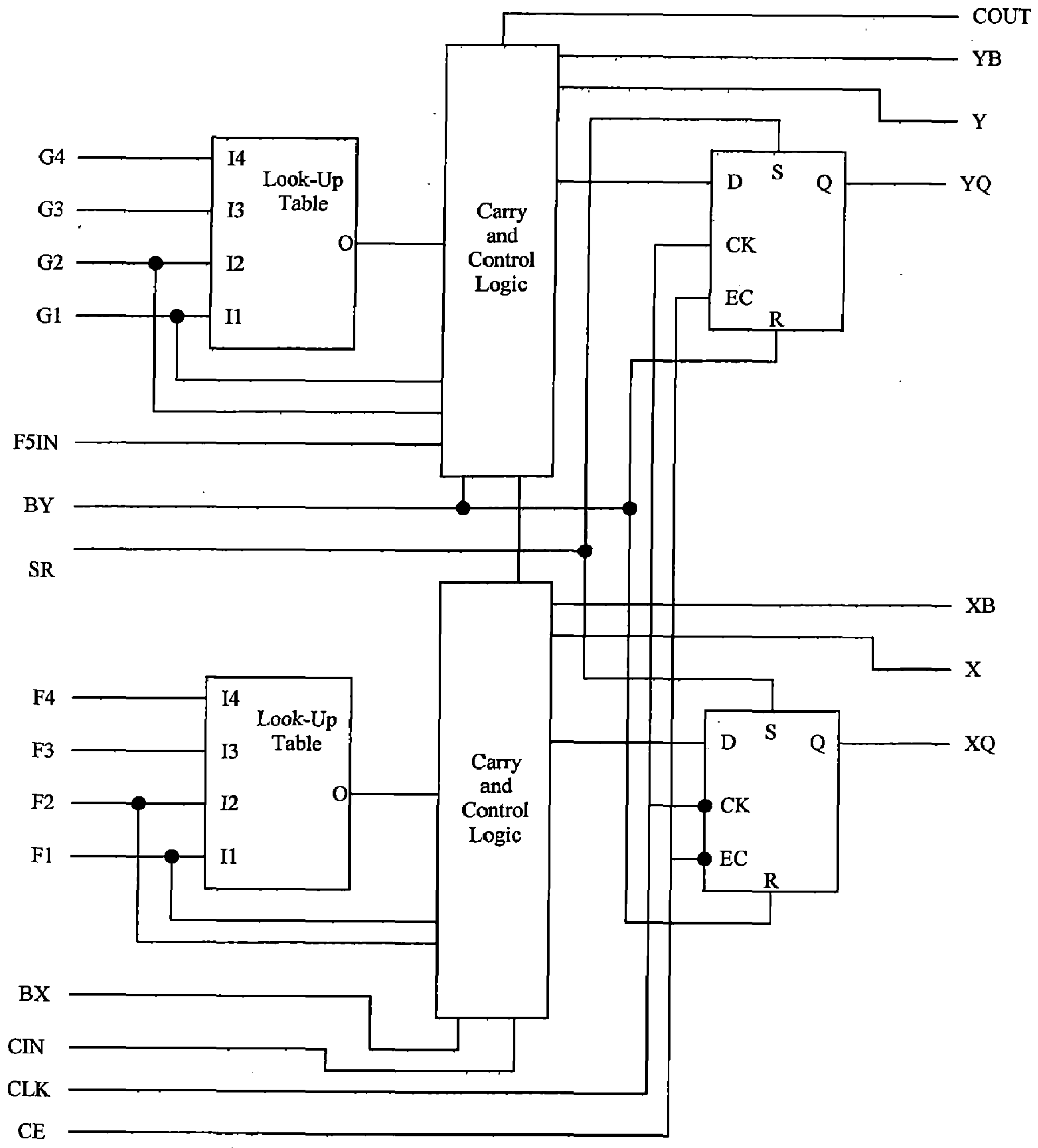


Fig. 6.4 Spartan-II CLB Slice (two identical in each CLB)

❖ **Arithmetic Logic.** Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Spartan-II CLB supports two separate carry chains, one per slice. The height of the carry chains is two bits per CLB. The arithmetic logic includes an XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementation. The dedicated carry path can also be to cascade function generators for implementing wide logic functions [20].

- **BUFTs.** Each Spartan-II CLB contains two 3-state drivers (BUFTs) that can drive on-chip busses. Each Spartan-II BUFT has an independent 3-state control pin and an independent input pin.
- **Block RAM.** Spartan-II FPGAs incorporate several large block RAM memories. These complements the distributed RAM Look-Up Tables (LUTs) that provide shallow memory structures implemented in CLBs. Block RAM memory blocks are organized in columns. All Spartan-II devices contain two such columns, one along each vertical edge. These columns extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Spartan-II device eight CLBs high will contain two memory blocks per column and a total of four blocks. Each block RAM cell is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion. The Spartan-II block RAM also includes dedicated routing to provide an efficient interface with both CLBs and other block RAMs [20]

Spartan-II Device	**No. of Blocks**	**Total Block RAM Bits**
XC2S15	4	16K
XC2S30	6	24K
XC2S50	8	32K
XC2S100	10	40K
XC2S150	12	48K
XC2S200	14	56K

Table 6.2 Spartan-II Block RAM Amounts

Chapter 7

SIMULATION RESULTS AND CONCLUSION

Basic entities have been programmed using behavioral model and they are used as components to form the different units of RISC processor. These units in term are used as components for building the RISC processor using structural programming. The simulation results for different unit and final RISC processor are shown below.

Simulation results for stage1 shown in fig, 7.1. Based on value of selection input (OPC) generated by control unit for instruction in execution stage, next program counter value (pcValue) is decided. Instruction from prefetch unit at address specified in program counter is taken and is forwarded to next stage. If flush input is high, instruction fetch stage register is flushed out.

Simulation results for stage2 are shown in fig. 7.2. This stage decides the combination of read registers based on regSelect input. Input rf_enable is used for enabling the write operation in register file. If branch instruction is there in execution stage, then type of branching (absolute or relative) is decided by branch_select input. The contents of read registers are forwarded to next stage.

Simulation results for stage3 are shown in fig. 7.3. The ex_select input activated one of the components out of basic ALU, shift unit and move immediate unit. The input alu_function decides the operation to be carried out on the data at input. Contents of source registers and sign extended immediate data are input to this stage. The result and the code for write back register are forwarded to next stage.

Simulation results for stage4 are shown in fig. 7.4. The input memSel is used to select from data on input port, data read from memory and result of execution stage as final result to be written back into destination. When output_enable is high, the result is placed on the output port. Final result and code for write-back register are output from this stage.

Simulation results for control unit are shown in fig. 7.5. Based on combinations of opcode and function, it generates the control signals for all the units of processor. For example, it generates alu_function signal for stage3 (execution stage). The control signals are sequenced properly.

Simulation results for branch forwarding unit are shown in fig. 7.6. This unit selects the proper branch target address. If resource conflict is there in branch instruction, then result from execution unit is selected as target address. Otherwise the content of register specified in instruction is used as target address.

Simulation results for hazard detection unit are shown in fig. 7.7. If the destination of instruction is source of next instruction then the necessary control signals are generated by this unit. Also in case of branch instruction, the instruction following the branch instruction needs to be flushed out.

Simulation results for interrupt and exception unit are shown in fig. 7.8. RESET has been assigned highest priority and when it is high, the pcSel output is set to FH to select the new PC value as 0012H in stage1. In case of hardware interrupts and exceptions, the proper return address is saved in TRAP register.

Simulation results for prefetch unit are shown in fig. 7.9. It receives the address from instruction fetch stage and supplies instructions to it. When data bus is idle it prefetches the instructions from memory and store them in prefetch queue.

