A
Project Report
on
"16-bit Microprocessor using Verilog"

Prepared by

Soham Patel (16EC085)

Under the guidance of

Prof. Hitesh N Patel

Submitted to

Charotar University of Science & Technology

for Partial Fulfillment of the Requirements for the

Degree of Bachelor of Technology

in Electronics & Communication

EC448 - Project

of 8th Semester of B.Tech

Submitted at

CHARUSAT

CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF ELECTRONICS & COMMUNICATION

Faculty of Technology & Engineering, CHARUSAT

Chandubhai S. Patel Institute of Technology

At: Changa, Dist: Anand – 388421

April 2020

CHARUSAT
CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

CERTIFICATE

This is to certify that the report entitled **“16-bit Microprocessor using Verilog”** is a bonafide work carried out by **Soham Patel (16EC085)** under the guidance and supervision of **Prof. Hitesh N Patel** for the subject **Project (EC448)** of 8th Semester of Bachelor of Technology in Electronics & Communication at Faculty of Technology & Engineering (C.S.P.I.T.) – CHARUSAT, Gujarat.

To the best of my knowledge and belief, this work embodies the work of candidate himself, has duly been completed, and fulfills the requirement of the ordinance relating to the B.Tech. Degree of the University and is up to the standard in respect of content, presentation and language for being referred to the examiner.

Under the supervision of,

Prof. Hitesh N Patel
Assistant Professor
Department of Electronics & Communication,
C.S.P.I.T., CHARUSAT-Changa. Gujarat.

Dr. Trushit K. Upadhyaya
Head of Department,
Department of Electronics & Communication
C.S.P.I.T., CHARUSAT- Changa, Gujarat.

Chandubhai S Patel Institute of Technology (C.S.P.I.T.)

Faculty of Technology & Engineering, CHARUSAT

At: Changa, Ta. Petlad, Dist. Anand, Gujarat - 388421

TABLE OF CONTENTS

Abstract……………………………………………………………..…………..….I

Acknowledgement……………………………………………….…………II

List of figures……………………….………………………………………III

List of tables………………………………………………….………..…..V

Chapter 1 Introduction……………………………………………..………1

1.1 Overview…………………………………………..…………..……………..1

1.1.1 What is microprocessor…………………………………..……………1

1.1.2 Classification of microprocessor……………………………..……………..1

1.1.3 Working of microprocessor……………………………….……….………...3

1.2 About Project…………………………………………………..……..………..4

1.2.1 Tools and languages……………………………………….…………..…..4

1.2.2 Specifications……………………………………………….…………4

1.2.3 Instruction Table……………………………………..………………...5

Chapter 2 Components…………………………………..….………6

2.1 Program Counter (PC)…………………………..……………..………6

2.1.1 Explanation and pin diagram…….……………………………….…….……..6

2.1.2 Test-bench results……………………..……………………….……….…7

2.2 Instruction Register (IR)…………………………………….……….…...7

2.2.1 Explanation and pin diagram of registers……..………………….……….…..7

2.2.2 Explanation and pin diagram of IR…….……...…………….………….…..7

2.2.3 Test-bench results…………………..…………………………………………8

2.3 Memory………………………………………………………………………...9

2.3.1 Explanation…………………………………………………………………9

2.3.2 Test-bench results……………………………………………………………..9

2.4 Arithmetic and Logic Unit (ALU)…...……………………………….………9

2.4.1 Significance and pin diagram………………………………………………9

2.4.2 Functions and modes of ALU……………………………………………...10

2.4.3 Test-bench results……………………………………………………………..12

2.5 General Purpose Registers (GPR)……………………………………………12

2.5.1 Explanation and pin diagram………………………………………………12

2.5.2 Test-bench results…………………………………………………………….13

2.6 Flag Register…………………………………………………..…….…………14

2.6.1 Explanation and role of each bit…………………………………………...14

2.7 Decoder / Control Unit……………………………………………….……….15

2.7.1 Significance and pin diagram……………………………………….……..15

Chapter 3 Testing instructions……………………………….…………………..16

3.1 Test program 1………………………..……………………………………….16

3.1.1 Program……………………………………………………………………….16

3.1.2 Equivalent machine code……………………………………………………...17

3.1.3 Simulator results……………………………………………………………17

3.1.4 Explanation…………………………………………………………………17

3.2 Test program 2……………………………..…………………………………18

3.2.1 Program…………………………………………………………………….18

3.2.2 Equivalent machine code…………………………………………………...18

3.2.3 Simulator results……………………………………………………………19

3.2.4 Explanation…………………………………………………………………19

3.3 Test program 3……………………………….…………………………….20

3.3.1 Program…………………………………………………………………….20

3.3.2 Equivalent machine code…………………………………………………...20

3.3.3 Simulator results……………………………………………………………20

3.3.4 Explanation…………………………………………………………………21

3.4 Test program 4……………………………………..………………………21

3.4.1 Program…………………………………………………………………….21

3.4.2 Equivalent machine code…………………………………………………...22

3.4.3 Simulator results……………………………………………………………22

3.4.4 Explanation…………………………………………………………………23

Chapter 4 Limitations and future enhancement…………..…...………..24

Chapter 5 Conclusion and discussion……………………………………25

References…………………………………….……………………….26

Appendix……………………………………………………………..27

ABSTRACT

In today’s world, the demand of low power computing, gaming, extreme graphics, powerful processing is growing rapidly. All such cannot be possible without processor. Microprocessor forms the heart of every embedded or computer system. It can be general purpose or specific purpose processor. The project is aimed at developing 16-bit general purpose microprocessor using verilog HDL. It supports load, store, arithmetic and logic commands/instructions.

ACKNOWLEDGEMENT

I would like to show my sincere gratitude to Prof. Hitesh N Patel for his invaluable support and providing precious knowledge about vlsi domain. He constantly motivated me regarding future aspects and job opportunities for the same. I consider myself very lucky to get the opportunity to learn and work under Hitesh sir and his guidance proved to be very fruitful for my major project.

Soham Patel (16EC085)

LIST OF FIGURES

Fig 1.1 Basic block diagram 1

Fig 1.2 RISC vs CISC 3

Fig 2.1 PC Block 6

Fig 2.2 PC Test 7

Fig 2.3 Register Block 7

Fig 2.4 IR Block 8

Fig 2.5 Register Test 8

Fig 2.6 IR Test 8

Fig 2.7 Memory Test 9

Fig 2.8 ALU Block 10

Fig 2.9 ALU Arithmetic Test 12

Fig 2.10 ALU Logic Test 12

Fig 2.11 GPR Block 13

Fig 2.12 GPR Test (Part 1) 13

Fig 2.13 GPR Test (Part 2) 13

Fig 2.14 Flag Register Block 14

Fig 2.15 Decoder Block 15

Fig 3.1 Test 1 (Part 1) 17

Fig 3.2 Test 1 (Part 2) 17

Fig 3.3	Test 2 (Part 1)	19
Fig 3.4	Test 2 (Part 2)	19
Fig 3.5	Test 3 (Part 1)	20
Fig 3.6	Test 3 (Part 2)	21
Fig 3.7	Test 4 (Part 1)	22
Fig 3.8	Test 4 (Part 2)	23

LIST OF TABLES

Table 1.1 Instruction table 5

CHAPTER 1 : INTRODUCTION

1.1 OVERVIEW

1.1.1 What is microprocessor ?

A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions and provide results as output.

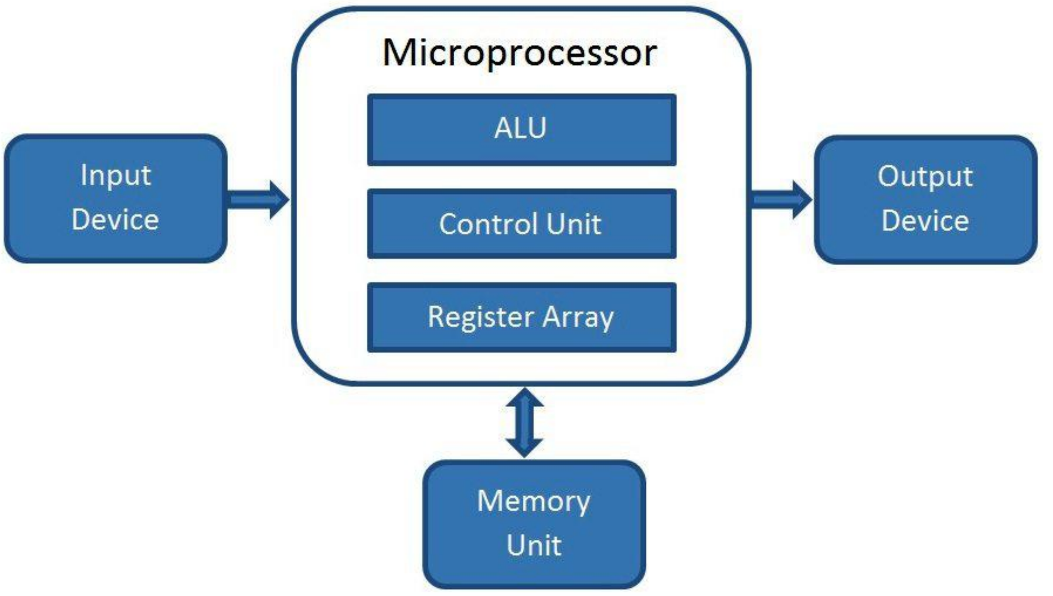

Fig 1.1 Basic block diagram

1.1.2 Classification of microprocessors

There are 2 types of processors: RISC and CISC.

RISC is a type of microprocessor architecture which uses small, general purpose and highly optimized instruction set rather than more specialized set of instructions found in others. RISC offers high performance over its opposing architecture CISC (see below). In a processor, execution of each instruction require a special circuit to load and process the data. So by reducing instructions, the processor will be using simple circuits and faster in operation.

- Simple instruction set
- Larger program
- Consists of large number of registers
- Simple processor circuitry (small number of transistors)
- More RAM usage
- Fixed length instructions
- Simple addressing modes
- Usually fixed number of clock cycles for executing one instruction

CISC is the opposing microprocessor architecture for RISC. It is made to reduce the number of instructions per program, ignoring the number of cycles per instruction. So complex instructions are directly made into hardware making the processor complex and slower in operation.This architecture is actually designed to reduce the cost of memory by reducing the program length.

- Complex instruction set
- Smaller program
- Less number of registers
- Complex processor circuitry (more number of transistors)
- Little RAM usage
- Variable length instructions
- Variety of addressing modes
- Variable number of clock cycles for each instructions

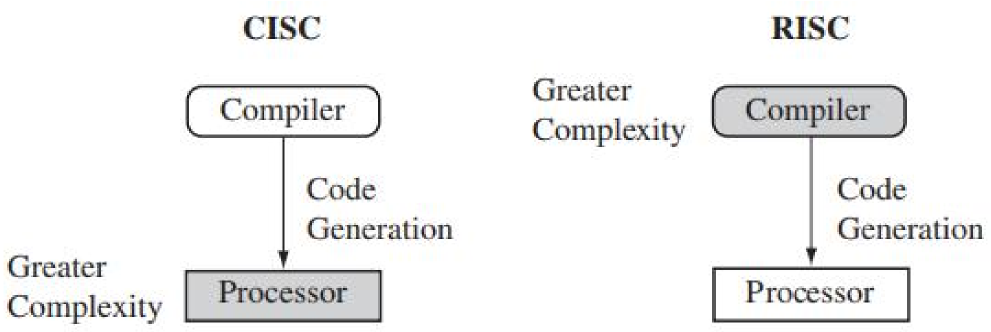

Fig 1.2 RISC vs CISC

However, processors today are developed using both concepts.

