VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnanasangama, Macche, Santibastwada Road
Belagavi-590018, Karnataka

A
Project Report
on

Implementation of a Custom 16-Bit Core on FPGA

Submitted in partial fulfillment of the requirement for the degree of

Bachelor of Engineering
in
Electronics & Communication Engineering
by

1DS16EC013	**Aldrich Shawn Lewis**
1D16SEC090	**Paarthvi Sharma**
1DS16EC096	**Prajwal Shenoy KP**
1DS16EC115	**Sanath S. Naik**

Under the guidance
of

Prof. Druva Kumar S.
Internal Departmental Project Guide
Designation, ECE Dept., DSCE, Bengaluru

Department of Electronics & Communication Engineering
(An Autonomous College affiliated to VTU Belgaum, accredited by NBA & NAAC)
Shavige Malleshwara Hills, Kumaraswamy Layout,
Bengaluru-560078, Karnataka, India

2019-20

Dedicated to our

Beloved

Teachers

Certificate

Certified that the project work entitled "Implementation of a Custom 16-Bit Core on FPGA" carried out by Aldrich Shawn Lewis (1DS16EC013), Paarthvi Sharma (1DS16EC090), Prajwal Shenoy KP (1DS16EC096), Sanath S Naik (1DS16EC115) are bonafide students of Dayananda Sagar College of Engineering, Bangalore, Karnataka, India in partial fulfillment for the award of Bachelor of Engineering in Electronics & Communication Engineering of the Visvesvaraya Technological University, Belagavi, Karnataka during the academic year 2019-20. It is certified that all corrections / suggestions indicated for project work have been incorporated in the report deposited to the ECE department, the college central library & to the university. This project report **(EC83)** has been approved as it satisfies the academic requirement in respect of project work prescribed for the said degree.

Dept. Project Coordinator Convener	Project Guide
Dr. K.N. Pushpalatha	Druva Kumar S
Head of the Department	Dr. C.P.S. Prakash
Dr. T.C.Manjunath	Principal

External Project Viva-Voce

Name of the project examiners :

1 : Signature : _____

2 : Signature : _____

Declaration

Certified that the project work entitled, "Implementation of a Custom 16-Bit Core on FPGA" is a bonafide work that was carried out by ourselves in partial fulfillment for the award of degree of Bachelor of Engineering in Electronics & Communication Engg. of the Visvesvaraya Technological University, Belagavi, Karnataka during the academic year 2019-20. We, the students of the project group/batch no. R-32 hereby declare that the entire project work has been done on our own & we have not copied or duplicated any other's work. The results embedded in this project report has not been submitted elsewhere for the award of any type of degree.

Mr. Aldrich Shawn Lewis
USN : 1DS16EC013

Sign : _____

Ms. Paarthvi Sharma
USN : 1DS16EC090

Sign : _____

Mr. Prajwal Shenoy KP
USN : 1DS16EC096

Sign : _____

Mr. Sanath S. Naik
USN : 1DS16EC115

Sign : _____

Date : / /

Place : Bengaluru -78

Acknowledgement

The satisfaction that accompanies the successful presentation of our final year project would be incomplete without mentioning the people who made it possible, whose relentless support and encouragement crowns all the efforts with success.

We are thankful to – Dr. Hemachandra Sagar – Chairman, Dr. Premachandra Sagar–Vice Chairman, Galiswamy –Secretary, Dr. C P S Prakash, Principal, DSCE, Bangalore for providing me an opportunity to work on this project.

We are thankful to HOD – Dr. T.C Manjunath, Dept. of ECE for his support and encouragement at all times.

We would specially like to thank my Project guide Prof. Druva Kumar S. ECE, DSCE for his continuous support, encouragement and help throughout this venture.

We owe our deep gratitude to sir for taking keen interest in our project work and guided us all along till the completion of our project work by providing all the necessary information.

We take this opportunity to express our profound gratitude and deep regards to the Project Convener – Dr. K.N. Pushpalatha, ECE for her cordial support and helping us throughout the project.

We would also like to thank the Project Coordinators – Dr. P.Vimala, Dr. A. Rajagopal, Prof. Sapna P J,Prof. Deepa N P for the cordial support, encouragement and exemplary guidance throughout the project.

We wish to express our profound gratitude to all staff members of Electronics and Communication Department, DSCE, family and friends for their guidance,

necessary support, co-operation, encouragement, fullest effort of them to the success of this opportunity, all are grateful and unforgettable.

Finally, we thank the almighty Lord for giving us life.