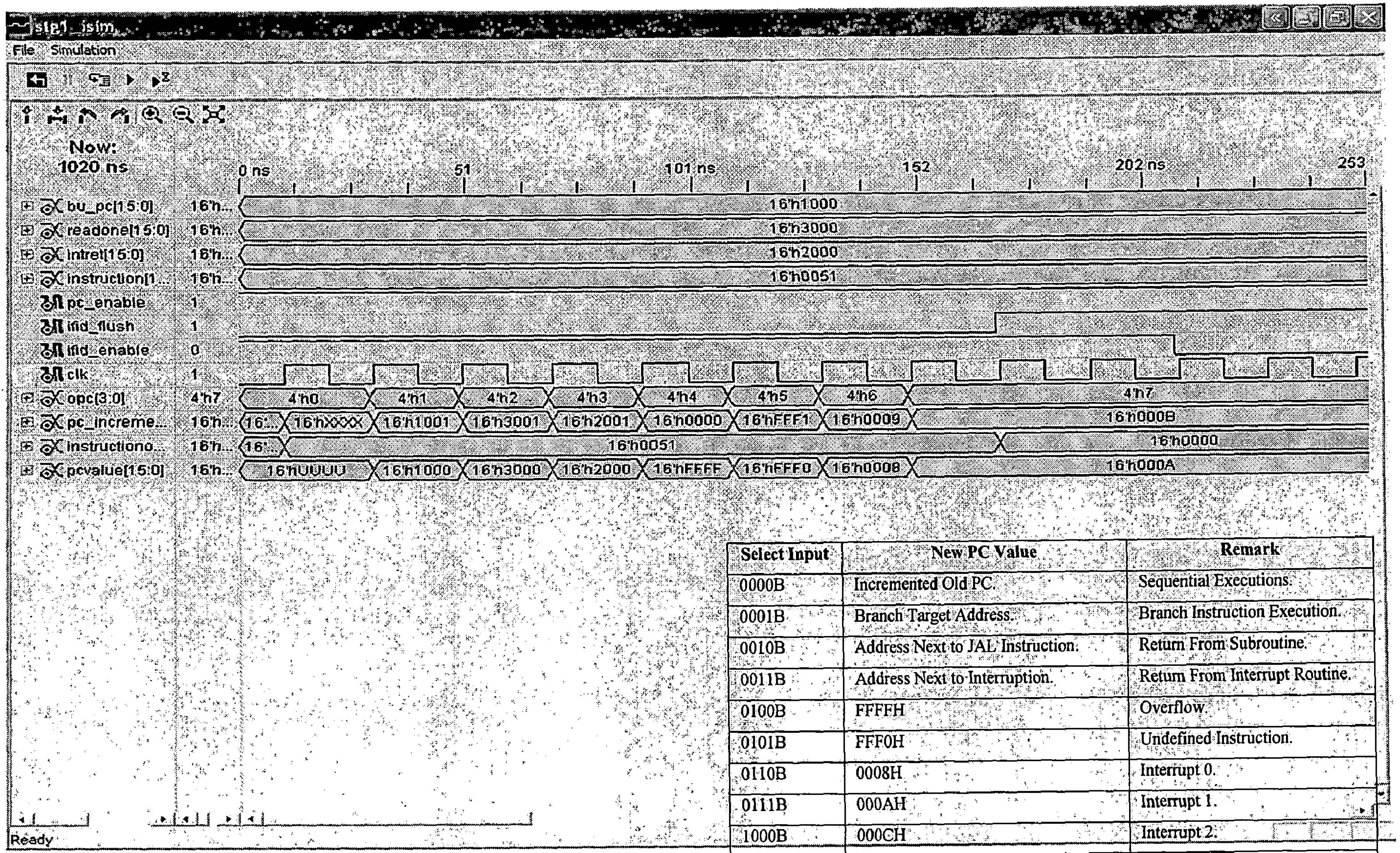
Simulation results for designed RISC processor connected to memory are shown in fig. 7.10. This is the simulation result for small program of addition having resource conflicts. The data for addition was 1010H and 3030H. Simulation result shows that this conflict is successfully handled.

Hierarchical approach greatly simplifies the design of a processor. Because of this it is possible to model basic units of processor using behavioral programming method of VHDL at an elementary level. Pipelining is the most important part of any processor. Division of execution process of an instruction is critical in designing pipelining. If the work is not distributed properly over different stages of pipeline, performance degrades because the slowest stage decides the throughput of the pipeline. Maintaining the sequence in the working of the pipeline is also one of the important points to consider while designing stages of pipeline.

Hardwired approach is used in this design because instructions are very few, simple operations are to be carried out on the data, instruction length is fixed and hence less number of control signals need to be generated for all the instructions. Clock signal is used as the basis for generating control signals in sequence for all the pipelined stages and other units of processor.

Structural hazards, data hazards and control hazards are resolved using hazard detection unit, execution forwarding unit and branch forwarding unit. An instruction called JAL and RJAL has been designed equivalent to CALL and RETURN with the difference that the return address is saved in one of the register rather than external memory. Only LOAD and STORE instructions access the external memory. All other instructions operate on a data present in internal registers or immediate data present in the instruction itself.

An attempt has been made to implement the designed processor on Spartan-II FPGA.


stp1_isim
File Simulation
Now:
1020 ns
0 ns
51
101 ns
152
202 ns
253
bu_pc[15:0] 16'h... 16'h1000
readone[15:0] 16'h... 16'h3000
intret[15:0] 16'h... 16'h2000
instruction[1... 16'h... 16'h0051
pc_enable 1
ifid_flush 1
ifid_enable 0
clk 1
opc[3:0] 4'h7 4'h0 4'h1 4'h2 4'h3 4'h4 4'h5 4'h6 4'h7
pc_increme... 16'h... 16'... 16'hXXXX 16'h1001 16'h3001 16'h2001 16'h0000 16'hFFF1 16'h0009 16'h000B
instructiono... 16'h... 16'... 16'h0051 16'h0000
pcvalue[15:0] 16'h... 16'hUUUU 16'h1000 16'h3000 16'h2000 16'hFFFF 16'hFFF0 16'h0008 16'h000A
Ready

Select Input	New PC Value	Remark
0000B	Incremented Old PC.	Sequential Executions.
0001B	Branch Target Address.	Branch Instruction Execution.
0010B	Address Next to JAL Instruction.	Return From Subroutine.
0011B	Address Next to Interruption.	Return From Interrupt Routine.
0100B	FFFFH	Overflow.
0101B	FFF0H	Undefined Instruction.
0110B	0008H	Interrupt 0.
0111B	000AH	Interrupt 1.
1000B	000CH	Interrupt 2.
1001B	000EH	Interrupt 3.
1010B	0010H	Interrupt 4.
1011B	FFF8H	Interrupt 5.
Others	0012H	RESET.

Fig. 7.1 Simulation Result of Stage-1 (Instruction Fetch Stage)

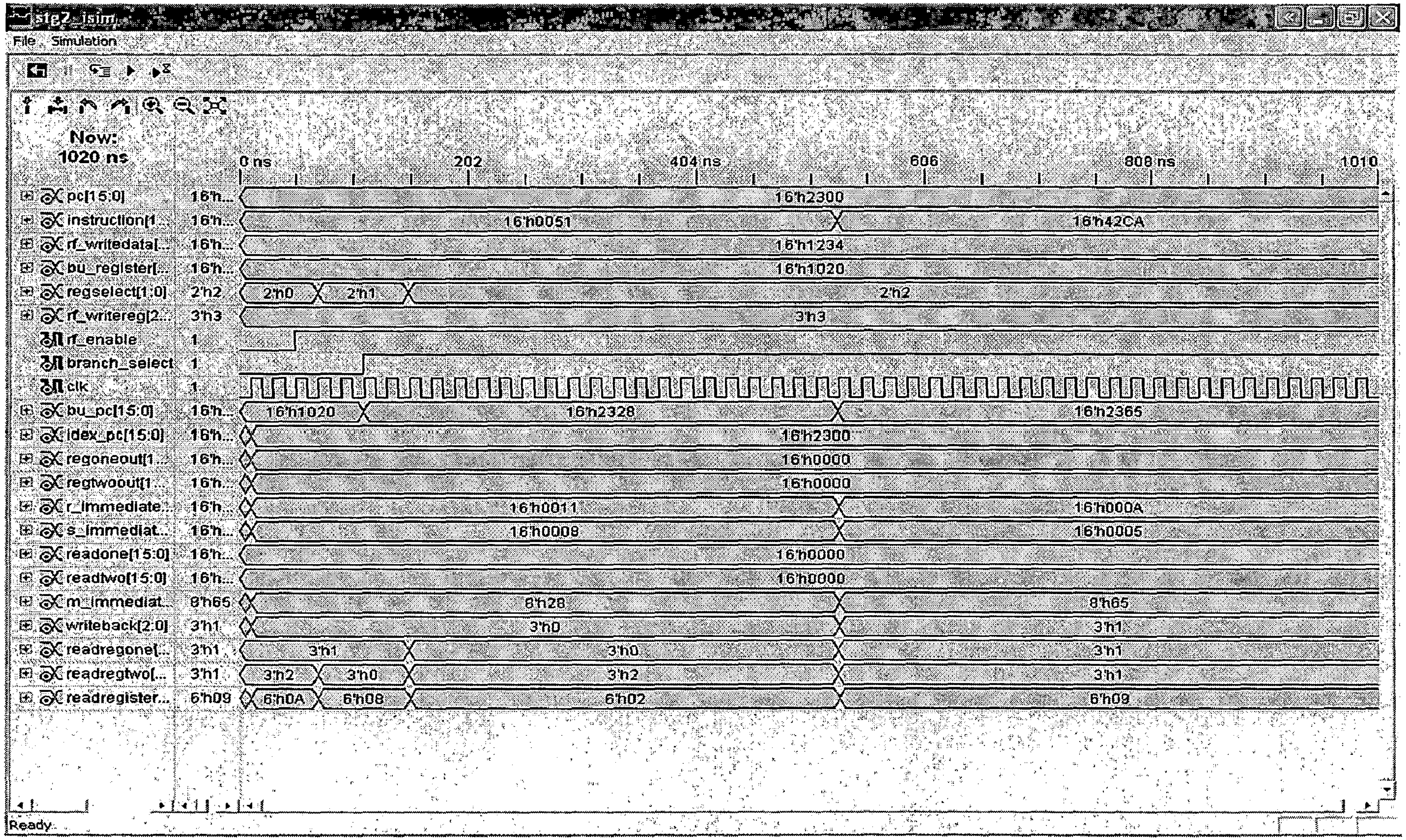

stg2_isim
File Simulation
Now:
1020 ns
0 ns
202
404 ns
606
808 ns
1010
pc[15:0] 16'h... 16'h2300
instruction[1... 16'h... 16'h0051 16'h42CA
rf_writedata[... 16'h... 16'h1234
bu_register[... 16'h... 16'h1020
regselect[1:0] 2'h2 2'h0 2'h1 2'h2
rf_writereg[2... 3'h3 3'h3
rf_enable 1
branch_select 1
clk 1
bu_pc[15:0] 16'h... 16'h1020 16'h2328 16'h2365
idex_pc[15:0] 16'h... 16'h2300
regoneout[1... 16'h... 16'h0000
regtwoout[1... 16'h... 16'h0000
r_immediate... 16'h... 16'h0011 16'h000A
s_immediat... 16'h... 16'h0008 16'h0005
readone[15:0] 16'h... 16'h0000
readtwo[15:0] 16'h... 16'h0000
m_immediat... 8'h65 8'h28 8'h65
writeback[2:0] 3'h1 3'h0 3'h1
readregone[... 3'h1 3'h1 3'h0 3'h1
readregtwo[... 3'h1 3'h2 3'h0 3'h2 3'h1
readregister... 6'h09 6'h0A 6'h08 6'h02 6'h09
Ready