1.1.3 Basic working of microprocessor

The 3 basic steps of working of processor are: Fetch (F), Decode (D) and Execute (E). Due to fixed instruction size and cycles, pipeline method is used in RISC based processors. CISC processors cannot implement the same due to different number of clock cycles for each instructions as well as variable size of instruction size. The processor developed under this project implements the same ideology of pipelining. Pipeline speeds up the execution by fetching the new instruction while other instructions are decoded and executed.

Fetch (F) - loads instruction from memory

Decode (D) - Identifies the instruction to be executed

Execute (E) - Processes the instruction

Without pipeline (3 instructions, 9 cycles , 9/3 = 3 cycles/instruction) :-

```
F D E _ _ _ _ _ _
_ _ _ F D E _ _ _
_ _ _ _ _ _ F D E
```

With pipeline (3 instructions, 5 cycles , 5/3 = 1.67 cycles/instruction) :-

```
F D E _ _
_ F D E _
_ _ F D E
```

1.2 ABOUT PROJECT

1.2.1 Tools and languages used

The software used is Xilinx ISE Design suite and the entire project is developed using Verilog HDL. The result of the entire project is based on verilog simulations in ISIM simulator.

1.2.2 Specifications of processor

- 16-bit processor (16-bit ALU, registers, program counter, address and data bus).
- Harvard architecture memory organization (separate program and data memory).
- 16-bit instruction set.
- 18 executable instructions.
- Fully compatible signed number operation by ALU.
- Interface 64k data memory
- 2 Cycle operation of each instruction with pipelining
- 8 general purpose registers

1.2.3 Instruction Table

Table 1.1 Instruction table

Sr. No	Instruction Type	Instruction	1st Cycle		2nd Cycle	Information
			1st Byte (reg)	2nd Byte (opcode)	2 Bytes (data/addr)	
1	Load / Store operations	LDM	00 to 07	01	Address	Loads data from memory address to register
2		LDR	00 to 07 (source)	02	00 00 to 07 (other)	Loads data from source register to other register
3		LDV	00 to 07	03	Data	Loads data to register
4		STR	00 to 07	04	Address	Stores data from register to memory address
5	Arithmetic operations	ADR	00	10	0000	Adds data from reg 0 and reg 1
6		ADD	00	11	Data	Adds data to accumulator (reg 0)
7		SBR	00	12	0000	Substracts data in reg 0 from reg 1 (r0 - r1)
8		SUB	00	13	Data	Substracts data in accumulator (r0) from given data
9		INC	00	14	0000	Increments the value of accumulator (r0) by 1
10		DEC	00	15	0000	Decrements the value of accumulator (r0) by 1
11	Logical operations	XOR	00	1f	Data	Performs xor operation on accumulator and given data
12		AND	00	1e	Data	Performs and operation on accumulator and given data
13		OR	00	1d	Data	Performs or operation on accumulator and given data
14		XNOR	00	1c	Data	Performs xnor operation on accumulator and given data
15		NAND	00	1b	Data	Performs nand operation on accumulator and given data
16		NOR	00	1a	Data	Performs nor operation on accumulator and given data
17		INV	00	19	0000	Performs inverter operation on accumulator
18	Other operation	NOP	00	00	0000	Performs no operation
19		REG	00 to 07	ff	0000	Loads data of source register on data bus

The table above mentions all the working instructions of this project. There are total 18 instructions of 4 different types each explained along with opcode and working. Each instruction requires total 2 cycles of 16-bit machine code. This data is necessary to convert machine code from instructions. The same machine code is stored in program memory.

CHAPTER 2 : COMPONENTS

This chapter discusses various smaller components of processor developed in this project and assembled in top module to make this 16-bit processor function. Also significance of each components along with test timing diagrams in simulator are discussed.

2.1 PROGRAM COUNTER (PC)

2.1.1 Explanation and pin diagram

PC is a 16-bit register which contains (program) memory address. PC contains that very memory address from where the next instruction is to be fetched for execution. Suppose the PC contents are 8000H, then it means that the processor desires to fetch the instruction byte at 8000H. After fetching the byte at 8000H, the PC is automatically incremented by 1. This way processor becomes ready to fetch the next byte of the instruction.

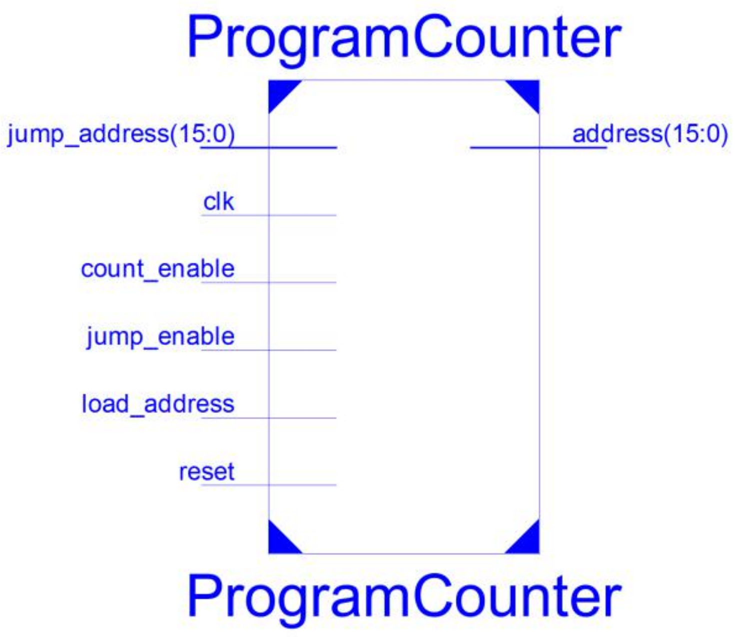

Fig 2.1 PC Block

2.1.2 Test-bench results

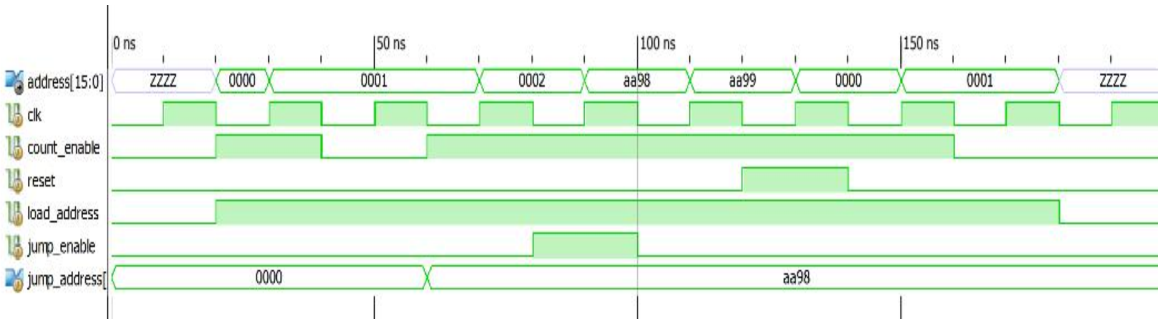

Fig 2.2 PC Test

2.2 INSTRUCTION REGISTER (IR)

2.2.1 Explanation and pin diagram of register

Register is a sequential small building block of any digital system which is used for storing bit/s. It is usually edge triggered.

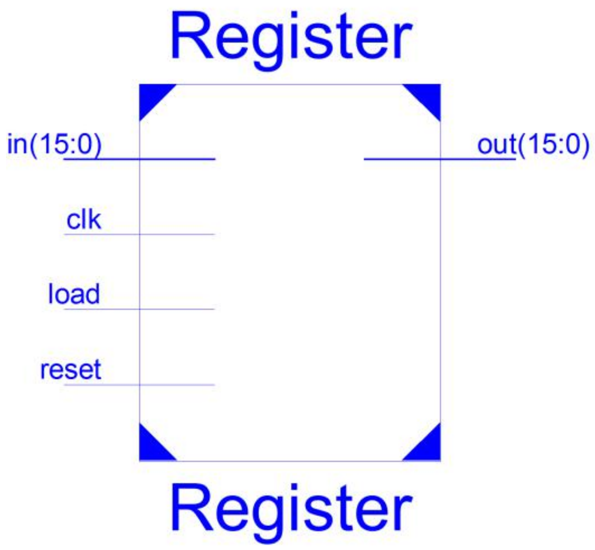

Fig 2.3 Register Block

2.2.2 Explanation and pin diagram of IR

Those registers are used to store machine code from program memory. The IR in this project is 16-bit with 4 stage pre-fetch unit. The main purpose of IR is to store machine code in queue before entering decoder circuit. It holds the instruction currently being executed.

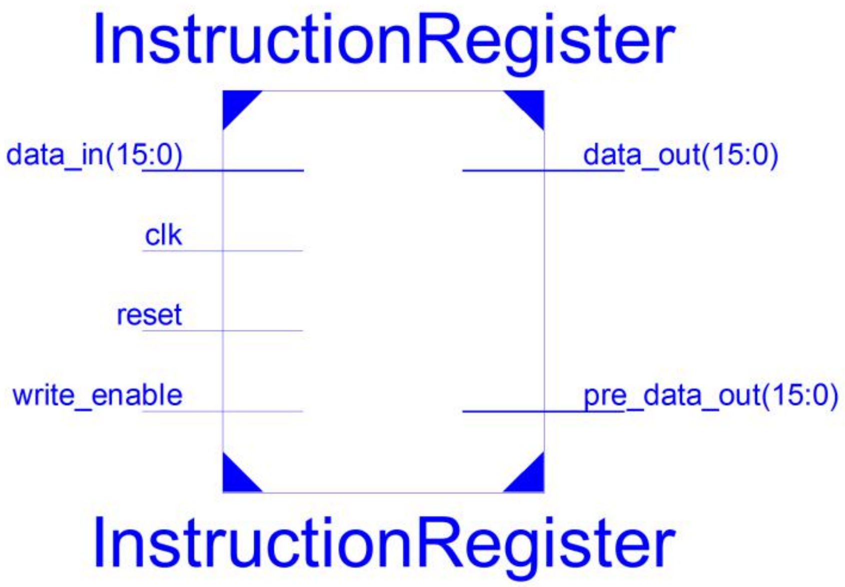

Fig 2.4 IR Block

2.2.3 Test-bench results

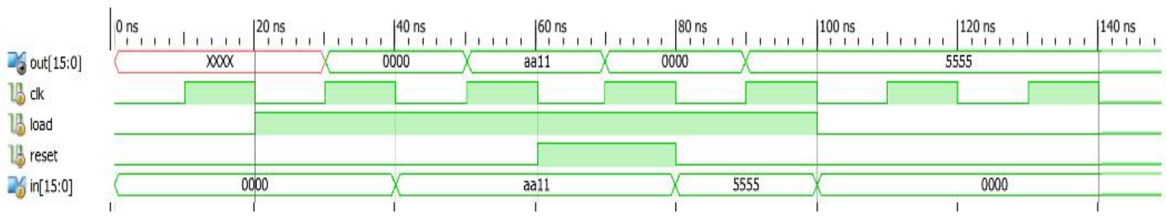

Fig 2.5 Register Test

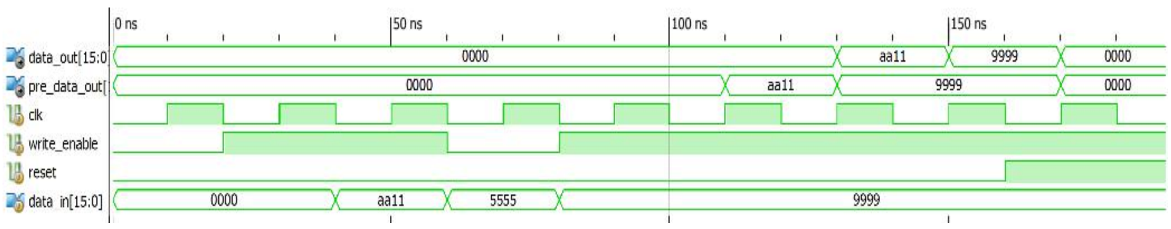

Fig 2.6 IR Test

2.3 MEMORY

2.3.1 Explanation

Memory is used to store data connected to processor. There are 2 types of memory here as the processor is designed on the basis of Harvard Architecture : Program memory and Data memory. Unlike Von-neumann architecture where single memory is used for both purposes, here there are different address and data bus for both memories. This improves performance of processor as data transfer and instruction fetch can be performed simultaneously which helps to use pipeline execution.