Abstract

This project presents the design and implementation of a 16-Bit hack CPU which is a modular processor. The paper is intended to showcase the process involved in building a complex circuit capable of performing real world computations, from the most basic component used for digital data representation that is the CMOS. The design methodology used is a bottom up approach, this starts with the construction of basic gates and moving up to major components like the program counter, ALU, etc. and ends with the complete construction of the CPU using the previously built components in a modular manner. The aim of this paper is to give the reader a complete understanding of the functioning of a simple computer in a digital electronics abstract. This paper will also give an idea about the data flow in a CPU triggered by a CPU. An idea regarding the way a low level programming language controls this data flow can also be understood. This CPU design is easily implementable on an FPGA and is hence a great tool to teach students about the basics of Computer Architecture and Digital System Design. The CPU reads instructions from the ROM and performs operations using the A register, D register, or the RAM memory units based on the instruction type. There are mainly 2 types of instructions, A instructions and C instructions. The A instructions have the sole purpose of storing values into the A register while the C instruction can perform multiple operations.

Keywords — *Arithmetic Logical Unit(ALU), Program Counter(PC), Central Processing Unit(CPU), Processor, Decoder.*

Table of Contents

List of Figures

List of Tables

Nomenclature and Acronyms

Abbreviations :

IEEE	Institute of Electrical & Electronics Engineers
DSCE	Dayananda Sagar College of Engineering
ECE	Electronics & Communication Engineering
CPU	Central Processing Unit
ALU	Arithmetic Logic Unit
PC	Program Counter

CHAPTER 1

INTRODUCTION

1.1 Overview

Processors are small silicon chips which execute instructions to complete tasks. These instructions are written in the storage device connected to the processor. These instructions are executed one at a time in a logical manner by the processor. These instructions can be of varied sizes. The unit of an instruction is a bit, a digital value which can represent 2 states 1 (logic HIGH and 0 logic LOW). Some of the most common instruction sizes are 8bit, 16bit, 32bit and 64bits. An instruction of size 16bits can represent a total of 2^{16} unique instructions. These instructions decide which type of operation the processor should execute. The instructions are designed such that each bit has a specific function and controls a specific part of the processor. The 16bit processor is designed in a modular manner such that any individual component could be swapped to a different component to observe performance changes without the need to redesign the whole processor again. The processor has also been designed in a bottom up approach (starting by creation of the logic gates using MOSFETS) so that any changes made in any level reflect upwards without any redesigning. The method of abstraction has also been implemented here such that an architect does not have to worry about the underlying core components present.

[13]There are different view points from which one can view a computer system depending on how they interact with the computer system. The concept of abstraction is important to explore the details that are necessary from a particular viewpoint. Computer systems can be viewed from different perspective depending on the user. Programmers uses the instruction set architecture (ISA) is as a useful abstraction to understand the processor's internal details. ISA defines the processor at a logic level, it defines the personality of a processor. ISA includes the type of instructions and the meaning of those instructions. [18]Many commercial microcontrollers have been built to suit different requirements featuring many private-and-licensed instruction sets. Licensed instruction sets and microprocessor cores restrict the process of modifying the core for different purposes such as improving performance and adapting it to specific applications.[16]Most current computer architectures are based on a sequential model of program execution in which an architectural program counter sequences through instructions one-by-one, finishing one before starting the next. In contrast, a high performance implementation may be pipelined, permitting several instructions to be in some phase of execution at the same time.[17]Implementation of an efficient microcontroller requires a reliable and fast communication between masters and slaves blocks in the microcontroller. Nowadays, many bus-based communication architecture standards are found. In this work we are using the AMBA and APB protocols such that we can compare with microcontrollers based on ARM-M0 cores.

This project presents the design and implementation of a 16-Bit CPU which is a modular processor. The paper is intended to showcase the process involved in building a complex circuit capable of performing real world computations, from the most basic component used for digital data representation that is the CMOS. The design methodology used is a bottom up approach, this starts with the construction of basic gates and moving up to major components like the program counter, ALU, etc. and ends with the complete construction of the CPU using the previously built components in a modular manner. The aim of this paper is to give the reader a complete understanding of the functioning of a simple computer in a digital electronics abstract. This paper will also give an idea about the data flow in a CPU triggered by a CPU. An idea regarding the way a low level programming language controls this data flow can also be understood. This CPU design is easily implementable on an FPGA and is hence a great tool to teach students about the basics of Computer Architecture and Digital System Design. The CPU reads instructions from the ROM and performs operations using the A register, D register, or the RAM memory units based on the instruction type. There are mainly 2 types of instructions, A instructions and C instructions. The A instructions have the sole purpose of storing values into the A register while the C instruction can perform multiple operations.