Fig. 7.2 Simulation Result of Stage-2 (Instruction Decode Stage)

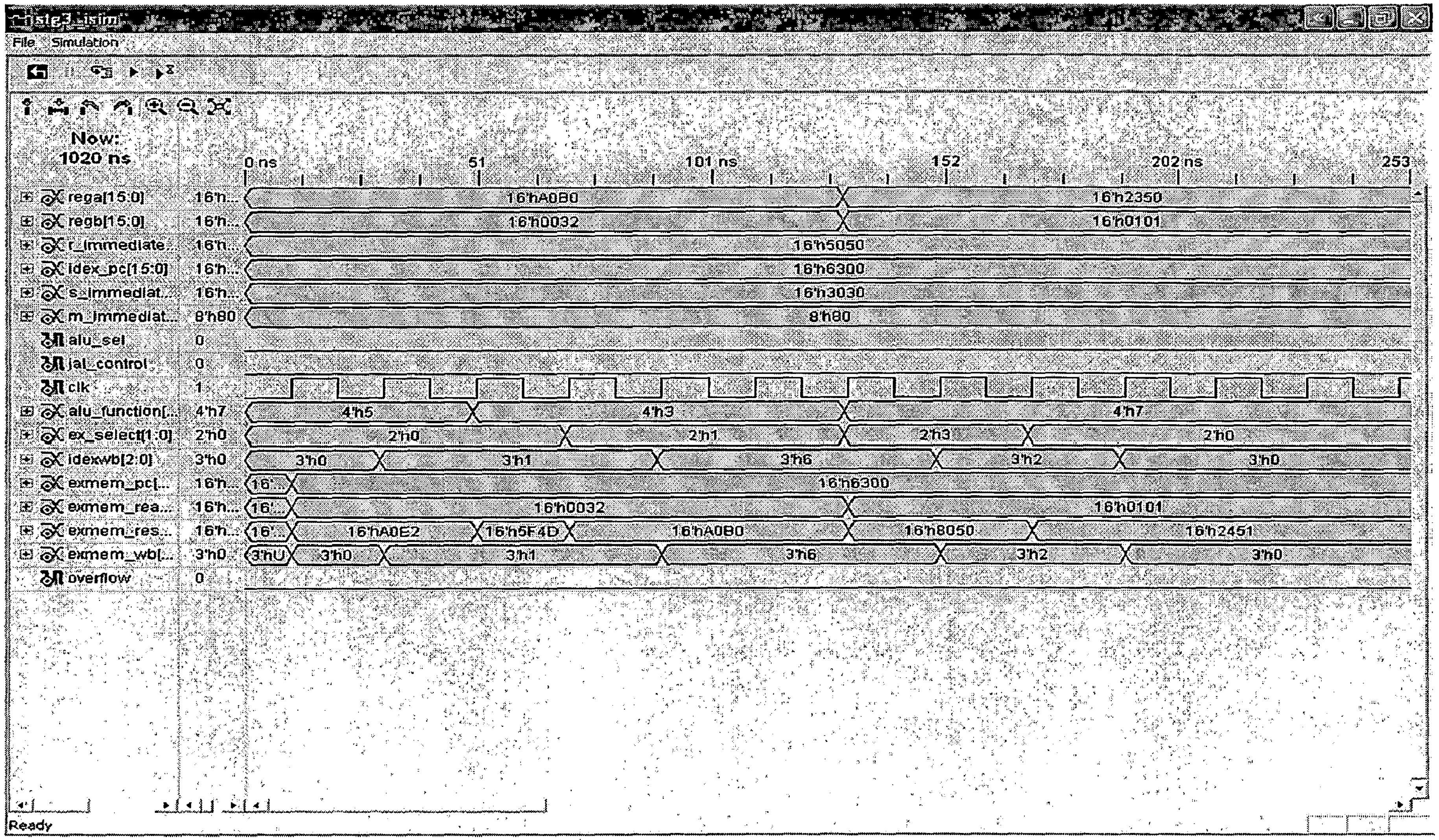

stg3_isim
File Simulation
Now: 1020 ns
0 ns
51
101 ns
152
202 ns
253
rega[15:0] 16'h... 16'hA0B0 16'h2350
regb[15:0] 16'h... 16'h0032 16'h0101
r_immediate... 16'h... 16'h5050
idex_pc[15:0] 16'h... 16'h6300
s_immediat... 16'h... 16'h3030
m_immediat... 8'h80 8'h80
alu_sel 0
jal_control 0
clk 1
alu_function[... 4'h7 4'h5 4'h3 4'h7
ex_select[1:0] 2'h0 2'h0 2'h1 2'h3 2'h0
idexwb[2:0] 3'h0 3'h0 3'h1 3'h6 3'h2 3'h0
exmem_pc[... 16'h... 16'... 16'h6300
exmem_rea... 16'h... 16'... 16'h0032 16'h0101
exmem_res... 16'h... 16'... 16'hA0E2 16'h5F4D 16'hA0B0 16'h8050 16'h2451
exmem_wb[... 3'h0 3'hU 3'h0 3'h1 3'h6 3'h2 3'h0
overflow 0
Ready

Fig. 7.3 Simulation Result of Stage-3 (Instruction Execution Stage)

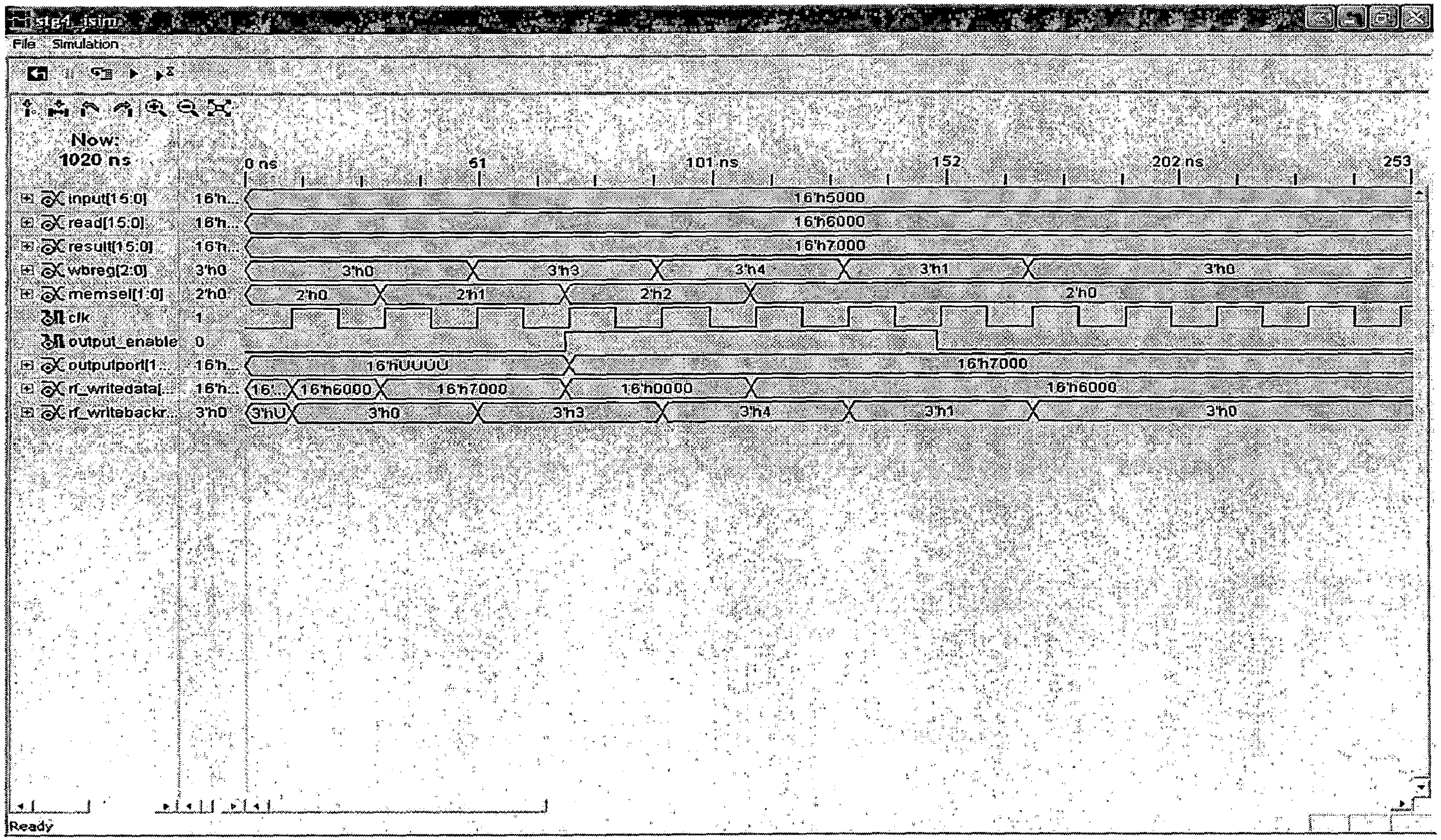

stg4_isim
File Simulation
Now:
1020 ns
0 ns
51
101 ns
152
202 ns
253
input[15:0] 16'h... 16'h5000
read[15:0] 16'h... 16'h6000
result[15:0] 16'h... 16'h7000
wbreg[2:0] 3'h0 3'h0 3'h3 3'h4 3'h1 3'h0
memsel[1:0] 2'h0 2'h0 2'h1 2'h2 2'h0
clk 1
output_enable 0
outputport[1... 16'h... 16'hUUUU 16'h7000
rf_writedata[... 16'h... 16'... 16'h6000 16'h7000 16'h0000 16'h6000
rf_writebackr... 3'h0 3'hU 3'h0 3'h3 3'h4 3'h1 3'h0
Ready