2.3.2 Test-bench results

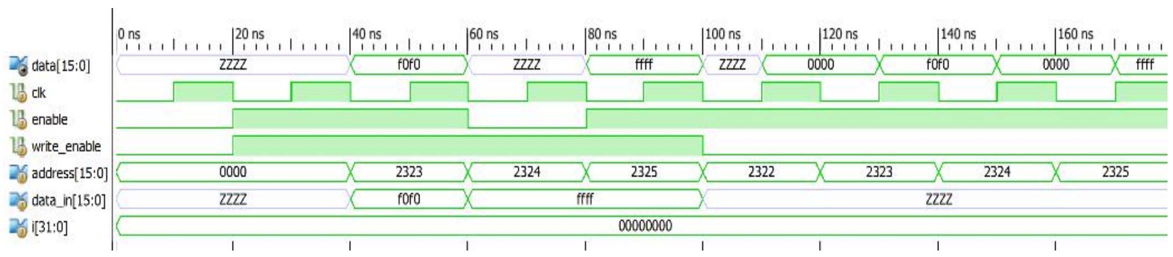

Fig 2.7 Memory Test

2.4 ARITHMETIC AND LOGIC UNIT (ALU)

2.4.1 Significance and pin diagram

An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs contain very powerful and complex ALUs. Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A register is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data and the ALU stores the result in an output register called accumulator.

The ALU developed in this project supports full signed data for any input combinations. The signed magnitude convention is used for input as well as output data where 1 bit

(MSB) is reserved as sign bit. Addition is using carry look-ahead adder and subtraction is using 2's complement method.

ALU generates different signals like :-

Equal - if both inputs are equal

Parity - for even parity in result data (excluding sign bit)

Zero - if result is 16'h0000

Greater_than - if the 1st input is greater than second

Sign - if the result is negative

Carry_out - if result has carry bit

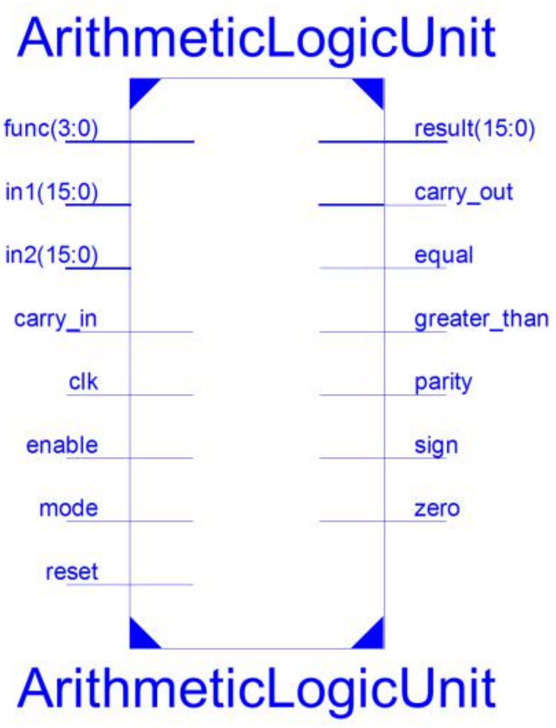

Fig 2.8 ALU Block

2.4.2 Functions and modes of ALU

The ALU performs desired task on the basis of inputs like func (4-bit) and mode (1-bit).

<u>Logic Functions</u> (Mode 0) :-

```
xor_op = 4'h0;

and_op = 4'h1;

or_op = 4'h2;

xnor_op = 4'h3;

nand_op = 4'h4;

nor_op = 4'h5;

high = 4'hf;          // all output lines are logic high

low = 4'he;           // all output lines are logic low

input1 = 4'hf;        // reflect 1st input as output

input2 = 4'he;        // reflect 2nd input as output

invert_input1 = 4'hd; // invert 1st input as output

invert_input2 = 4'hc; // invert 2nd input as output
```

Arithmetic Functions (Mode 1) :-

```
addition = 4'h0;

substraction = 4'h1;

increment = 4'h2;     // increment of 1st input by 1

decrement = 4'h3;     // decrement of ist input by 1
```

2.4.3 Test-bench results

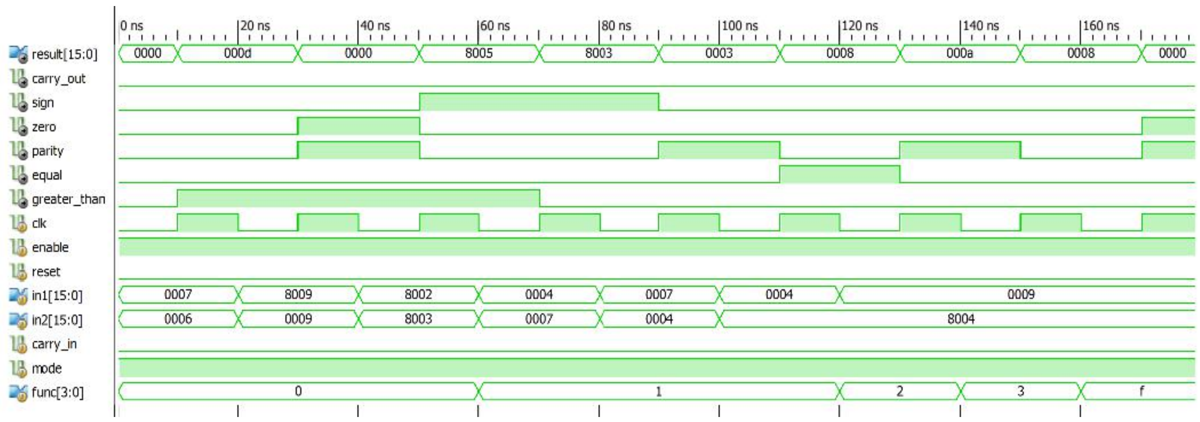

Fig 2.9 ALU Arithmetic Test

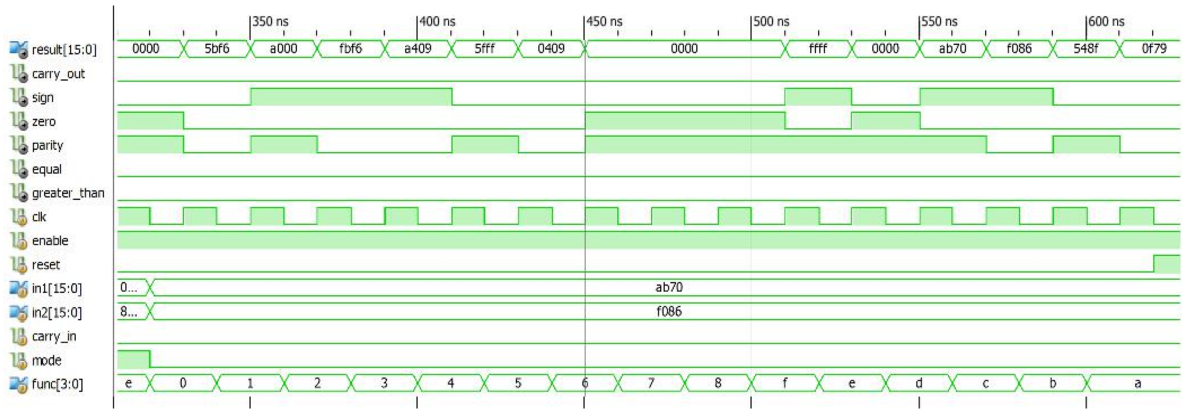

Fig 2.10 ALU Logic Test

2.5 GENERAL PURPOSE REGISTERS (GPR)

2.5.1 Explanation and pin diagram

The general purpose registers are used as storage devices in processors for faster operations. There are 8 16-bit general purpose registers (R0 to R7) where R0 is accumulator and also 1st input to ALU. While R1 is 2nd input to ALU if desired*. The gpr operates in read, write, reset and register transfer mode.

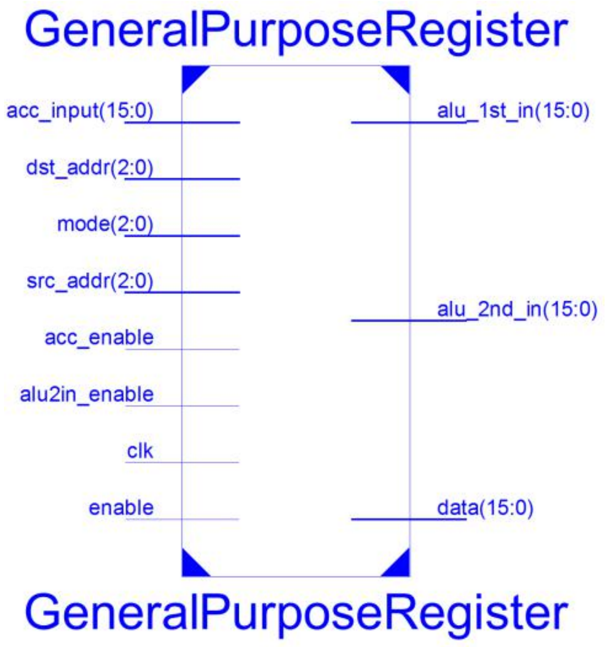

Fig 2.11 GPR Block

2.5.2 Test-bench results

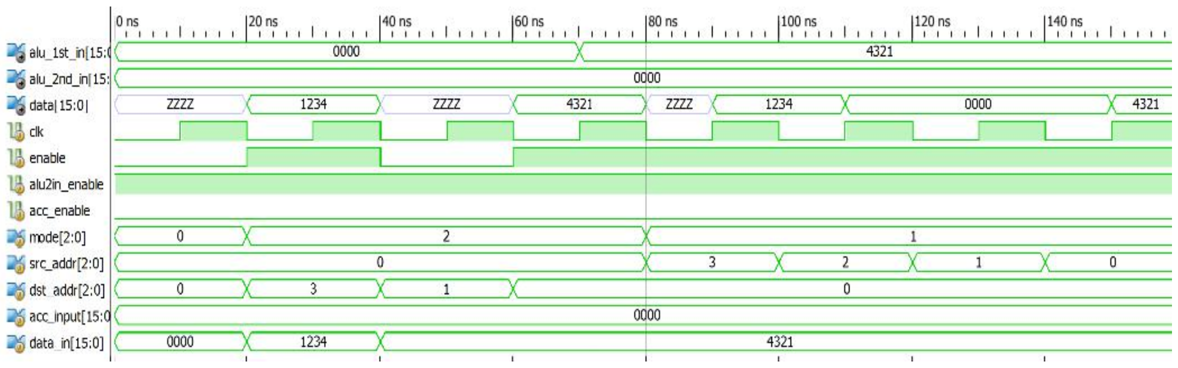

Fig 2.12 GPR Test (Part 1)

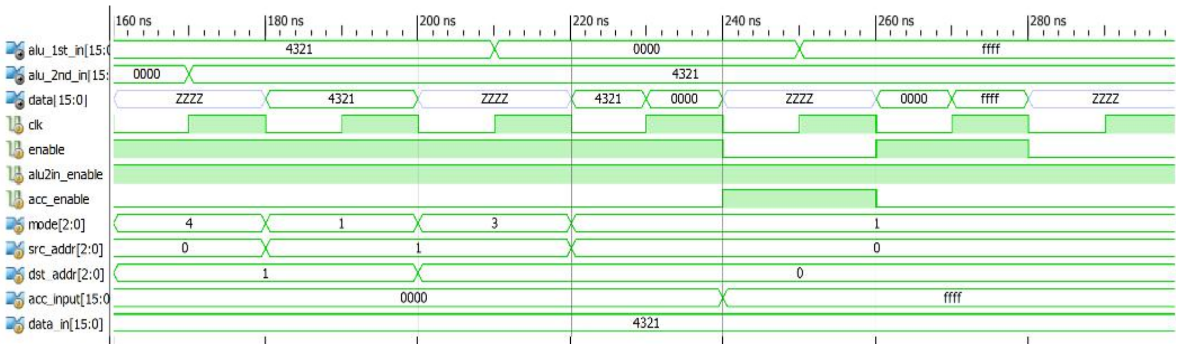

Fig 2.13 GPR Test (Part 2)

2.6 FLAG REGISTER

2.6.1 Explanation and role of each bit

Flag register is based on result of ALU and is updated by ALU itself. It is used for conditional or branch instructions. The flag register of this processor is of 6 bits.