[14]At the top level, the processor core can be divided into four logical modules as shown in Figure 1. The fetch, decode and control logic block is responsible for fetching the instruction from the instruction memory, decoding the instruction and generating the control signals. It is also responsible for resolving jump and branch target addresses. It hosts the program counter, the target address selection logic, the instruction memory controller, the instruction decoder and a control unit.

The platform is a 16-bit von Neumann machine, consisting of a CPU, two separate memory modules serving as instruction memory and data memory. The computer executes programs that reside in its instruction memory. The instruction memory is a read-only device, and thus programs are loaded into it using some exogenous means. For example, the instruction memory can be implemented in a ROM chip that is preburned with the required program.

Loading a new program can be done by replacing the entire ROM chip. In order to simulate this operation, hardware simulators of the platform must provide a means for loading the instruction memory from a text file containing a program written in the machine language. (From now on, we will refer to data memory and instruction memory as RAM and ROM, respectively.

The CPU consists of the ALU and three registers called data register (D), address register (A), and program counter (PC). D and A are general-purpose 16-bit registers that can be manipulated by arithmetic and logical instructions like A=D-1, D=D|A, and so on, following the machine language. While the D-register is used solely to store data values, the contents of the A-register can be interpreted in three different ways, depending on the instruction's context: as a data value, as a RAM address, or as a ROM address.

The machine language is based on two 16-bit command types. The address instruction has the format 0vvvvvvvvvvvvvv, each v being 0 or 1. This instruction causes the computer to load the 15-bit constant vvv...v into the A-register. The compute instruction has the format 111accccccdddjjj. The a- and c-bits instruct the ALU which function to compute, the d-bits instruct where to store the ALU output, and the j-bits specify an optional jump condition, all according to the custom machine language specification.

The computer architecture is wired in such a way that the output of the program counter (PC) chip is connected to the address input of the ROM chip. This way, the ROM chip always emits the word ROM[PC], namely, the contents of the instruction memory location whose address is "pointed at" by the PC. This value is called the current instruction. With that in mind, the overall computer operation during each clock cycle is as follows:

Execute: Various bit parts of the current instruction are simultaneously fed to various chips in the computer. If it's an address instruction (most significant bit = 0), the A-register is set to the 15-bit constant embedded in the instruction. If it's a compute instruction (MSB = 1), its underlying a-, c-, d- and j-bits are treated as control bits that cause the ALU and the registers to execute the instruction.

Fetch: Which instruction to fetch next is determined by the jump bits of the current instruction and by the ALU output. Taken together, these values determine whether a jump should materialize. If so, the PC is set to the value of the A-register; otherwise, the PC is incremented by 1. In the next clock cycle, the instruction that the program counter points at emerges from the ROM's output, and the cycle continues.

This particular fetch-execute cycle implies that in the platform, elementary operations involving memory access usually require two instructions: an address instruction to set the A register to a particular address, and a subsequent compute instruction that operates on this address (a read/write operation on theRAM or a jump operation into the ROM).

1.2 Literature survey

1. Title:DESIGN AND APPLICATION OF RISC PROCESSOR

Authors: Mohammad Zaid, Prof. Pervez Mustajab,Department of Electronics Engineering Zakir Hussain College of Engineering & Technology Aligarh Muslim University, Aligarh. U.P. India

Year: May 2018

This paper presents the design of multi-cycle 32-bit Reduced Instruction Set Computer (RISC) processor for better performance and higher speed of operation. The processor is capable of executing more number of instructions with simple design and less critical path delay. The processor executes each and every instruction in more than one

cycle that's why the term multi-cycle. Each instruction is divided into three main states namely the Fetch state, the Decode state and the Execution state.

This paper will act as a reference for the comparison of the performance of our RISC-V core pipelines with the described MIPS core's pipeline.

2.Title: Single Cycle RISC-V Micro Architecture Processor and its FPGA Prototype

Authors:Don Kurian Dennis, Ayushi Priyam, Sukhpreet Singh Virk, Sajal Agrawal, Tanuj Sharma, Arijit Mondal and Kailash Chandra Ray Indian Institute of Technology Patna

Year: March 2018

In this paper, development of a fully synthesizable 32-bit processor based on the open-source RISC-V (RV32I) ISA is presented. A RISC-V development and validation framework with assembling tools and automated test suits is also presented in this paper. The resulting processor is a single core, in-order, non-bus based, RISC-V processor with low hardware complexity. The proposed processor is implemented in Verilog HDL and further prototyped on FPGA "Spartan 3E XC3S500E" board.