Fig. 7.4 Simulation Result of Stage-4 (Memory/IO-Write Back Stage)

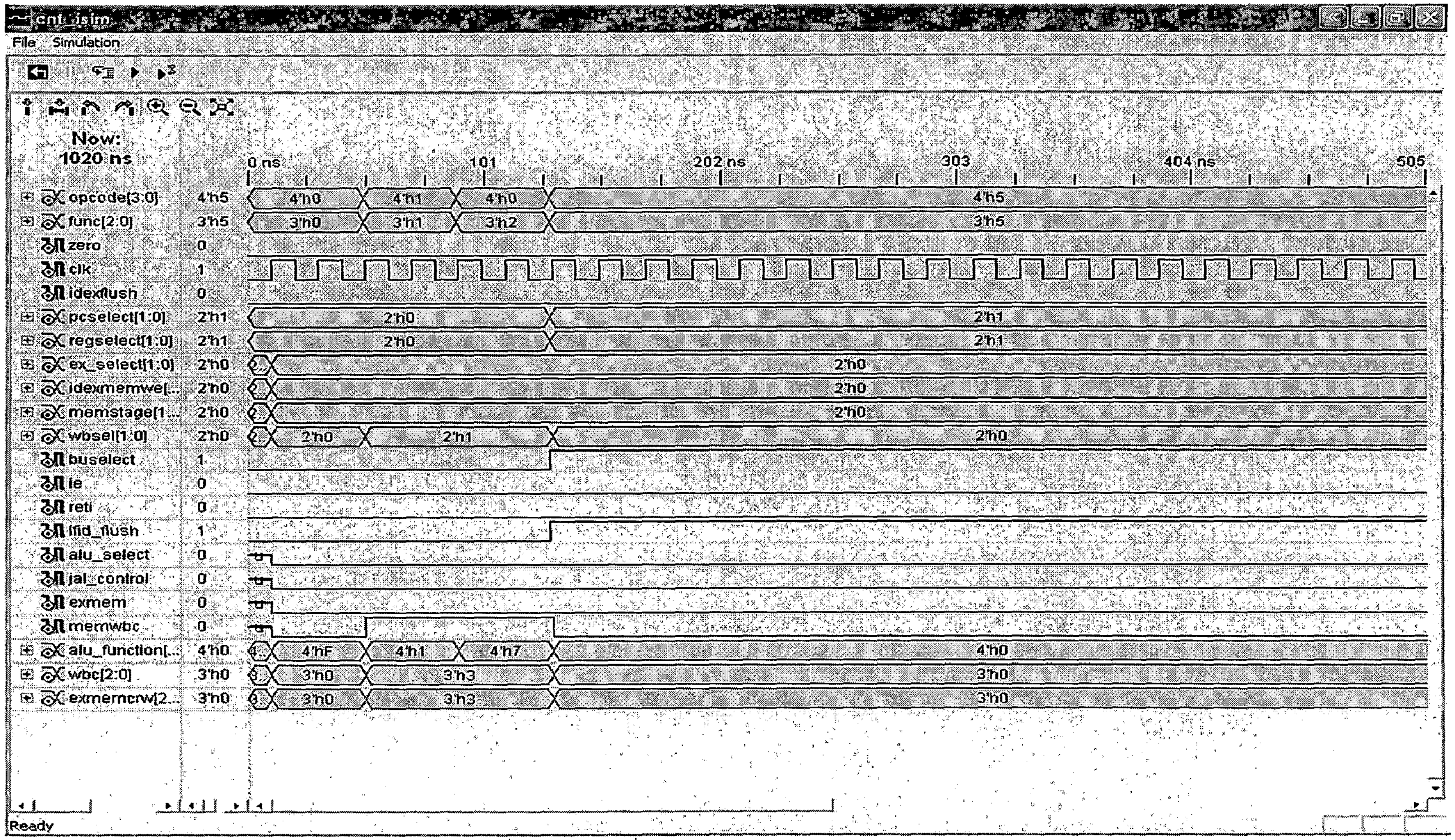


Fig. 7.5 Simulation Result of Control Unit

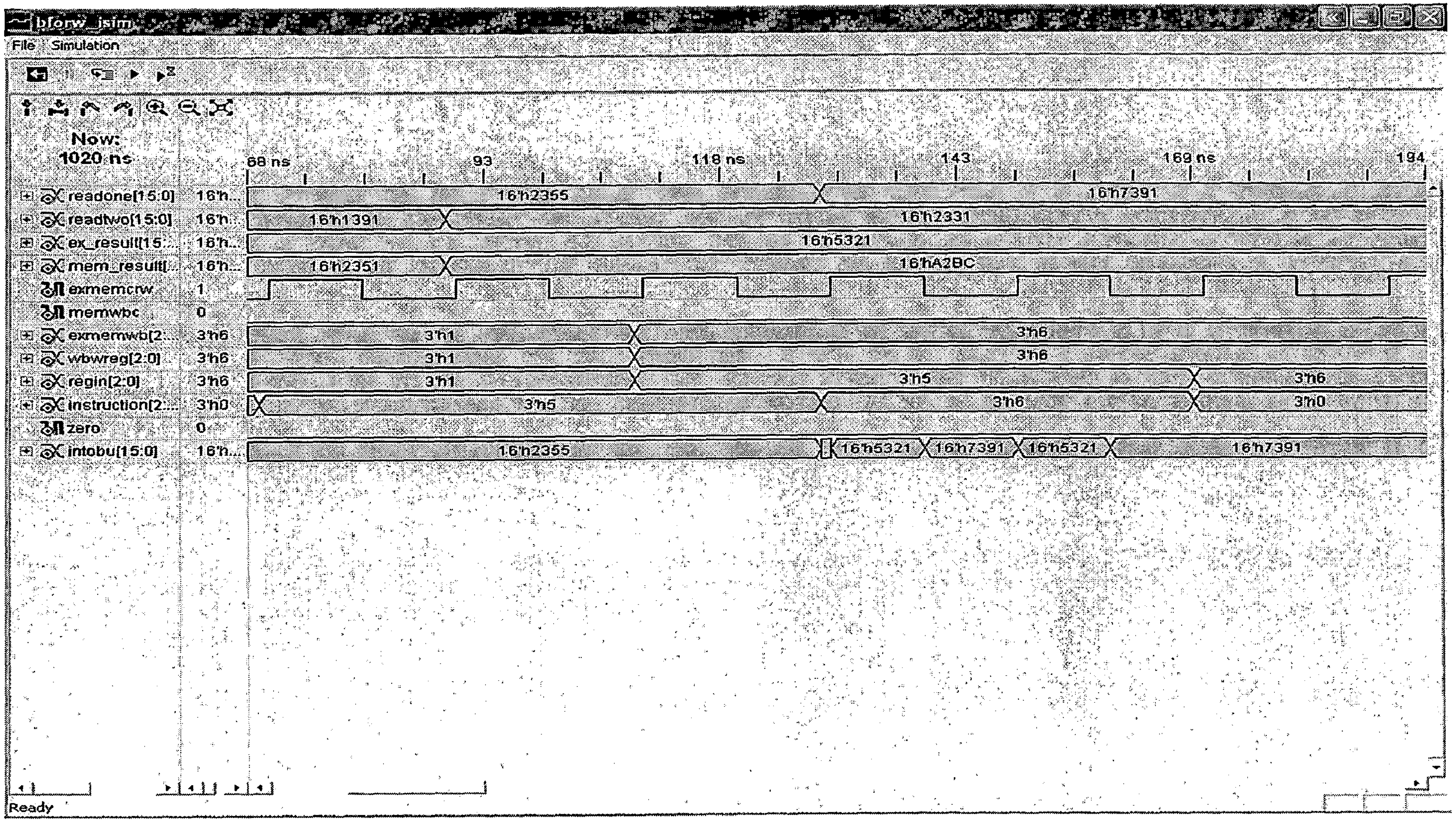


Fig. 7.6 Simulation Result of Branch Forwarding Unit

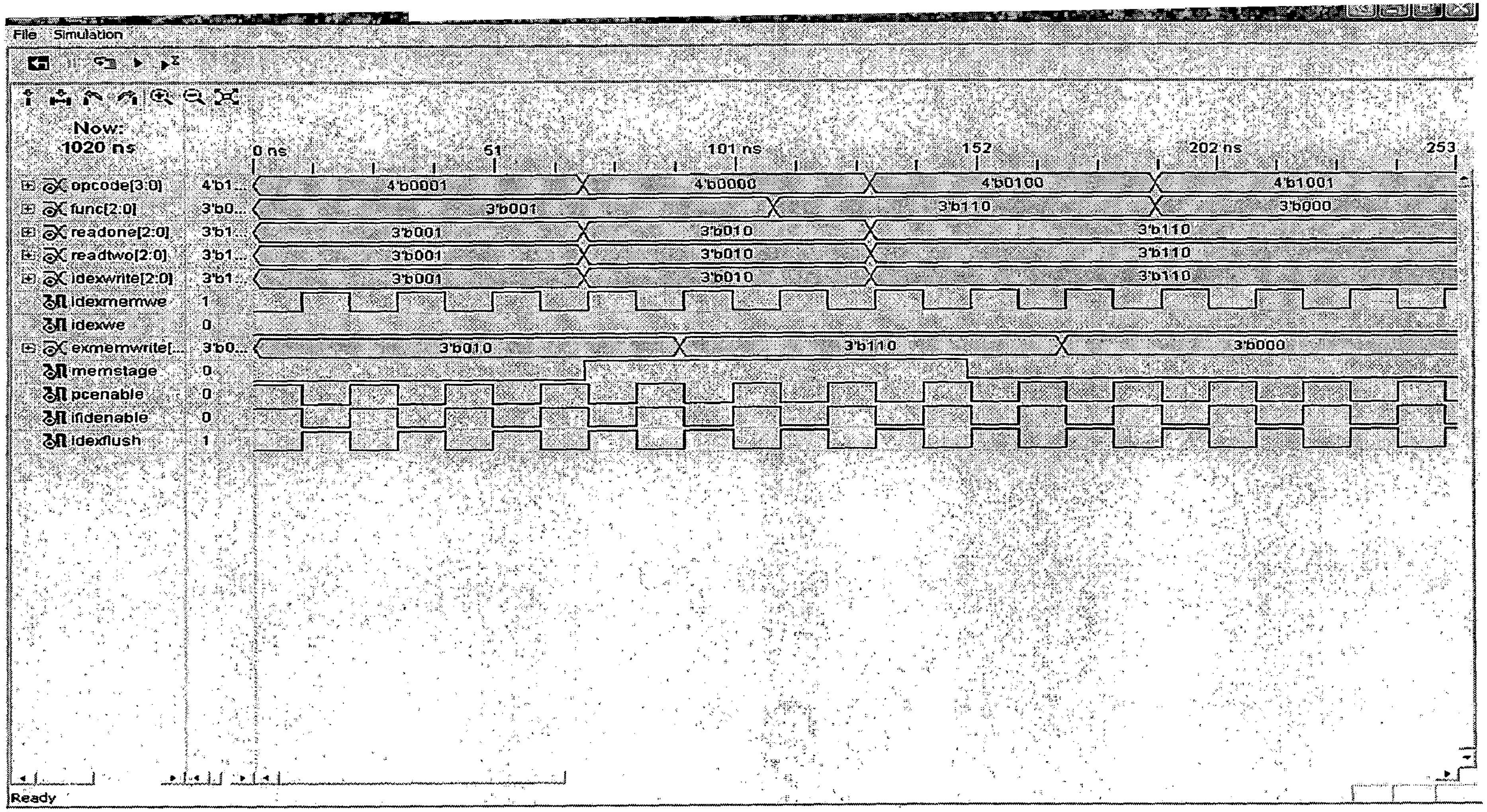


Fig. 7.7 Simulation Result of Hazard Detection Unit

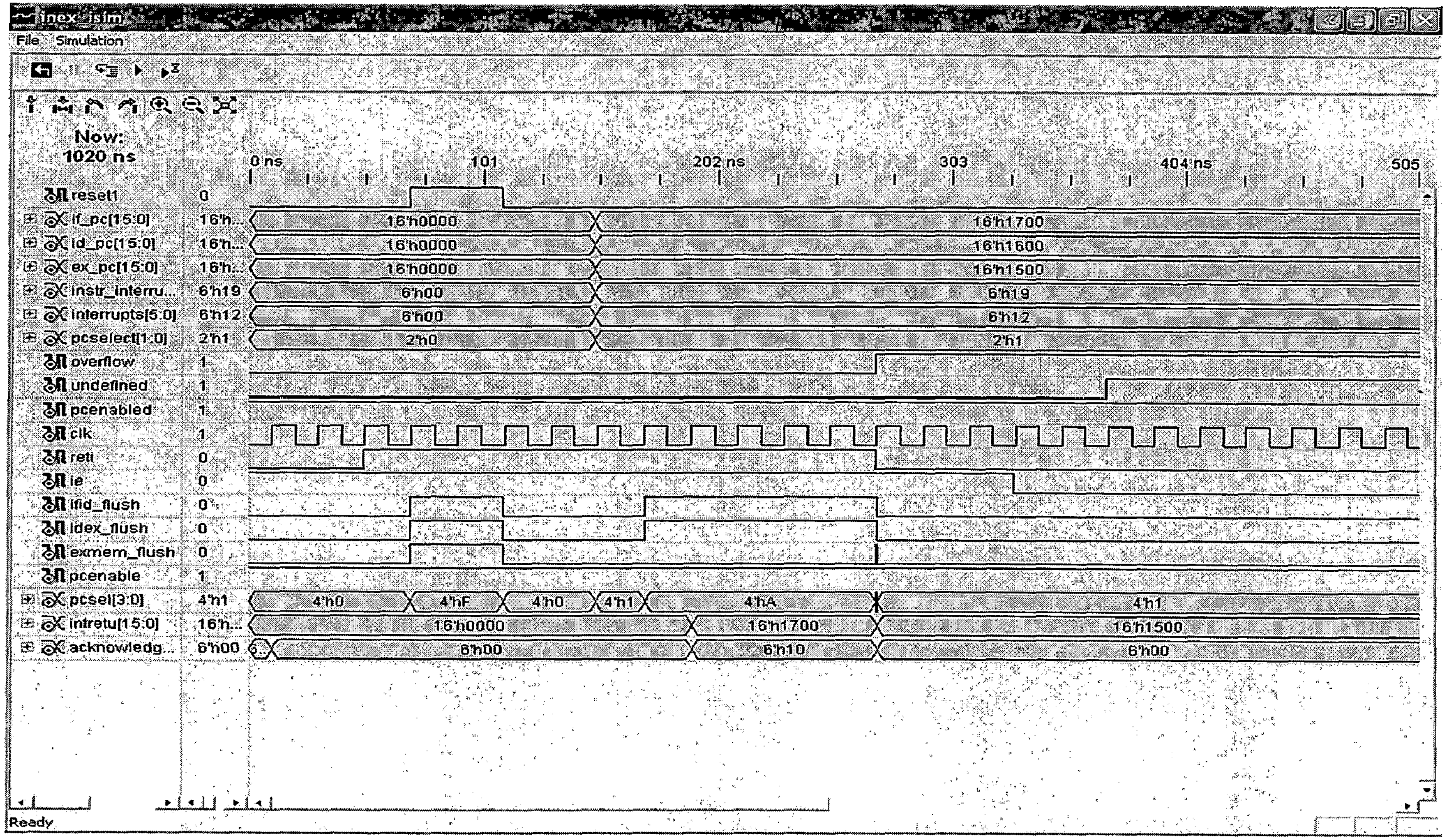

inex_isim
File Simulation
Now: 1020 ns
0 ns
101
202 ns
303
404 ns
505
reset1 0
if_pc[15:0] 16'h... 16'h0000 16'h1700
id_pc[15:0] 16'h... 16'h0000 16'h1600
ex_pc[15:0] 16'h... 16'h0000 16'h1500
instr_interru... 6'h19 6'h00 6'h19
interrupts[5:0] 6'h12 6'h00 6'h12
pcselect[1:0] 2'h1 2'h0 2'h1
overflow 1
undefined 1
pcenabled 1
clk 1
reti 0
ie 0
ifid_flush 0
idex_flush 0
exmem_flush 0
pcenable 1
pcsel[3:0] 4'h1 4'h0 4'hF 4'h0 4'h1 4'hA 4'h1
intretu[15:0] 16'h... 16'h0000 16'h1700 16'h1500
acknowledg... 6'h00 6'h00 6'h10 6'h00
Ready