Bit 0 = carry flag

Bit 1 = zero flag

Bit 2 = sign flag

Bit 3 = greater_than flag

Bit 4 = equal flag

Bit 5 = parity (even) flag

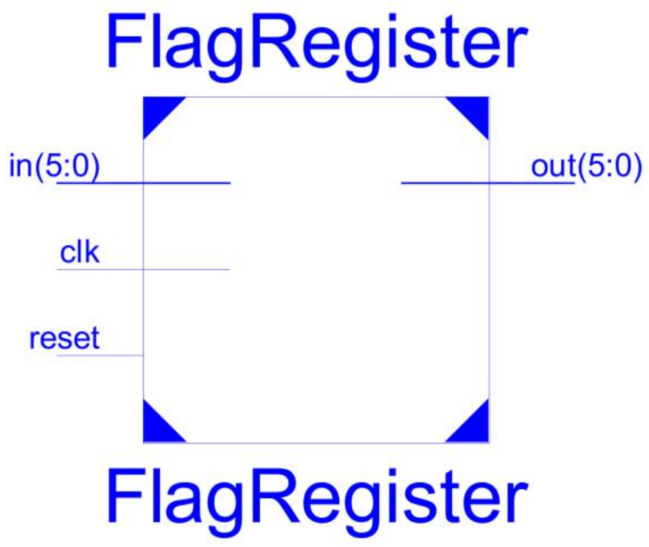

Fig 2.14 Flag Register Block

2.7 DECODER CIRCUIT / CONTROL UNIT

2.7.1 Significance and pin diagram

Decoder circuit decodes the instruction data and generates necessary control signals for execution of instruction. For the same reason, it is also called control unit. It controls working of most of the inner components of processor. It is like a instructor for all components of processor who instructs components to work as per requirement.

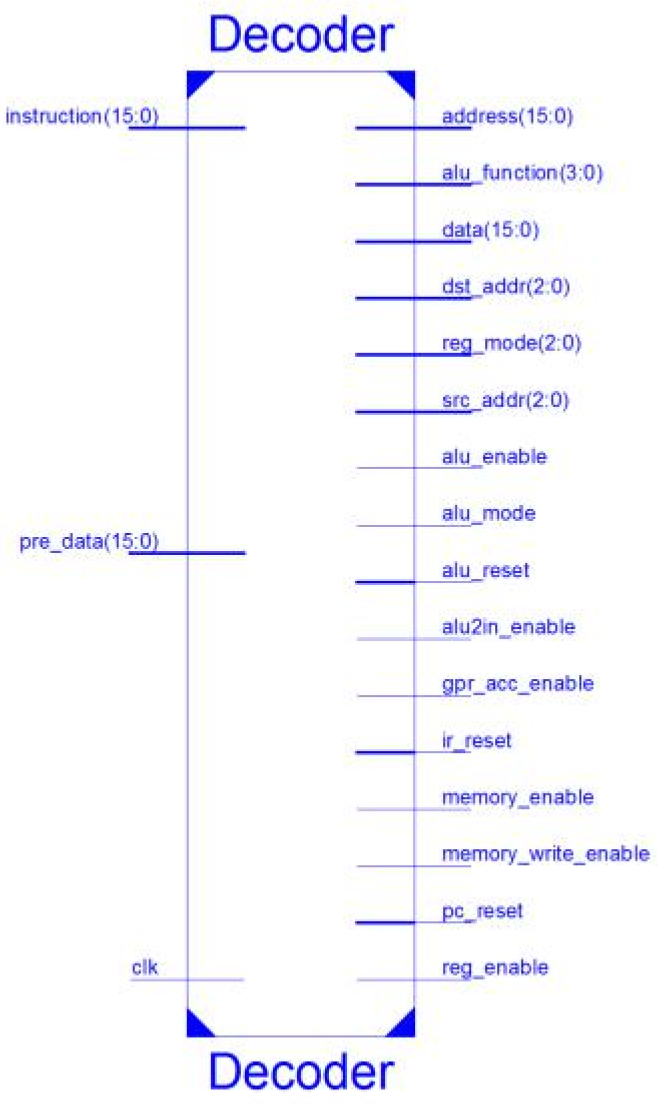

Fig 2.15 Decoder Block

Note:- The flag register and decoder / control units would be tested in next chapter during test of each instructions. So for this reason test-bench results of both components are not mentioned here.

CHAPTER 3 : TESTING INSTRUCTIONS

This chapter contains testing sample programs by manually converting them to machine code and testing working of processor based on given instructions. The testing is entirely on ISIM verilog simulator. The machine code is written in notepad as and the same instructions are mapped to program memory using special task functions in verilog before simulating. By observing timing diagrams we can test whether the processor is working as per instructions of user.

Conversion of machine code from instruction is based on the instruction set table designed for this project. The table can be found in Chapter 1 > About project > Instruction table.

3.1 Test program 1

3.1.1 Program :-

```
NOP

REG R0                   // loads data in register 0 to data bus

REG R1                   // loads data in register 1 to data bus

REG R2                   // loads data in register 2 to data bus

LDV R0, 5555H            // loads 5555 in register 0

STR R0, [2222H]          // stores data in register 0 to memory location 2222

LDM R1, [2222H]          // loads data from memory location 2222 to register 1

NOP

LDR R2, R1               // loads data from register 1 to register 2

NOP

REG R0                   // loads data in register 0 to data bus
REG R1          // loads data in register 1 to data bus

REG R3          // loads data in register 1 to data bus

REG R2          // loads data in register 2 to data bus

NOP
```

3.1.2 Equivalent machine code :-

0000,00ff,0000,01ff,0000,02ff,0000,0003,5555,0004,2222,0101,2222,0000,0102,0002,

0000,00ff,0000,01ff,0000,03ff,0000,02ff,0000,0000

3.1.3 Simulator results :-

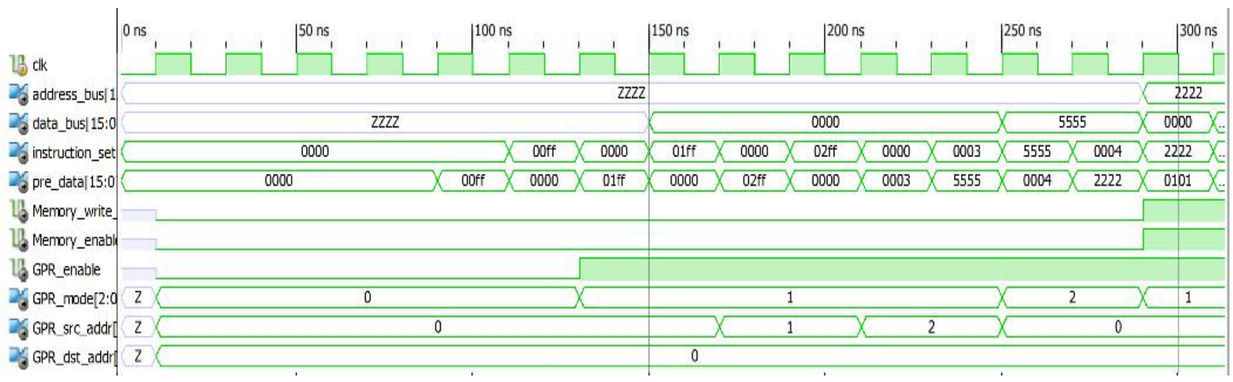

Fig 3.1 Test 1 (Part 1)

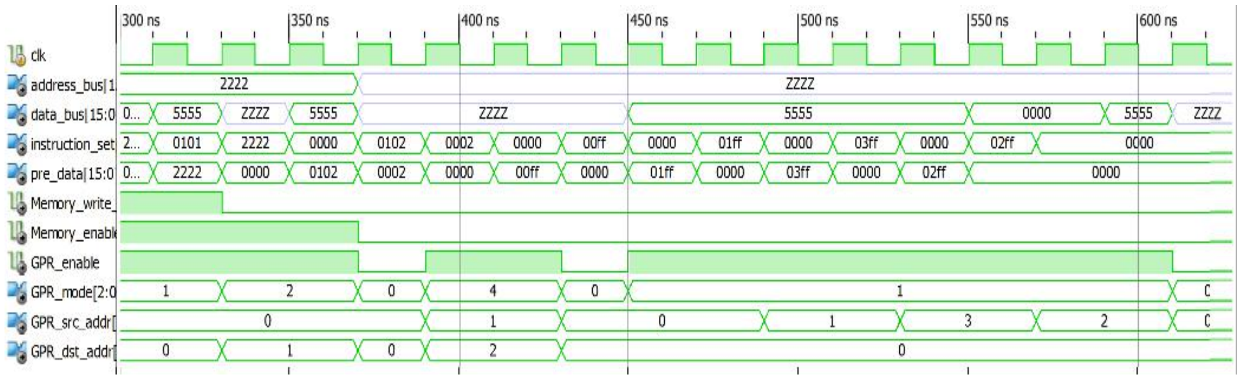

Fig 3.2 Test 1 (Part 2)

3.1.4 Explanation :-

The program first checks input of registers 0, 1 and 2. Then 5555 data is stored in register then stores the same data in memory location 2222. The register 1 retrieves same data from memory location 2222. Register data transfer takes place from register 1 to register 2. So now register 0, 1 and 2 contain same data 5555 which can be verified at the end. But data in register 3 is zero as it was unused.

The program was intended to test memory and register operations.