The paper helps us with the understanding of the RV32I Instruction set we plan on using for our micro-controller. The FPGA implementation methodology will also help us understand the hardware implementation involving the utilization of the on board RAM and ROM present on the FPGA.

3. Title: The history and use of pipelining computer architecture: MIPS pipelining implementation.

Authors: Iro Pantazi – Mytarelli New York Institute of Technology Old Westbury, New York, 11568

Year: August 2013

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes the advantage of parallelism that exists among the actions needed to execute an instruction.

In this paper, implementation of pipelining in MIPS architecture is done and methods of handling pipelining hazards have been explained. This paper may be referenced to help us identify errors and determine possible solutions to deal with Structural, Data and Control hazards in the pipeline of our project.

4. Title: A 32-bit RISC-V AXI4-lite bus-based Microcontroller with 10-bit SAR ADC.

Authors: Ckristian Duran, Luis Rueda D., Giovanny Castillo, Anderson Agudelo, Camilo Rojas, Luis Chaparro, Harry Hurtado, Juan Romero, Wilmer Ramirez, Hector Gomez, Javier Ardila, Luis Rueda, Hugo Hernandez, Jose Amaya and Elkim Roa Design Group of Integrated Systems CIDIC, Universidad Industrial de Santander, Bucaramanga, Colombia.

Year: April 2016

In this paper a complete implementation and design of a fully-synthesized 32-bit micro-controller in a 130nm CMOS technology is presented. This is the first micro-controller featuring the open source RISC-V instruction set all mounted through AXI4-Lite and APB buses for communication process. The micro-controller contains a 10-bit SAR ADC, a 12-bit DAC, an 8-bit GPIO module, a 4kB-RAM, an SPI AXI slave interface for output verification, and an SPI APB slave interface for checking the correct behavioural of the APB bridge. All peripherals are controlled by a RISC-V and an SPI AXI master interface that is used for programming the device and checking the data flowing through all the slaves. This paper will act as a reference RISC-V processor with a bus capable of communication with peripherals via SPI protocol.

We will be utilizing the above mentioned papers as reference to design our controller. Since our controller is built from scratch and has a 2 stage pipeline, each of its blocks will be unique.

Related Work

The machine language is adopted from the book The Elements of Computing system by Noam Nisan and Shimon Schocken [2] which provides us with the knowledge of designing a processor from scratch. The book uses custom HDL (hardware description language) to design the components and the integration is done using those components. The assembler is also created by referring to the same book [2]. For further understanding of how the different components of the processor works, we have referred to a few computer architectures and embedded system designing books [4], [5], [6], [7]. We learnt how to design the custom instruction set and how to reduce the number of components required to execute these instructions from the past work of some authors. [8], [9], [10], [11]. The components are built using standard design in Verilog. Verilog coding methodology and design flow were referred from the book Verilog HDL-A guide to digital design and synthesis by Samir Palnitkar [1]. Similar kind of work was done by an author [3] but the drawback of that processor is that it's not modular. In another reference paper, a simple CPU is programmed using C language and assembly language which aims to teach the designing basics of a simple and customisable CPU [12]. Similar approach has been considered while designing our processor.

1.3 Objectives / Scope / Aim of the project work

The project aims to design and implement a 16-Bit CPU which is a modular processor. The project intends to showcase the process involved in building a complex circuit capable of performing real world computations, from the most basic component used for digital data representation that is the CMOS. The design methodology to be used is a bottom up approach, this starts with the construction of basic gates and moving up to major components like the program counter, ALU, etc. and ends with the complete construction of the CPU using the previously built components in a modular manner. The 16bit processor is designed in a modular manner such that any individual component could be swapped to a different component to observe performance changes without the need to redesign the whole processor again.

1.4 Methodology

The processor has been designed in a bottom up approach (starting by creation of the logic gates using MOSFETS) so that any changes made in any level reflect upwards without any redesigning. The method of abstraction has also been implemented here such that an architect does not have to worry about the underlying core components present.

The program to be translated is stored is an .asm file. The assembler translates the .asm file into the binary equivalent. These binary instructions are written onto the ROM. Each command is translated individually. Each field of the assembly command is translated into machine code in accordance to the tables given above. To translate the assembly code to machine code the assembler first checks if the instruction is an A instruction or C instruction.