Fig. 7.8 Simulation Result of Interrupt and Exception Unit

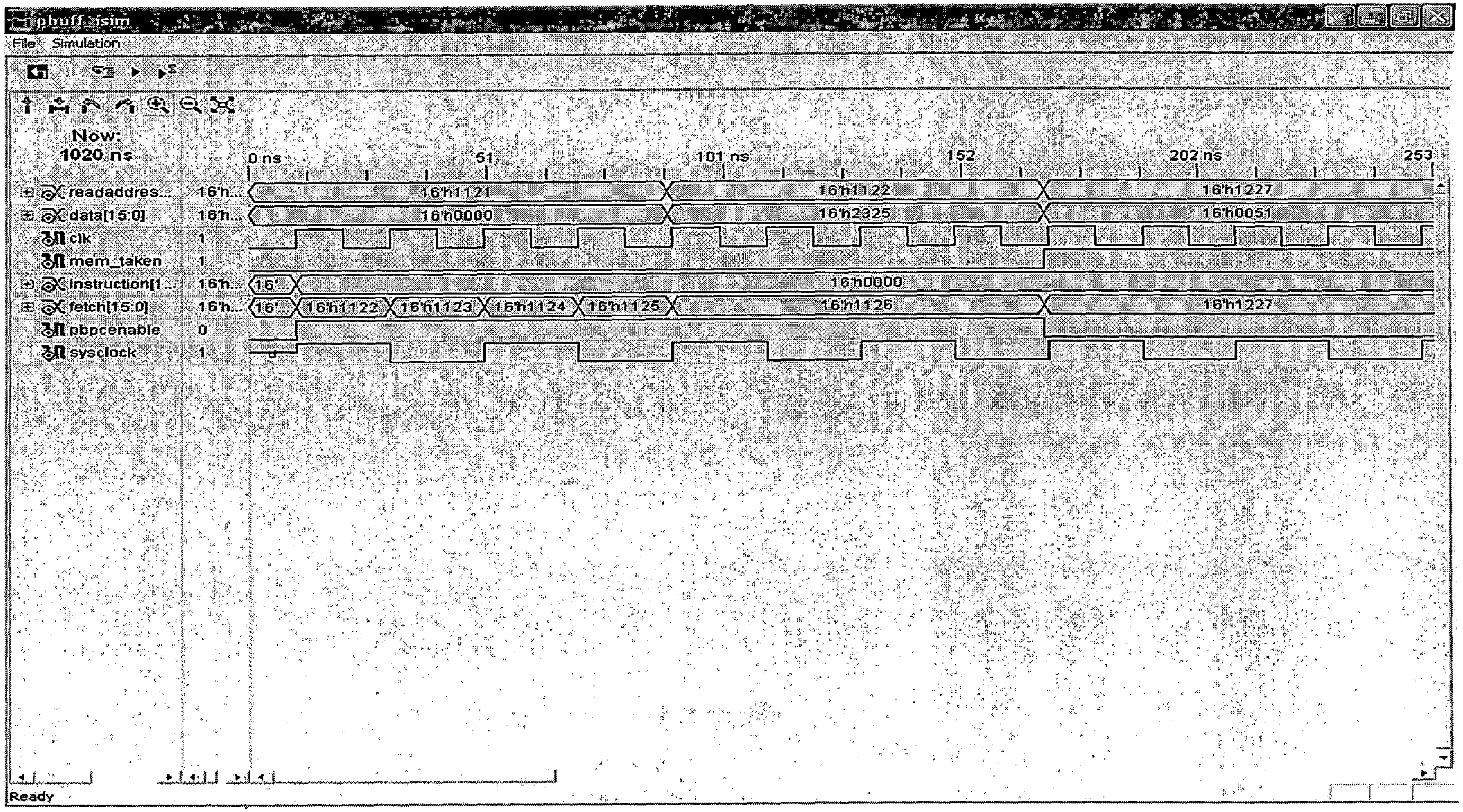

pbuff.isim
File Simulation
Now:
1020 ns
0 ns
51
101 ns
152
202 ns
253
readaddres... 16'h... 16'h1121 16'h1122 16'h1227
data[15:0] 16'h... 16'h0000 16'h2325 16'h0051
clk 1
mem_taken 1
instruction[1... 16'h... 16'... 16'h0000
fetch[15:0] 16'h... 16'... 16'h1122 16'h1123 16'h1124 16'h1125 16'h1126 16'h1227
pbpcenable 0
sysclock 1
Ready

Fig. 7.9 Simulation Result of Prefetch Unit

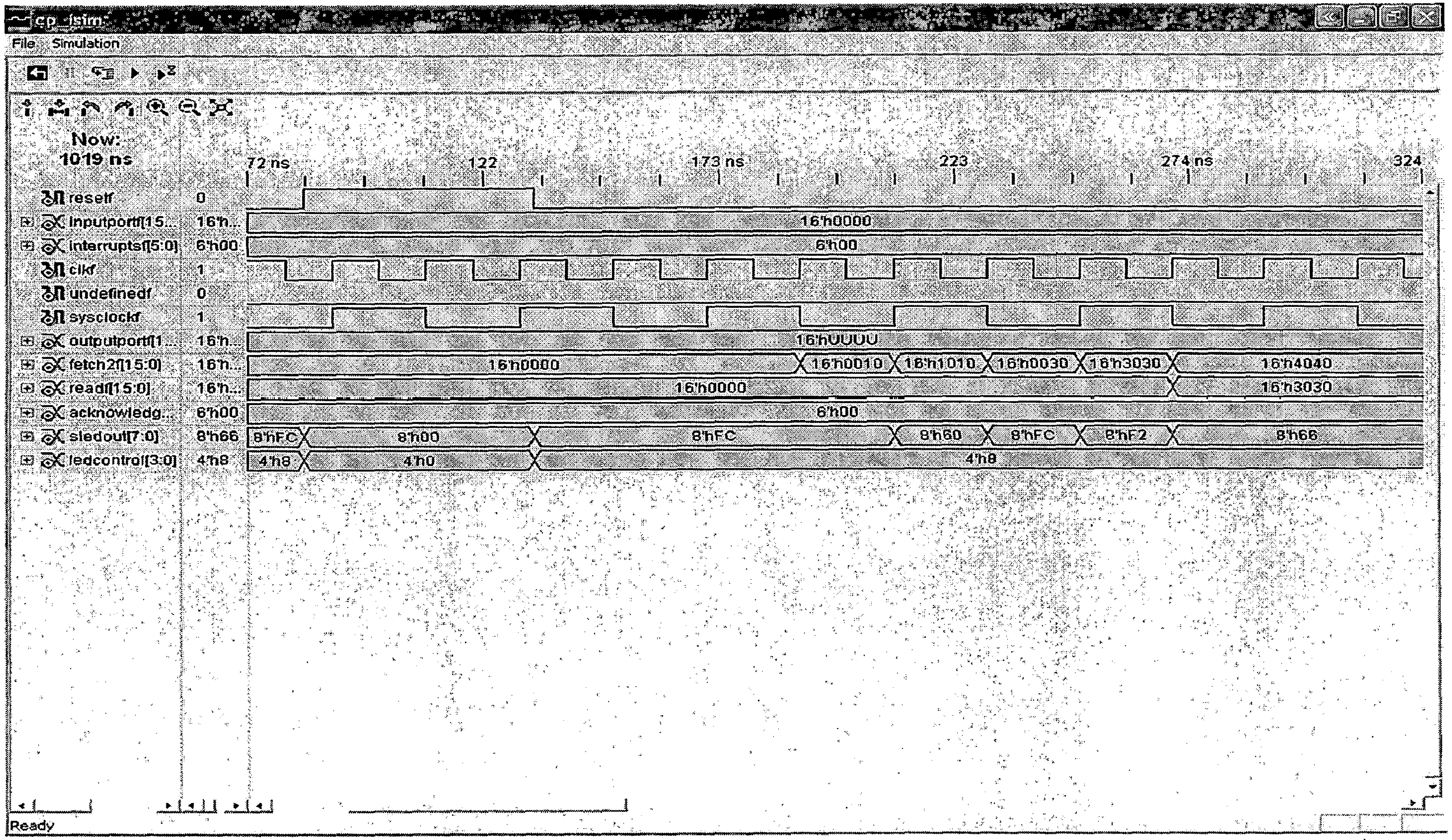

cp_isim
File Simulation
Now:
1019 ns
72 ns
122
173 ns
223
274 ns
324
resetf 0
inputportf[15... 16'h... 16'h0000
interruptsf[5:0] 6'h00 6'h00
clkf 1
undefinedf 0
sysclockf 1
outputportf[1... 16'h... 16'hUUUU
fetch2f[15:0] 16'h... 16'h0000 16'h0010 16'h1010 16'h0030 16'h3030 16'h4040
readf[15:0] 16'h... 16'h0000 16'h3030
acknowledg... 6'h00 6'h00
sledoutf[7:0] 8'h66 8'hFC 8'h00 8'hFC 8'h60 8'hFC 8'hF2 8'h66
ledcontrolf[3:0] 4'h8 4'h8 4'h0 4'h8
Ready

Fig. 7.10 Simulation Result of Designed RISC Processor Connected with Memory

Chapter 8
FUTURE SCOPE

The emphasis is placed on only implementing the common features of RISC processor in this work. Other aspects like device utilization on FPGA, power consumption can be taken into consideration and methods can be found out for their minimization. The work can be extended to

- Development of a 32-bit RISC processor having the features like protection for data, support for virtual memory etc..
- Development of separate hardware for memory.
- Development of full cache memory subsystem inside the processor.
- Implementation of branch prediction logic.

REFRENCES

1. Sivarama P. Dandamudi, "Fundamentals of Computer Organization and Design", Springer-Verlog, New York, Inc., 2003.
2. John L. Hennessy and David A. Patterson, "Computer Architecture a Quantitative Approach", 3rd Edition, Morgan Kaufmann Publishers, Inc., 2000.
3. William Stallings, "Computer Organization and Architecture: Designing for Performance", New Delhi: Prentice Hall of India, 2000.
4. Morris M. Mano, "Computer System Architecture", New Delhi : Prentice Hall of India, 1989.
5. Holzmann D.J. and Mayer U., "RISC and ASIC-technologies of the nineties", CompEuro 1989, VLSI and Computer Peripherals. VLSI and Microelectronic Applications in Intelligent Peripherals and their Interconnection Networks', Proceedings, pp. 148-150.
6. Xiao Li, Longwei Ji, Bo Shen, Wenhong Li and Qianling Zhang, "VLSI Implementation of a High-Performance 32-Bit RISC Microprocessor", Communications, Circuits and Systems and West Sino Expositions, IEEE 2002, International Conference, pp. 1458-1461, vol.2.
7. Smyth N., McLoone M. and McCanny J.V., "Reconfigurable Cryptographic RISC Microprocessor", VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium, pp. 29-32.
8. Wicks J.A., Jr. and Martin H.L., "Design of a Fault-Tolerant RISC Microprocessor Using VHDL", System Theory, 1991, Proceedings, Twenty-Third Southeastern Symposium, pp. 354-358.
9. Dolle M. and Schlett M., "A Cost-Effective RISC/DSP Microprocessor for Embedded Systems", Micro, IEEE, 1995, pp. 32-40.
10. Seung Ho Lee, Beoyng Yoon Choi and Moon Key Lee, "ASIC Implementation of a RISC Microprocessor for Portable Workstation", TENCON '95. 1995 IEEE Region 10 International Conference on Microelectronics and VLSI, pp. 484-487.
11. Bailey D.H., "RISC Microprocessors and Scientific computing", Supercomputing '93. Proceedings, IEEE, 1993, pp. 645-654.