3.2 Test program 2

3.2.1 Program :-

```
NOP

LDV R0, 8006H     // loads -6 in register 0 (accumulator)

LDV R1, 0006H     // loads 6 in register 1

ADR               // adds reg 1 and reg 0 and stores result in accumulator

ADD 0009H         // adds 9 to reg 0 and stores in accumulator

SBR               // subtracts reg 1 from reg 0

SUB 8004          // subtracts -4 from reg 0

INC               // increments value in accumulator

DEC               // decrements value in accumulator

REG R0            // loads data in reg 0 (accumulator) to data bus

NOP
```

3.2.2 Machine code :-

```
0000,0003,8006,0103,0006,0010,0000,0011,0009,0012,0000,0013,8004,0014,0000,0015

0000,00ff,0000,0000
```

3.2.3 Simulator results :-

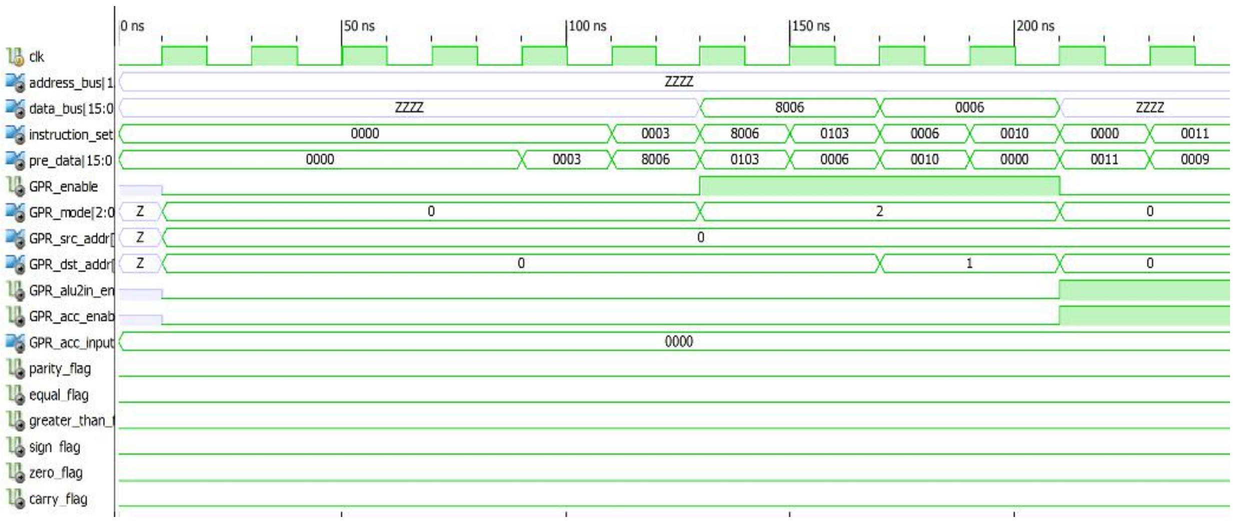

Fig 3.3 Test 2 (Part 1)

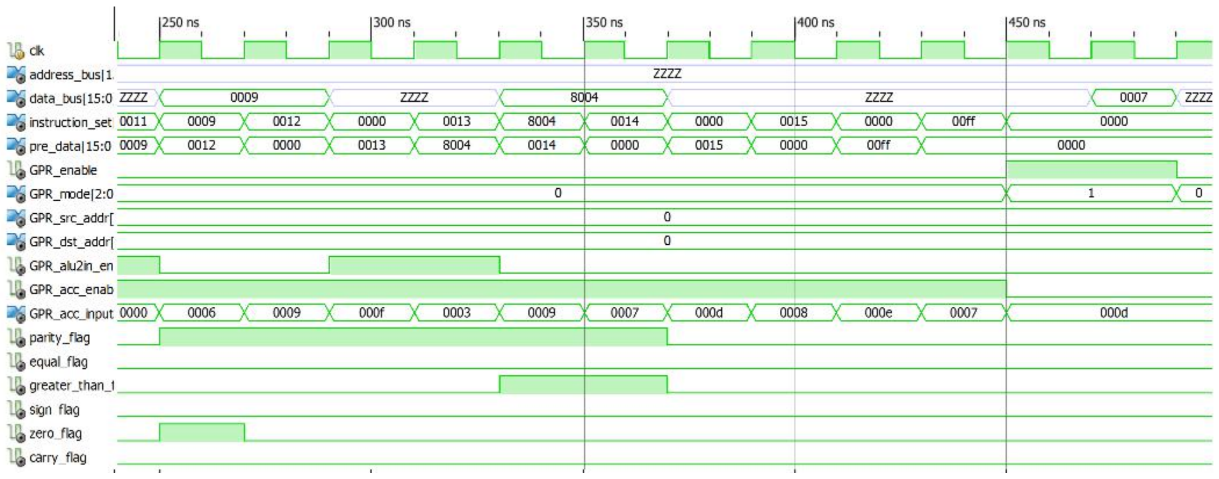

Fig 3.4 Test 2 (Part 2)

3.2.4 Explanation :-

The program begins by loading data -6 and 6 in register 0 and 1 respectively. Then addition of both is performed which results in 0 stored in register 0/ accumulator. Again data in register 0 is added to 9 then subtracted by data in register 1 (6).So now accumulator contains 3. There after data is subtracted with -4 (addition by 4). Then increment and decrement are performed. This results in accumulator storing 7 which can be verified at end.

The program was intended to test arithmetic operations.

3.3 Test program 3

3.3.1 Program :-

NOP

LDV R0, 3A0FH // loads 3a0f in reg 0 (accumulator)

XOR F0F0H // performs xor operation of reg 0 and f0f0

REG R0 // loads data in reg 0 (accumulator) to data bus

AND 8888H // performs and operation of reg 0 and 8888

REG R0 // loads data in reg 0 (accumulator) to data bus

OR 3030H // performs or operation of reg 0 and 3030

REG R0 // loads data in reg 0 (accumulator) to data bus

NOP

3.3.2 Machine code :-

0000,0003,3A0F,001F,F0F0,00FF,0000,001E,8888,00FF,0000,001D,3030,00FF,0000,

0000

3.3.3 Simulator results :-

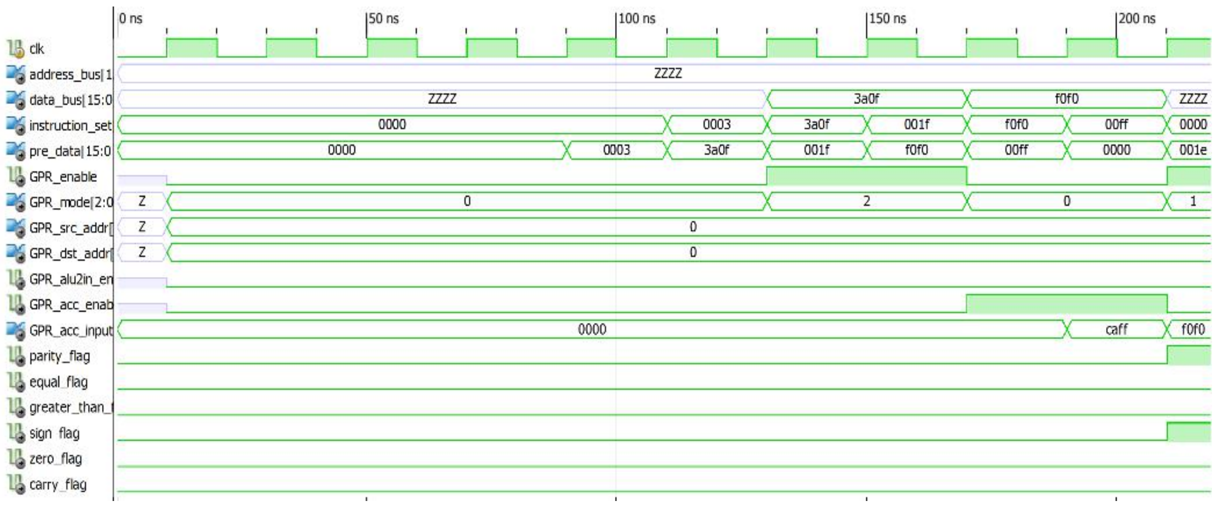

Fig 3.5 Test 3 (Part 1)

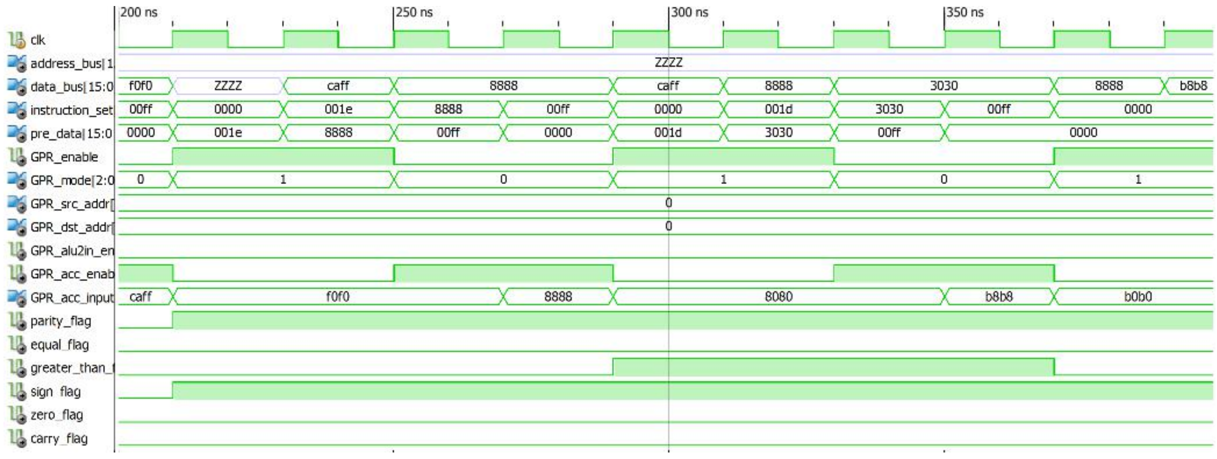

Fig 3.6 Test 3 (Part 2)

3.3.4 Explanation :-

The program stores data in accumulator and performs various logic operations on it which can be verified from simulation. The program was intended to test logic operations.

3.4 Test program 4

3.4.1 Program :-

NOP

LDV R0, FFFFH // loads ffff to reg 0

AND 000AH // performs and operation with reg 0 and 00a

DEC // decrease the value of reg 0 by 1

REG R0 // loads data in reg 0 (accumulator) to data bus

REG R1 // loads data in reg 1 to data bus

LDR R1, R0 // loads data from reg 0 to reg 1

LDM R0, [1111H] // loads data from memory location 1111 to reg 0

SUB // subtracts reg 1 from reg 0

REG R0 // loads data in reg 0 (accumulator) to data bus

REG R1 // loads data in reg 1 to data bus

NOP

3.4.2 Machine code :-

0000,0003,ffff,001e,000a,0015,0000,00ff,0000,01ff,0000,0002,0001,0001,1111,0012

0000,00ff,0000,01ff,0000,0000

3.4.3 Simulator results :-

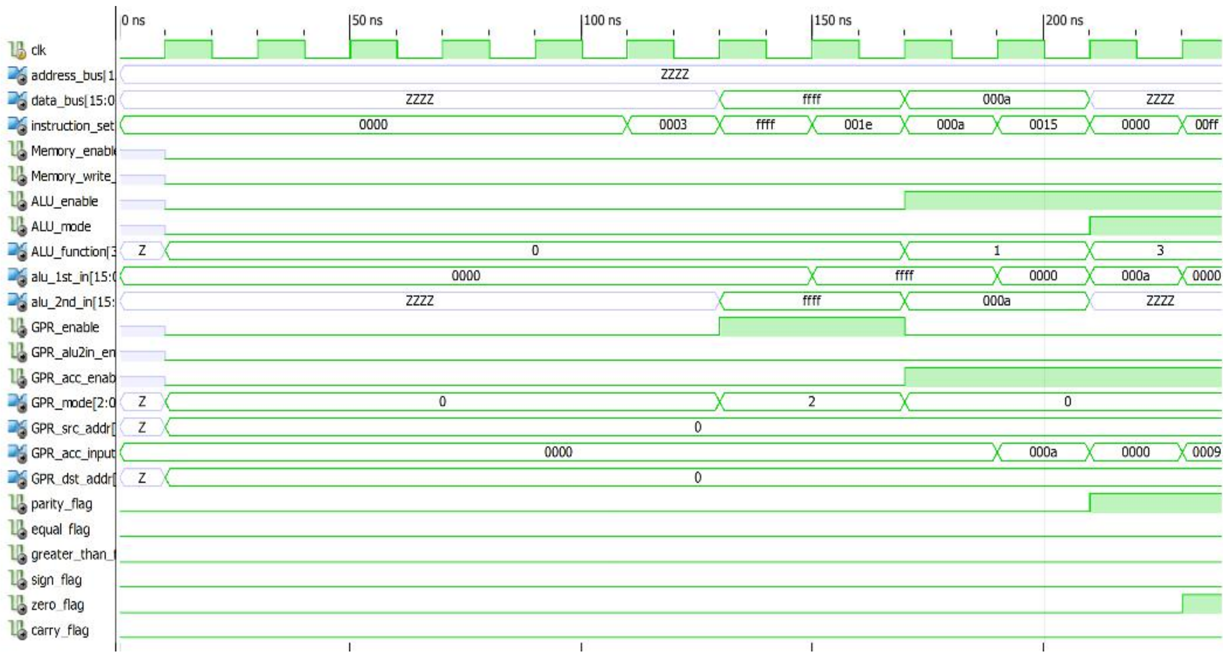

Fig 3.7 Test 4 (Part 1)