12. Choquette J., Gupta M., McCarthy D. and Veenstra J., "High Performance RISC Microprocessors", Micro, IEEE, 1999, pp. 48-55.
13. Becker J.E., Bieser C., Thomas A., Muller-Glaser K.D. and Becker J., "Hardware/Software Co-training by FPGA/ASIC Synthesis and Programming of a RISC Microprocessor-core", Microelectronic Systems Education, 2003. Proceedings. 2003 IEEE International Conference, pp. 134-135.
14. Peter J. Ashenden, "The Designer's Guide to VHDL", 2nd Edition, Morgan Kaufmann Publishers, 2003.
15. J.Bhaskar, "VHDL Primer", 3rd Edition, Pearson Education Asia, 2001.
16. Douglas L. Perry, "VHDL Programming by Example", 4th Edition, Tata McGraw Hill, 2002.
17. Ben Cohen, Kluwer, "VHDL Coding Styles and Methodologies", 2nd Edition, Academic Publishers, 2000.
18. Stephen M. Triberger., "Filed Programmable Gate Array Technology", Kluwer Academic Publishers.
19. Xilinx Application Note, Spartan-II Series and Xilinx ISE 7.1i Design Environment, 2001.
20. Spartan-II Platform FPGA Handbook October 24, 2002.

APPENDICES

Appendix A
DESIGN FLOW

A.1 Introduction

This chapter describes the design flow used to create complex FPGA and ASIC devices. The designer starts with a design specification, creates RTL description, verifies that description, synthesizes the description to gates, uses place and route tools to implement the design in the chip and then verifies that the final result is correct in terms of function and timing. The design flow is shown in fig. A.1.

A.2 Specification

All designs should start with a detailed specification of the exact tasks the application should do and include details on how fast tasks must de completed.

A.3 Design Entry

In general design entry would done through any hardware description language (HDL) such as VHDL or Verilog. In this thesis, VHDL is used for design entry. One of the best uses of VHDL today is to synthesis ASIC and FPGA devices.

A.4 Simulator

Simulation is the representation of the structure and behavior of a digital logic system through the use of computer. A simulator interpret the HDL description and produces readable output, such as timing diagram, that predicts how the hardware will behave before it is actually fabricated. Simulation allows the detection of functional errors in a design without having to physically create the circuit. The stimulus that tests the functionality of the design is called a test bench. Thus, to simulate a digital system, the design is first described in HDL and then verified by simulating the design and checking it with a test bench, which is also written in HDL.

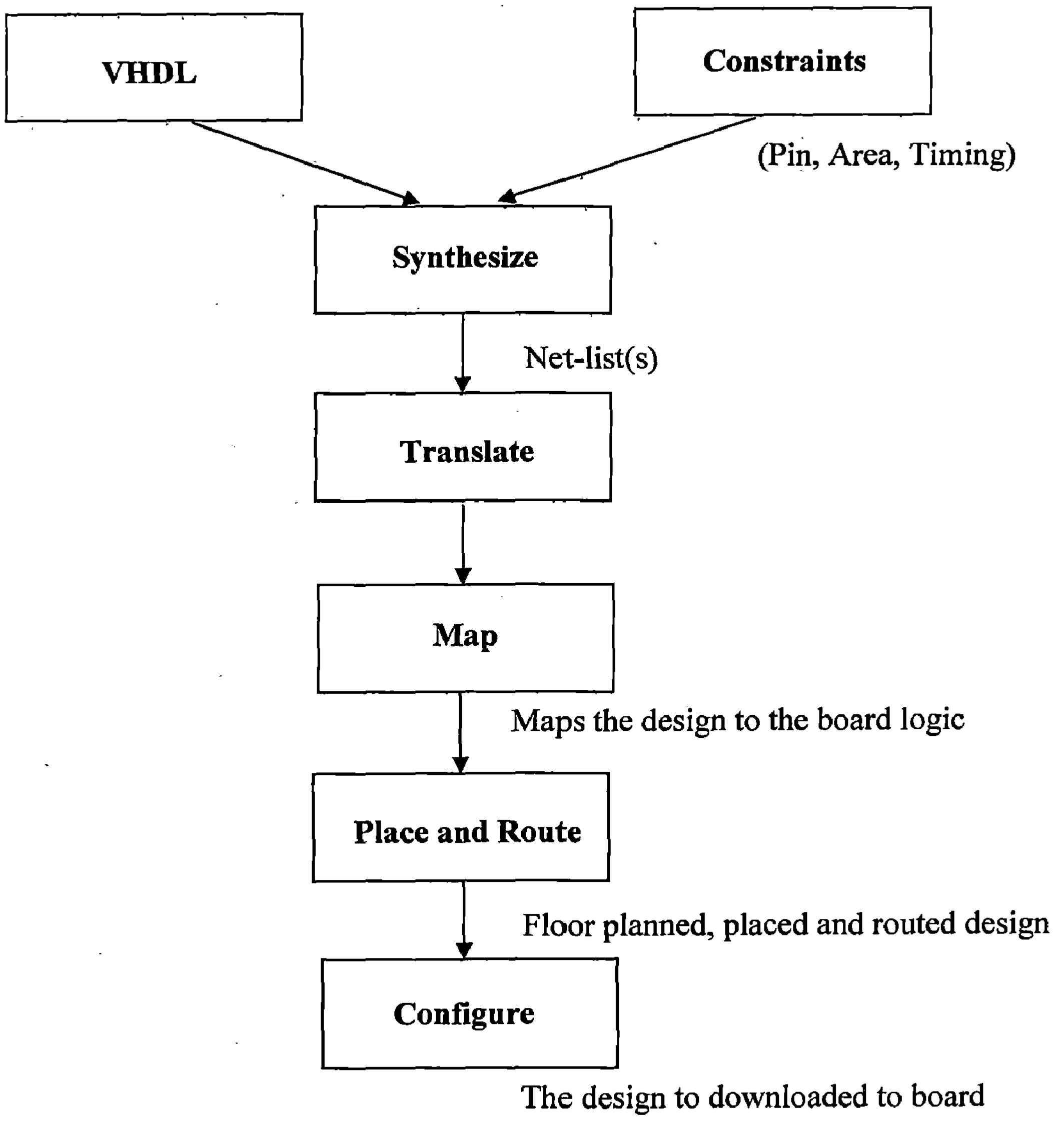

VHDL
Constraints
(Pin, Area, Timing)
Synthesize
Net-list(s)
Translate
Map
Maps the design to the board logic
Place and Route
Floor planned, placed and routed design
Configure
The design to downloaded to board

Fig. A.1 The High-Level Design Flow

A.5 User Constrain File

The UCF file maps signals in VHDL code to pins on the FPGA board. The signal name in your .vhd file must match the net name in the UCF file. If the names do not match, change the name in your .vhd file, not the net name in the .UCF file. This UCF file and .vhd files are the input to the synthesis process.

A.6 Synthesis

After the hardware has been written, simulated and debugged, it needs to be synthesized. In some cases, rewriting the hardware description will be necessary to make the hardware partitions synthesizable. If any code is rewritten, the hardware must be simulated again to make sure it still meets the requirements of the specifications.

Synthesis is an automatic method of converting a higher level of abstraction to a lower level of abstraction. There are several synthesis tools available currently. In this thesis, ISE tool which is provided by Xilinx was used for synthesis.

The current synthesis tool converts the Register Transfer Level (RTL) descriptions to gate level netlists. These gate level netlists consists of interconnected gate level macro cells. Models for the gate level cells are contained in technology libraries for each type of technology supported. The netlists, which are generated from synthesis tool, are device independent, so its contents do not depend on the particulars of the FPGA. It is usually stored in a standard format called the Electronic Design Interchange Format (EDIF) [19].

A.7 Implementation

In the Design Implementation stage, the netlist produced by the design entry program is converted into the bitstream file which configures the FPGA. The first step Maps the design onto the FPGA resources; the second step Places or assign logic blocks created in the mapping process in specific locations in the FPGA. The third step Routes the interconnect paths between the logic blocks. The output is a Logic Cell Array File (LCA) for the particular FPGA; this process is explained in detail in section A.7. This LCA file is then converted into a bitstream file for configuring the FPGA [19].

5.8 Place and Route

Place and route tools are used to take the design netlist and implement the design in the target technology device. The place and route tools place each primitive from the netlist into an appropriate location on the target device and then route signals between the primitives to connect the device according to the netlist. Place and route tools are typically very architecture and device dependent. These tools are tuned to take advantage of each architectural and routing advantage the device contains. FPGA vendors provide these tools because the differences in architectures are large enough that writing a common tool for all architectures would be very difficult. Fig. A.2 shows a dataflow diagram of the place and route [19].

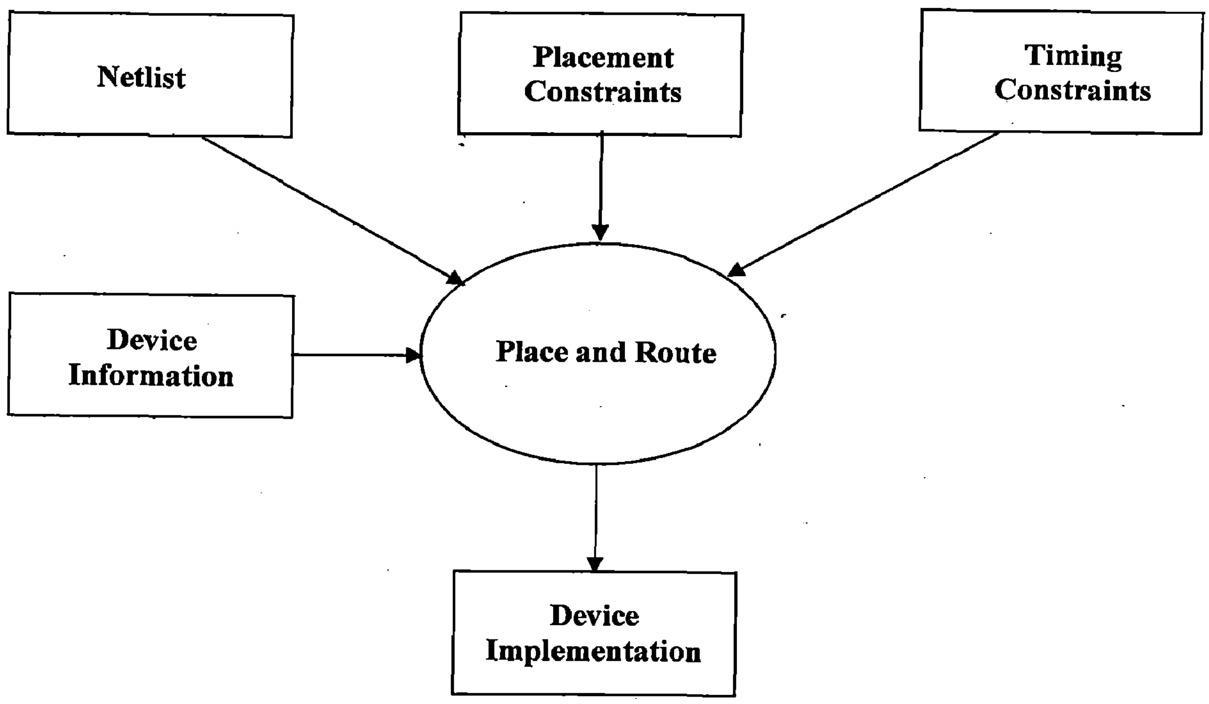


Fig. A.2 Place and Route Data Flow

Input to the place and route tools are the netlist in EDIF or another netlist format and possibly timing constraints. The format of the netlist input file varies from manufacture to manufacturer. Some tools use EDIF [19].

Another input to some place and route tools is the timing constraints, which give the place and route tools an indication about which signals have critical timing associated with them and to route these nets in the most timing efficient manner. These nets are typically identified during the static timing analysis process during synthesis. These constraints tell the place and route toll to place the primitives in close proximity to one another and to use the fastest routing. The closer the cells are, the shorter the routed signals will be and the shorter the time delay [19].

Some place and route tools allow the designer to specify the placement of large parts of the design. This process is also known as floorplanning. Floorplanning allow the user to pick locations on the chip for large blocks of the design so that routing wires are as short as possible. The designer lays out blocks on the chip as general areas. The floorplanner feeds this information to the place and route tools so that these blocks are placed properly. After the cells are placed, the router makes the appropriate connections.

After all the cells are placed and routed, the output of the place and route tools consists of data files that can be used to implement the chip. In the case of FPGAs, these files describe all of the connections needed to make the FPGA macro cells implement the functionality required. Antifuse FPGAs use this information to burn the appropriate fuses, while reprogrammable devices downloaded this information to the device to turn on the appropriate transistor connections.

The other output from the place and route software is a file used to generate the timing file. This file describes the actual timing of the programmed FPGA device or the final ASIC device. This timing file, as much as possible, describes the timing extracted from the device when it is plugged into the system for testing. The most common format of this file for most simulators is SDF (Standard Delay Format). Sometimes, proprietary formats are generated and later translated to SDF. SDF is used to back-annotate the post route timing information from place and route tools into the post layout timing simulation.

A.9 FPGA Configuration

Configuration is a process in which the circuit design (bitstream file) is downloaded into the FPGA. The method of configuring the FPGA determines the type of bitstream file. FPGAs can be configures by PROM. The serial PROM is the most common. The FPGA can either actively read its configuration data out of external serial or byte parallel PROM (master mode), or the configuration data can be written into the FPGA (slave and peripheral mode).

Appendix B

INSTRUCTION SET

Sr. No	Instruction	Type	Opcode	Description
1.	NOP	R-type	0 0 0 0 x x x x x x x x x 0 0 0	No operation. It does nothing.
2.	ADD RD, RS1, RS2	R-type	0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 0 1	Signed addition. RD = RS1 + RS2
3.	ADDu RD, RS1, RS2	R-type	0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 0	Unsigned addition. RD = RS1 + RS2
4.	SUB RD, RS1, RS2	R-type	0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 1	Signed subtraction. RD = RS1 - RS2
5.	SUBu RD, RS1, RS2	R-type	0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 0 0	Unsigned subtraction. RD = RS1 - RS2
6.	SLT RD, RS1, RS2	R-type	0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 0 1	Signed set less than. RD = 0 if RS1< RS2
7.	SLTu RD, RS1, RS2	R-type	0 0 0 0 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 1 0	Unsigned set less than. RD = 0 if RS1< RS2
8.	NOT RD, RS	R-type	0 0 0 0 RD RD RD RS RS RS 1 1 1	Logical NOT. RD = NOT (RS)
9.	AND RD, RS1, RS2	R-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 0 0	Logical AND. RD = RS1 AND RS2
10.	OR RD, RS1, RS2	R-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 0 1	Logical OR. RD = RS1 OR RS2
11.	XOR RD, RS1, RS2	R-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 0	Logical XOR. RD = RS1 XOR RS2
12.	NOR RD, RS1, RS2	R-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 0 1 1	Logical NOR. RD = RS1 NOR RS2
13.	SLL RD, RS1, RS2	S-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 0 0	Logic shift left. RD = RS1 shifted by RS2
14.	SRL RD, RS1, RS2	S-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 0 1	Logic shift right. RD = RS1 shifted by RS2
15.	SRA RD, RS1, RS2	S-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 1 0	Arithmetic shift right. RD = RS1 shifted by RS2
16.	ROR RD, RS1, RS2	S-type	0 0 0 1 RD RD RD RS1 RS1 RS1 RS2 RS2 RS2 1 1 1	Rotated right. RD = RS1 rotated by RS2
17.	IN RD	R-type	0 0 1 0 RD RD RD x x x x x x 0 0 0	RD = data on input port

18.	OUT RS	R-type	0 0 1 0 RS RS RS x x x x x x 0 0 1	Output port = data in register RS
19.	BZ RD, RS	R-type	0 0 1 0 RD RD RD RS RS RS x x x 0 1 0	Branch on zero. If RS = 0 jump to loc RD
20.	BNZ RD, RS	R-type	0 0 1 0 RD RD RD RS RS RS x x x 0 1 1	Branch not zero. If RS /= 0 jump to loc RD
21.	EI D6	I-type	0 0 1 0 D6 D6 D6 D6 D6 D6 x x x 1 0 0	Enable interrupt whose value is 1 in D6
22.	JAL RD, RS	R-type	0 0 1 1 RD RD RD RS RS RS x x x 0 0 0	Jump to loc RS and link back to loc RD
23.	RJAL RD	R-type	0 0 1 1 RD RD RD x x x x x x 0 0 1	Link back to loc RD
24.	RETI	R-type	0 0 1 1 x x x x x x x x x 0 1 0	Return from interrupt to loc in TRAP register
25.	MVIL RD, D8	I-type	0 1 0 0 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 0	Move D8 in lower byte of RD
26.	MVIH RD, D8	I-type	0 1 0 0 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 1	Move D8 in upper byte of RD
27.	BZI RD, D8	I-type	0 1 0 1 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 0	If RD = 0, jump to loc = (PC) + D8
28.	BNZI RD, D8	I-type	0 1 0 1 RD RD RD D8 D8 D8 D8 D8 D8 D8 D8 1	If RD /= 0, jump to loc = (PC) + D8
29.	SLLI RD, RS, D5	SI-type	0 1 1 1 RD RD RD RS RS RS D5 D5 D5 D5 D5 0	RD = RS shifted left logically by D5
30.	SRLI RD, RS, D5	SI-type	0 1 1 1 RD RD RD RS RS RS D5 D5 D5 D5 D5 1	RD = RS shifted right logically by D5
31.	SRAI RD, RS, D5	SI-type	1 0 0 0 RD RD RD RS RS RS D5 D5 D5 D5 D5 0	RD = RS shifted right arithmetic by D5
32.	RORI RD, RS, D5	SI-type	1 0 0 0 RD RD RD RS RS RS D5 D5 D5 D5 D5 1	RD = RS rotated right by D5
33.	ADDI RD, RS, D6	RI-type	1 0 0 1 RD RD RD RS RS RS D6 D6 D6 D6 D6 D6	Signed addition. RD = RS + D6
34.	SUBI RD, RS, D6	RI-type	1 0 1 0 RD RD RD RS RS RS D6 D6 D6 D6 D6 D6	Signed subtraction. RD = RS - D6
35.	LW RD, RS, D6	RI-type	1 0 1 1 RD RD RD RS RS RS D6 D6 D6 D6 D6 D6	RD = data from memory loc (RS + D6)
36.	SW RD, RS, D6	RI-type	1 1 0 0 RD RD RD RS RS RS D6 D6 D6 D6 D6 D6	Loc (RS + D6) in memory = RD
37.	HLT	R-type	1 1 0 1 x x x x x x x x x x x x	Halt the program execution.

Appendix C

SOFTWARE CODE FOR RISC PROCESSOR

The CD given along with this report contains the software code for all the units in RISC processor. The coding is done in VHDL and Xilinx ISE 7.1i is used as a synthesis tool. Xilinx ISE simulator is used for simulation.