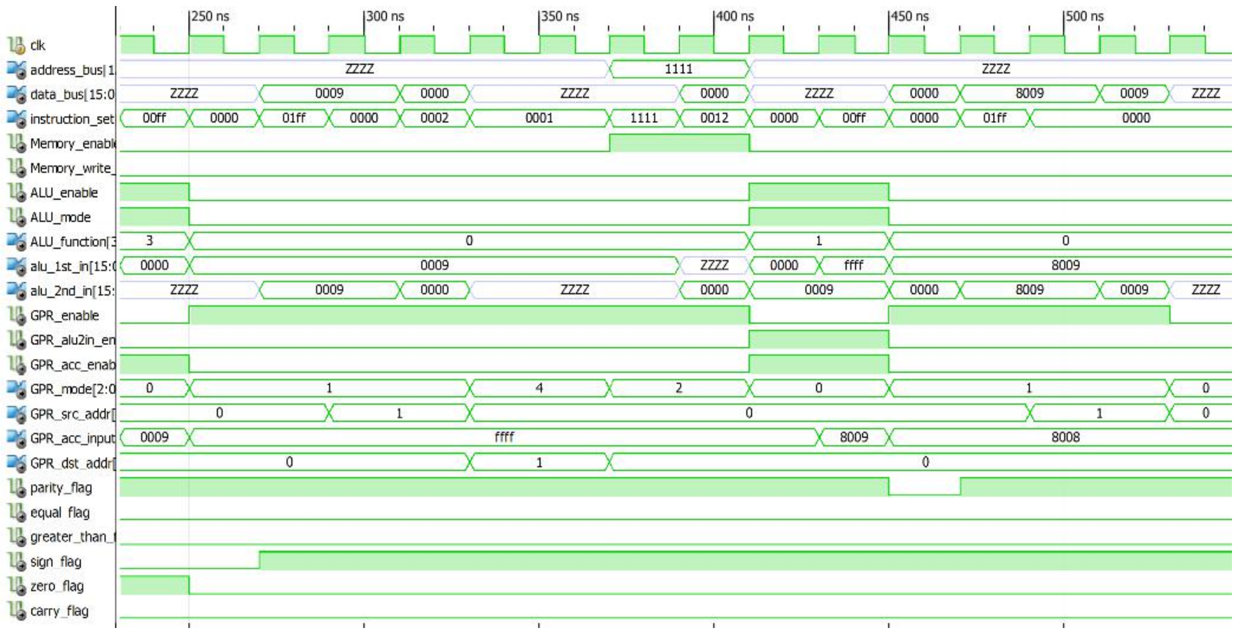

Fig 3.8 Test 4 (Part 2)

3.4.4 Explanation :-

The program starts by loading data in accumulator which turns out to be 000a after and operation. After performing decrement operation the value in accumulator becomes 0009. Then the same data is transferred to register 1 and register 0 loads 0000 in itself from memory. Now register 1 and 0 contains 9 and 0 respectively. So after subtraction accumulator contains -9.

CHAPTER 4 : LIMITATION AND FUTURE ENHANCEMENT

There are several limitations of this 16-bit processor. The processor is only tested on the basis of simulation so hardware performance cannot be predicted (in terms of frequency). Branch instructions are missing in this processor. The memory interface is only limited to 16-bit address line i.e. 64k of memory. Also the processor doesn't support advanced I/O handling and is not capable of data transfer using any protocol. There are several advanced components which are missing to make this processor market ready. The processor doesn't support interrupt handling too. Other unknown limitations of processor may be possible as this is only designed for simulation purpose. The ALU is not very powerful and there are still some bugs in flag signals.

By inclusion of the above mentioned features/components the processor can be made even more powerful and market ready. Additional DSP functions, ethernet functions, audio - video functions, etc. can be included for application specific use of processor.

CHAPTER 5 : CONCLUSION AND DISCUSSION

This project clears many glitches still left in mind on topics like microprocessor and digital electronics. The project helped to improve ability to program in verilog HDL. Overall the project also taught patience as some errors took days to rectify and improvise. There are stages when one needs to experiment to make outcome happen. This project is proof of my imagination turning to reality. This project made me realize that processor development is a very vast and every growing field in the field of engineering.

Overall, this project gave great opportunity to learn and improve technical skills and proved to be great decision to dive into and create.

REFERENCES

1) Andrew Sloss, Dominic Symes, Chris Wright. *ARM System Developer's Guide.* Morgan Kaufmann Publishers

2) Ben Eater *youtube channel*. Available at *https://www.youtube.com/channel/UCS0N5baNlQWJCUrhCEo8WlA*

3) ChipVerify *www.chipverify.com*

4) Douglas V Hall, SSSP Rao. *Microprocessor and Interfacing 3rd edition.* Mc Graw Hill Publication

5) FPGA4Student *www.fpga4student.com*

6) GeekforGeeks *www.geeksforgeeks.org*

7) Sivakumar. *ASIC Design Flow , Maven Silicon*

APPENDIX

Verilog Codes

1. **Program Counter :-**

```
module ProgramCounter (clk, count_enable, reset, load_address,
jump_enable, jump_address, address);


// 16 - bit Program Counter

// Works on rising edge of clock

// Every input signal is active high


/* Inputs */

// clk regulates working of program counter

// count_enable increments the address value by 1 location

// reset resets counter to zero

// load_address loads address on instruction address bus
( Harvard Architecture )

// jump_enable enables counter to take value of jump address as
input

// jump_address bus takes address value to jump


/* Outputs */

// address bus points value of next instruction to be executed
from program memory


parameter word_size = 16;

parameter high_impedance = 16'hzzzz;

parameter zero = 16'h0000;


input clk, count_enable, reset, load_address, jump_enable;
input [word_size-1 : 0] jump_address;


output [word_size-1 : 0] address;


reg [word_size-1 : 0] temp_address;

reg jump_flag;


// Initializing reg variables

initial begin

      temp_address = zero;

      jump_flag = 1'b0;

end


// If load_address is low then output address bus is high
impedance

assign address = (load_address === 1'b1) ? temp_address :
high_impedance;


always @ (posedge clk) begin

      case ({reset, jump_enable, count_enable})

            3'b000 : begin

                  temp_address = temp_address;

            end

            3'b001 : begin

                  temp_address = temp_address + 1;

            end

            3'b010 : begin

                  temp_address = jump_address;

            end
```

```
        3'b011 : begin
            if (address === jump_address)
                temp_address = temp_address + 1;
            // else load jump address first then increment it
            else
                temp_address = jump_address;
        end
        3'b100 : begin
            temp_address = zero;
        end
        3'b101 : begin
            temp_address = zero;
        end
        3'b110 : begin
            temp_address = zero;
        end
        3'b111 : begin
            temp_address = zero;
        end
    endcase
end
endmodule
```

2. Memory :-

```
module Memory (clk, enable, write_enable, address, data);


// 65536 * 16 = 64k * 16 bits Memory

// Works on rising edge of clock

// Every input signal is active high


/* Inputs */

// clk regulates working of memory

// enable signal to control working of memory

// write_enable to decide reading or writing operation

// address provides location number to read/write data


/* Inout Port */

// data Bus to read/write data from/to memory


parameter word_size = 16;              // n Bits

parameter memory_locations = 65536;     // 2^n Locations

parameter high_impedance = 16'hzzzz;

parameter zero = 16'h0000;


input clk, enable, write_enable;

input [word_size-1 : 0] address;


inout [word_size-1 : 0] data;


reg [word_size-1 : 0] MEMORY [memory_locations-1 : 0];  // 2^n *
n bits memory initialize

reg [word_size-1 : 0] data_out;
```

```
integer i;


// Initialize memory to zero

// Replace data instead of zeros as provided in text file

// Generate MEMORY variable with 2 bytes long 64k locations

initial begin

      for (i = 0; i < memory_locations; i = i + 1)

            MEMORY [i] = zero;

      $readmemh ("code.txt", MEMORY);

      data_out = zero;

      i = 0;

end


// data bus is high impedance for low enable and/or high
write_enable (write mode)

assign data = (enable === 1'b1 && write_enable === 1'b0) ?
data_out : high_impedance;


always @ (posedge clk) begin

      case ({write_enable, enable})

            2'b00 : begin

                  data_out = high_impedance;

            end

            2'b01 : begin                       // Read Mode

                  if (address !== high_impedance)

                        data_out = MEMORY [address];

                  else

                        data_out = high_impedance;

            end
```

```
        2'b10 : begin
            data_out = high_impedance;
        end
        2'b11 : begin                    // Write Mode
            MEMORY [address] = data;
        end
    endcase
end

endmodule
```

3. ALU :-

```
module ArithmeticLogicUnit (clk, enable, reset, in1, in2,
carry_in, mode, func, result, carry_out, sign, zero, parity,
equal, greater_than);


// 16 bit ALU ( 1 bit MSB reserved for signed bit)

// Works on positive edge of clock

// Input signals are active high

// Uses Signed Magnitude format of data to process


/* Inputs */

// clk regulates working of program counter

// enable to control working of alu

// reset to clear contents of alu

// in1 and in2 are two data inputs to alu

// carry_in is input from flag register

// mode to control alu (1 for Arithmetic and 0 for Logic
operations)

// func to select operation to perform


/* Outputs */

// result is output of alu after operation

// carry_out to show carry signal of result

// sign to show if result is negative

// zero to show if result is zero (16'h0000)

// parity to show if there is even parity in result (EXCLUDING
sign bit)

// equal to show if both inputs to alu are equal

// greater_than to show if the 1st input is greater than 2nd
input
```

```
parameter word_size = 16;
parameter high_impedance = 16'hzzzz;

/* Arithmetic functions (Mode = 1) */
// All arithmatic functions are compatible with any combination
// of negative numbers as input
parameter addition = 4'h0;
parameter substraction = 4'h1;
parameter increment = 4'h2;     // increment of 1st input by 1
parameter decrement = 4'h3;     // decrement of ist input by 1

/* Logic functions (Mode = 0) */
parameter xor_op = 4'h0;
parameter and_op = 4'h1;
parameter or_op = 4'h2;
parameter xnor_op = 4'h3;
parameter nand_op = 4'h4;
parameter nor_op = 4'h5;

parameter high = 4'hf;          // all output lines are logic
high
parameter low = 4'he;           // all output lines are logic low
parameter input1 = 4'hd;        // reflect 1st input as output
parameter input2 = 4'hc;        // reflect 2nd input as output
parameter invert_in1 = 4'hb;    // invert 1st input as output
parameter invert_in2 = 4'ha;    // invert 2nd input as output

input clk, enable, reset, carry_in, mode;
input [word_size-1 : 0] in1, in2;

input [3 : 0] func;


output reg [word_size-1 : 0] result;

output reg carry_out, sign, zero, parity, equal, greater_than;


reg [word_size-1 : 0] temp_in1, temp_in2;

reg [word_size : 0] temp_result;


integer count;


// Initializing reg variables

initial begin

	result = 0;

	carry_out = 0;

	sign = 0;

	zero = 0;

	parity = 0;

	equal = 0;

	greater_than = 0;

	temp_in1 = 0;

	temp_in2 = 0;

	temp_result = 0;

	count = 0;

end


always @ (posedge clk) begin

	if(enable === 1'b1) begin   // only perform when enable is
high
```

```
            temp_in1 = in1;

            temp_in2 = in2;

            temp_result = 0;


            equal = ( temp_in1 === temp_in2 )? 1'b1 : 1'b0;

            if (temp_in1[15] > temp_in2[15])

                  greater_than = 1'b0;

            else if (temp_in1 > temp_in2)

                  greater_than = 1'b1;

            else

                  greater_than = 1'b0;


            if (mode === 1'b1) begin // Arithmetic operations

                  case (func)

                        addition: begin

                           // both inputs are positive

                              if ( {temp_in1[15] , temp_in2[15]} ===
2'b00) begin

                                    temp_result = adder (temp_in1,
temp_in2, carry_in);

                                    if ( temp_result [15] === 1)
begin

                                          temp_result [15] = 0;

                                          temp_result [16] = 1;

                                    end

                              end

                              else begin // any or both of input/s
is/are negative

                                 // 1st input is negative

                                    if (temp_in1[15] === 1'b1)
```

```
                                        temp_in1 = comp2 (temp_in1);
// taking 2's complement
                                    // 2nd input is negative
                                    if (temp_in2[15] === 1'b1)

                                        temp_in2 = comp2 (temp_in2);
// taking 2's complement


                                    temp_result = adder (temp_in1,
temp_in2, carry_in);
                                  // result is negative
                                    if (temp_result[15] === 1'b1)
begin
                                        temp_result = comp2
(temp_result);
                                        if(temp_result[15] === 1'b0)
                                            temp_result[15] = 1;
                                        else
                                            temp_result[15] = 0;
                                    end
                                end


                                carry_out = temp_result[16];
                            end


                            substraction: begin
                              // both inputs are positive
                                if ( {temp_in1[15] , temp_in2[15]} ===
2'b00)
                                    temp_in2 = comp2 (temp_in2);
                                // 1st input is negative
```

```
                           else if (temp_in1[15] === 1'b1) begin
                                 temp_in1 = comp2 (temp_in1); //
taking 2's complement
                                 temp_in2 = comp2 (temp_in2);
                           end
                         // 2nd input is negative
                           else
                                 temp_in2[15] = 0; // considering
2nd input as positive

                           temp_result = adder (temp_in1,
temp_in2, carry_in);
                         // result is negative
                           if (temp_result[15] === 1'b1) begin
                                 temp_result = comp2
(temp_result);
                                 if(temp_result[15] === 1'b0)
                                       temp_result[15] = 1;
                                 else
                                       temp_result[15] = 0;
                           end
                      end

                      increment: begin
                           temp_result = adder (temp_in1,
16'h0001, 0);
                      end

                      decrement: begin
                           temp_result = adder (temp_in1, comp2
(16'h0001), 0);
```

```
                end
                default: begin
                     temp_result = 17'h0000;
                end
           endcase
      end

      else  begin      // mode=0 for logical operation
           case (func)
                xor_op: begin
                     temp_result = temp_in1 ^ temp_in2;
                end

                and_op: begin
                     temp_result = temp_in1 & temp_in2;
                end

                or_op: begin
                     temp_result = temp_in1 | temp_in2;
                end

                xnor_op: begin
                     temp_result = (temp_in1 ^ temp_in2);
                     temp_result = invert (temp_result);
                end

                nand_op: begin
                     temp_result = (temp_in1 & temp_in2);
```

```
                    temp_result = invert (temp_result);
                end

                nor_op: begin
                    temp_result = (temp_in1 | temp_in2);
                    temp_result = invert (temp_result);
                end

                high: begin
                    temp_result[15:0] = 16'hffff;
                end

                low: begin
                    temp_result[15:0] = 16'h0000;
                end

        input1: begin
                    temp_result = temp_in1;
                end

                input2: begin
                    temp_result = temp_in2;
                end

                invert_in1: begin
                    temp_result = invert (temp_in1);
                end
```

```
                    invert_in2: begin
                         temp_result = invert (temp_in2);
                    end

                    default: begin
                         temp_result = 17'h0000;
                    end
               endcase
          end

          if (reset === 1'b1) begin
               temp_result = 0;
          end

          zero = ( temp_result [15 : 0] === 16'h0000 )? 1'b1 :
1'b0;
          parity = ~^temp_result [15 : 0];
          sign = temp_result [15];
          result = temp_result [15 : 0];
     end
end

function automatic [word_size-1 : 0] comp2;
     input [word_size-1 : 0] data_in;
     reg [word_size-1 : 0] temp;
     begin
          temp = invert (data_in);
          comp2 = adder (temp, 16'h0001, 0);
     end
```

```
endfunction


function automatic [word_size-1 : 0] invert;
      input [word_size-1 : 0] in;
      begin
      invert = ~in;
      end
endfunction


function automatic [word_size : 0] adder;
      input [word_size-1 : 0] in1, in2;
      input cin;
      reg [word_size-1 : 0] p, g, c;
      integer i;
      begin
            p = in1 ^ in2;
            g = in1 & in2;
            c[0] = g[0] | (cin&p[0]);
            adder[0] = p[0] ^ cin;
            for (i=1; i<16; i=i+1) begin
                  c[i] = g[i] | (c[i-1] & p[i]);
                  adder[i] = p[i] ^ c[i-1];
            end
      adder[16] = c[15];
      End
endfunction


endmodule
```

4. **Decoder :-**

```
module Decoder (clk, instruction, pre_data, data, address, 
src_addr, dst_addr, reg_enable, reg_mode,

                memory_enable, memory_write_enable, alu_enable, 
alu_reset, alu_mode, alu_function,

                            alu2in_enable, gpr_acc_enable, 
ir_reset, pc_reset);


// Decoder (Control Unit) is stage after ir register

// It decodes instruction_set and generates control signals

// Instructions finishes execution after 2 cycles

// It controls primary working of individual modules of processor

// Works on positive edge of clock


/* Inputs */

// clk regulates working of control unit

// instruction provides instruction_set to decode

// pre-data provides data/address before hand from ir


/* Outputs */

// data is conneted to data_bus

// address is connected to address_bus

// src_addr provides address of register for sending data

// dst_addr provides address of register for receiving data

// reg_enable controls working of gpr

// reg_mode used to toggle working modes of gpr

// memory_enable to control working of memory

// memory_write_enable to decide reading or writing operation

// alu_enable to control working of alu
// alu_reset to clear contents of alu

// alu_mode to control alu (1 for Arithmetic and 0 for Logic
operations)

// alu_function to select operation to perform by alu

// alu2in_enable controls 2nd input to alu (data bus or R1) from
gpr

// gpr_acc_enable controls updating R0 register (accumulator)

// ir_reset to clear contents of instruction register


parameter word_size = 16;

parameter high_impedance = 16'hzzzz;


// Load Instructions

parameter LDM = 8'h01;     // load data from given memory address
to register

parameter LDR = 8'h02;     // copy data from given register to
another register

parameter LDV = 8'h03;     // load given value/data to register


// Store Instructions

parameter STR = 8'h04;     // store data from given register to
memory address


// Arithmetic Instructions

parameter ADR = 8'h10;

parameter ADD = 8'h11;

parameter SBR = 8'h12;

parameter SUB = 8'h13;

parameter INC = 8'h14;

parameter DEC = 8'h15;
```

```
// Logic Instructions

parameter XOR_op = 8'h1f;

parameter AND_op = 8'h1e;

parameter OR_op = 8'h1d;

parameter XNOR_op = 8'h1c;

parameter NAND_op = 8'h1b;

parameter NOR_op = 8'h1a;

parameter INV_op = 8'h19;


// Other Instructions

parameter NOP = 8'h00;

parameter REG = 8'hff; //


input clk;

input [word_size-1 : 0] instruction, pre_data;


output reg [word_size-1 : 0] data, address;

output reg [2 : 0] src_addr, dst_addr, reg_mode;

output reg reg_enable, memory_enable, memory_write_enable,
alu_enable, alu_reset, alu_mode;

output reg [3 : 0] alu_function;

output reg alu2in_enable, gpr_acc_enable, ir_reset, pc_reset;


reg mode;


// initalizing reg variables

initial begin

     data = high_impedance;
```

```
        address = high_impedance;

        src_addr = 3'bzzz;

        dst_addr = 3'bzzz;

        reg_mode = 3'bzzz;

        reg_enable = 1'bz;

        memory_enable = 1'bz;

        memory_write_enable = 1'bz;

        alu_enable = 1'bz;

        alu_reset = 1'bz;

        alu_mode = 1'bz;

        alu_function = 4'hz;

        alu2in_enable = 1'bz;

        gpr_acc_enable = 1'bz;

        ir_reset = 1'bz;

        pc_reset = 1'bz;

        mode = 1'b0;

end


always @ (posedge clk) begin

        if (mode === 1'b0) begin

                data = high_impedance;

                address = high_impedance;

                src_addr = 3'b000;

                dst_addr = 3'b000;

                reg_mode = 3'b000;

                reg_enable = 1'b0;

                memory_enable = 1'b0;

                memory_write_enable = 1'b0;
```

```
alu_enable = 1'b0;
alu_reset = 1'b0;
alu_mode = 1'b0;
alu_function = 4'h0;
alu2in_enable = 1'b0;
gpr_acc_enable = 1'b0;
ir_reset = 1'b0;
pc_reset = 1'b0;

case (instruction [7 : 0])
   REG: begin
           src_addr = instruction [10 : 8];
           reg_enable = 1;
           reg_mode = 3'b001; // read mode
           mode = 1'b1;
      end
      NOP: begin
           mode = 1'b0;
      end

      LDM: begin
           address = pre_data;
           memory_enable = 1'b1;
           dst_addr = instruction [10 : 8];
           reg_enable = 1'b1;
           reg_mode = 3'b010; // write mode
           mode = 1'b1;
      end
```

```
LDR: begin
     src_addr = instruction [10 : 8];
     dst_addr = pre_data[2 : 0];
     reg_enable = 1'b1;
     reg_mode = 3'b100; // reg transfer
     mode = 1'b1;
end

LDV: begin
     dst_addr = instruction [10 : 8];
     data = pre_data;
     reg_enable = 1'b1;
     reg_mode = 3'b010; // write
     mode = 1'b1;
end

STR: begin
     src_addr = instruction [10 : 8];
     reg_enable = 1'b1;
     reg_mode = 3'b001; // read
     memory_write_enable = 1'b1;
     address = pre_data;
     memory_enable = 1'b1;
     mode = 1'b1;
end

ADR: begin
```

```
            alu_enable = 1'b1;

            alu2in_enable = 1'b1;

            alu_mode = 1'b1;

            alu_function = 4'h0;

            gpr_acc_enable = 1'b1;

            mode = 1'b1;

      end


      ADD: begin

            data = pre_data;

            alu_enable = 1'b1;

            alu_mode = 1'b1;

            alu_function = 4'h0;

            gpr_acc_enable = 1'b1;

            mode = 1'b1;

      end


      SBR: begin

            alu_enable = 1'b1;

            alu2in_enable = 1'b1;

            alu_mode = 1'b1;

            alu_function = 4'h1;

            gpr_acc_enable = 1'b1;

            mode = 1'b1;

      end


      SUB: begin

            data = pre_data;;
```

```
          alu_enable = 1'b1;

          alu_mode = 1'b1;

          alu_function = 4'h1;

          gpr_acc_enable = 1'b1;

          mode = 1'b1;

     end


     INC: begin

          alu_enable = 1'b1;

          alu_mode = 1'b1;

          alu_function = 4'h2;

          gpr_acc_enable = 1'b1;

          mode = 1'b1;

     end


     DEC: begin

          alu_enable = 1'b1;

          alu_mode = 1'b1;

          alu_function = 4'h3;

          gpr_acc_enable = 1'b1;

          mode = 1'b1;

     end


     XOR_op: begin

        data = pre_data;;

          alu_enable = 1'b1;

          alu_function = 4'h0;

          gpr_acc_enable = 1'b1;
          mode = 1'b1;
     end
     AND_op: begin
          data = pre_data;;
          alu_enable = 1'b1;
          alu_function = 4'h1;
          gpr_acc_enable = 1'b1;
          mode = 1'b1;
     end
     OR_op: begin
          data = pre_data;;
          alu_enable = 1'b1;
          alu_function = 4'h2;
          gpr_acc_enable = 1'b1;
          mode = 1'b1;
     end
     XNOR_op: begin
          data = pre_data;;
          alu_enable = 1'b1;
          alu_function = 4'h3;
          gpr_acc_enable = 1'b1;
          mode = 1'b1;
     end
     NAND_op: begin
          data = pre_data;;
          alu_enable = 1'b1;
          alu_function = 4'h4;
          gpr_acc_enable = 1'b1;
```

```
                    mode = 1'b1;

               end

               NOR_op: begin

                    data = pre_data;;

                    alu_enable = 1'b1;

                    alu_function = 4'h5;

                    gpr_acc_enable = 1'b1;

                    mode = 1'b1;

               end

               INV_op: begin

                    data = pre_data;;

                    alu_enable = 1'b1;

                    alu_function = 4'hd;

                    gpr_acc_enable = 1'b1;

                    mode = 1'b1;

               end

          endcase

     end


     else // mode = 1

          mode = 1'b0;


end


endmodule
```

5. Instruction Register :-

```
module InstructionRegister (clk, write_enable, reset, data_in,
data_out, pre_data_out);


// 16 - bit Instruction Register

// 4 stage prefetch unit

// Works on rising edge of clock

// Every input signal is active high


/* Inputs */

// clk regulates working of memory

// write_enable to enable writing to register

// reset to clear contents of register

// data_in is input data (machine code from program memory)


/* Outputs */

// data_out is output data (instruction set) from instruction
register to decoder module


parameter word_size = 16;

parameter high_impedance = 16'hzzzz;

parameter zero = 16'h0000;


input clk, write_enable, reset;

input [word_size-1 : 0] data_in;


output [word_size-1 : 0] data_out, pre_data_out;
```

```
wire [word_size-1 : 0] data1, data2;



Register R1(.clk (clk),
            .load (write_enable),
                      .reset (reset),
                      .in (data_in),
                      .out (data1)
                      );

Register R2(.clk (clk),
            .load (write_enable),
                      .reset (reset),
                      .in (data1),
                      .out (data2)
                      );

Register R3(.clk (clk),
            .load (write_enable),
                      .reset (reset),
                      .in (data2),
                      .out (pre_data_out)
                      );

Register R4(.clk (clk),
            .load (write_enable),
                      .reset (reset),
                      .in (pre_data_out),
                      .out (data_out)
                      );

Endmodule

module Register (clk, load, reset, in, out);

// 16 - bit Regiuster
// Works on rising edge of clock
// Every input signal is active high

/* Inputs */
// clk regulates working of register
// load to control writing to register
// reset to clear contents of register
// in provides data for storing in register

/* Output */
// out provides stored data in register

parameter word_size = 16;
parameter high_impedance = 16'hzzzz;
parameter zero = 16'h0000;

input clk, load, reset;
input [word_size-1 : 0] in;

output reg [word_size-1 : 0] out;
```

```
initial begin
      out = 16'h0000;
end

always @ (posedge clk) begin
      if (reset === 1'b1)
            out <= zero;

      else if (load === 1'b1)
            out <= in;

      else
            out <= out;
end

endmodule
```

6. Flag Register :-

```
module FlagRegister (clk, reset, in, out);


// 6 - bit Flag Register

// Works on positive edge of clock

// Input signals are active high


/* Inputs */

// clk regulates working of register

// reset overrides all values stored in register to zero

// in provides input data to register


/* Output */

// out provides flag register contents


/* Flag Register Contents */

// R[0] = carry flag

// R[1] = zero flag

// R[2] = sign flag

// R[3] = greater_than flag

// R[4] = equal flag

// R[5] = parity (even) flag


input clk, reset;

input [ 5 : 0 ] in;

output reg [ 5 : 0 ] out;


// initialize reg variables

initial begin
```

```
      out = 0;
end

always @ (posedge clk) begin
      if (reset === 1'b1) begin
            out <= 0;
      end
      out [ 0 ] <= in [ 0 ]; // carry
      out [ 1 ] <= in [ 1 ]; // zero
   out [ 2 ] <= in [ 2 ]; // sign
   out [ 3 ] <= in [ 3 ]; // greater_than
   out [ 4 ] <= in [ 4 ]; // equal
   out [ 5 ] <= in [ 5 ]; // parity
end

endmodule
```

7. General Purpose Register :-

```
module GeneralPurposeRegister (clk, enable, alu2in_enable,
acc_enable, mode,

                              src_addr, dst_addr, acc_input,
alu_1st_in, alu_2nd_in, data);


// 8 16-bit Genreal Purpose Registers (R0 to R7)

// Works on positive edge of clock

// Input signals are active high

// R0 is accumulator where all alu results are

// placed after alu operation

// R0 and R1 acts as alu 2 inputs


/* Inputs */

// clk regulates working of gpr

// enable controls working of gpr

// alu2in_enable controls 2nd input to alu (data bus or R1)

// acc_enable controls updating R0 register (accumulator)

// mode used to toggle working modes of register

// src_addr provides address of register for sending data

// dst_addr provides address of register for receiving data

// acc_input provides alu result data to store


/* Outputs */

// alu_1st_in provides 1st input to alu from R0 (accumulator)

// alu_2nd_in provides 2nd input to alu from R1


/* Inouts */
```

```
// data bus for input/output of data from registers


parameter word_size = 16;

parameter high_impedance = 16'hzzzz;

parameter zero = 16'h0000;


input clk, enable, alu2in_enable, acc_enable;

input [2 : 0] mode, src_addr, dst_addr;

input [word_size-1 : 0] acc_input;


output [word_size-1 : 0] alu_1st_in, alu_2nd_in;


inout [word_size-1 : 0] data;


reg [word_size - 1 : 0] register [7 : 0];

reg [word_size - 1 : 0] data_out;


// Initializing registers and reg variables

initial begin

      register[0] = zero;

      register[1] = zero;

      register[2] = zero;

      register[3] = zero;

      register[4] = zero;

      register[5] = zero;

      register[6] = zero;

      register[7] = zero;

      data_out = high_impedance;
```

```
end


assign alu_1st_in = register[0];

// if alu2in_enable = 1 then alu 2nd input is from R1

assign alu_2nd_in = (alu2in_enable === 1'b1) ? register[1] :
high_impedance;


assign data = (enable === 1'b1 && mode === 3'b001) ? data_out :
high_impedance;


always @ (posedge clk) begin

      if (enable) begin

      case (mode)

            3'b001 : begin

                  data_out = register[ src_addr ]; // read mode

            end


            3'b010 : begin

                  register[ dst_addr ] = data; // write mode

            end


            3'b011 : begin

                  register[ dst_addr ] = 16'h0000; // reset mode

            end


            3'b100 : begin

                  register[ dst_addr ] = register[ src_addr ]; //
register transfer mode

            end

            default : begin
```

```
                data_out = high_impedance;
           end
      endcase
      end
      // allow updating accumulator if acc_enable = 1
      if (acc_enable === 1'b1)
           register[ 0 ] = acc_input;
end

endmodule
```

8. Top module :-

```
module TopModule (clk);


// Top Module assembles all parts of processor together


parameter word_size = 16;

parameter high_impedance = 16'hzzzz;


/* Inputs */

// Clock regualtes working of all modules

input clk;


/* Main Buses */

// Buses responsible for primary data/address operations

wire [word_size-1 : 0] address_bus, data_bus, instruction_address,
instruction_set;


/* Program Counter Wires */

wire [word_size-1 : 0] PC_jump_address;


reg PC_count_enable, PC_reset, PC_load_address, PC_jump_enable;


/* Memory Wires */

wire Memory_enable, Memory_write_enable;

wire [word_size-1 : 0] machine_code;


/* Instruction Register Wires */

reg IR_write_enable, IR_reset;

wire [word_size-1 : 0] pre_data;
```

```
/* ALU Wires */

wire carry, sign, zero, parity, equal, greater_than, ALU_reset,
ALU_enable, ALU_mode;

wire [3 : 0] ALU_function;

wire [word_size-1 : 0] alu_1st_in, alu_R1_in;

wor [word_size-1 : 0] alu_2nd_in;


assign alu_2nd_in = alu_R1_in;

assign alu_2nd_in = data_bus;


/* GPR Wires */

wire GPR_enable, GPR_alu2in_enable, GPR_acc_enable;

wire [2 : 0] GPR_mode, GPR_src_addr, GPR_dst_addr;

wire [word_size-1 : 0] GPR_acc_input;


/* Flag Register Wires */

wire FR_reset, parity_flag, equal_flag, greater_than_flag,
sign_flag, zero_flag, carry_flag;


initial begin

     PC_reset = 1'b0;

     PC_jump_enable = 1'b0;

     PC_count_enable = 1'b1;

     PC_load_address = 1'b1;

     IR_write_enable = 1'b1;

end


/* Instantiation of modules */
```

```
ProgramCounter PC (.clk (clk),
                    .count_enable (PC_count_enable),
                                   .reset (PC_reset),
                                   .load_address (PC_load_address),
                                   .jump_enable (PC_jump_enable),
                                   .jump_address (PC_jump_address),
                                   .address (instruction_address)
                                   );

Memory ProgramMemory (.clk (clk),
                       .enable (1'b1),
                                           .write_enable (1'b0),
                                           .address
(instruction_address),
                                           .data (machine_code)
                                           );

Memory DataMemory (.clk (clk),
                    .enable (Memory_enable),
                                 .write_enable
(Memory_write_enable),
                                   .address (address_bus),
                                   .data (data_bus)
                                   );

InstructionRegister IR (.clk (clk),
                         .write_enable (IR_write_enable),
                                          .reset (IR_reset),
                                          .data_in
(machine_code),
                                          .data_out
(instruction_set),
                                          .pre_data_out
(pre_data)
                                          );


ArithmeticLogicUnit ALU(.clk (clk),
                         .enable (ALU_enable),
                                          .reset (ALU_reset),
                                          .in1 (alu_1st_in),
                                          .in2 (alu_2nd_in),
                                          .carry_in (carry_flag),
                                          .mode (ALU_mode),
                                          .func (ALU_function),
                         .result (GPR_acc_input),
                                          .carry_out (carry),
                                          .sign (sign),
                                          .zero (zero),
                                          .parity (parity),
                                          .equal (equal),
                                          .greater_than
(greater_than)
                                          );


GeneralPurposeRegister GPR (.clk (clk),
                               .enable (GPR_enable),
                                                           .alu2in_enable
(GPR_alu2in_enable),
                                                  .acc_enable
(GPR_acc_enable),
                                                  .mode
(GPR_mode),
                                                  .src_addr
(GPR_src_addr),
                                                  .dst_addr
(GPR_dst_addr),
                                                  .acc_input
(GPR_acc_input),
                                                  .alu_1st_in
(alu_1st_in),
                                                  .alu_2nd_in
(alu_R1_in),
                                                  .data (data_bus)
                                                  );


FlagRegister FR (.clk (clk),
                 .reset (FR_reset),
                             .in ({parity, equal, greater_than,
sign, zero, carry}),
                             .out ({parity_flag, equal_flag,
greater_than_flag, sign_flag, zero_flag, carry_flag})
                             );


Decoder dec (.clk (clk),
             .instruction (instruction_set),
                      .pre_data (pre_data),
                      .data (data_bus),
                      .address (address_bus),
                      .src_addr (GPR_src_addr),
                      .dst_addr (GPR_dst_addr),
                      .reg_enable (GPR_enable),
                      .reg_mode (GPR_mode),
                      .memory_enable (Memory_enable),
                      .memory_write_enable (Memory_write_enable),
                      .alu_enable (ALU_enable),
                      .alu_reset (ALU_reset),
                      .alu_mode (ALU_mode),
                      .alu_function (ALU_function),
                      .alu2in_enable (GPR_alu2in_enable),
                      .gpr_acc_enable (GPR_acc_enable),
                      .ir_reset (IR_reset),
                      .pc_reset(PC_reset)
                      );

endmodule
```