The A instruction will always start with a zero. It is used to set the A-register to a desired 15- bit value. The A instruction is the only way that we can enter data into the computer while the program is running. Also we can jump to certain memory locations to manipulate the data present at that location. Also it can be used to control the jump instruction by loading the jump address to the A register. When we use @constant we load an unsigned value to the A register. This can then be used to perform computation or any other operation that requires the constant.

If it is a C instruction, then the assembler checks the A-bit. If the A bit is 1, the ALU performs operations between the Memory and the D register. If the A bit is 0, the ALU performs operations between the A and D registers. Based on the instruction the binary value for the bits c1-c6 are assigned. The next three bits are the destination bits (d1, d2, d3) which are used to indicate where the data must be stored (RAM, A reg, D reg). The next 3 bits are the jump bits (j1, j2, j3). Jump bits are used only when there is a jump instruction. There are 7 jump instructions (6 compare and jump, 1 simple jump).

			Computation							Destination			Jump		
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

Table1.1: Instruction Format

The Computation Specification: The ALU is designed to compute a fixed set of functions on the D, A, and M registers (where M stands for Memory[A]). The computed function is specified by the a-bit and the six c-bits comprising the instruction's comp field. This 7-bit pattern can potentially code 128 different functions, of which only the 28 listed in Table 1.2.

Suppose we want to have the ALU compute D-1, the current value of the D register minus 1. According to Table 2, this can be done by issuing the instruction 1110 0011 1000 0000 (the 7-bit operation code is in bold). To compute the value of D | M, we issue the instruction 1111 0101 0100 0000. To compute the constant-1, we issue the instruction 1110 1110 1000 0000, and so on.

Comp	c1	c2	c3	c4	c5	c6	Comp
when a = 0							when a = 1
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

Table 1.2: Computation Table

The Destination Specification The value computed by the comp part of the C-instruction can be stored in several destinations, as specified by the instruction's 3-bit dest part Table 1.3. The first and second d-bits code whether to store the computed value in the A register

and in the D register, respectively. The third d-bit codes whether to store the computed value in M (i.e., in Memory[A]). One, more than one, or none of these bits may be asserted.

Destination	d1	d2	d3
Null	0	0	0
M	0	0	1
D	0	1	0
MD	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
AND	1	1	1

Table 1.3: Destination Table

The Jump Specification: The jump field of the C-instruction tells the computer what to do next. There are two possibilities: The computer should either fetch and execute the next instruction in the program, which is the default, or it should fetch and execute an instruction located elsewhere in the program. In the latter case, we assume that the A register has been previously set to the address to which we have to jump. Whether or not a jump should actually materialize depends on the three j-bits of the jump field and on the ALU output value (computed according to the comp field). The first j-bit specifies whether to jump in case this value is negative, the second j-bit in case the value is zero, and the third j-bit in case it is positive. This gives eight possible jump conditions, shown in Table 1.4.

Jump	j1	j2	j3
null	0	0	0
JGT	0	0	1
JEQ	0	1	0
JGE	0	1	1
JLT	1	0	0
JNE	1	0	1
JLE	1	1	0
JMP	1	1	1

Table1.4: Jump Table

1.5 Organization of the project report

The project work undertaken by us is organized in the following sequence as follows:

- Chapter 1: A brief introduction to the work is presented. The chapter is divided into various sub parts in which the aim and scope of the project is discussed, the literature survey is briefly explained and the methodology is explained in detail.

- Chapter 2: After we are done with the introduction, we move to chapter 2 where brief explanation is given about the principle of the project, different components designed are explained in detail and the process of integration of the components is discussed.

- Chapter 3: Details of the software and the hardware tools used to implement our work are described in this section of the report.

- Chapter 4: The chapter deals with the overall algorithm and flowchart containing the steps of creation of the project are described.

- Chapter 5: This chapter deals with the various results that we have obtained as the consequence of our algorithm and tools used.

- Chapter 6: This chapter talks about the advantages, disadvantages and limitations of the project.

- Chapter 7: Finally, the report ends with the conclusion and future scope of the project.

Appendices, Data Sheets, Program codes, Catalogue Sheets, Papers presented (a copy of the presented paper with the certificate) are presented at the end of the project report one after the other in succession.

CHAPTER 2

BLOCK DIAGRAM AND WORKING PRINCIPLE

Brief explanation about block diagram and working principle of project:

The entire computer architecture consists of three major blocks, the RAM, ROM and CPU .as we can see in the figure, the CPU is enclosed in the dotted box and consists of some registers, a control unit that is our decoder, multiplexers and the ALU.The CPU reads instructions from the ROM and performs operations using the A register, D register, or the RAM memory units based on the instruction type.

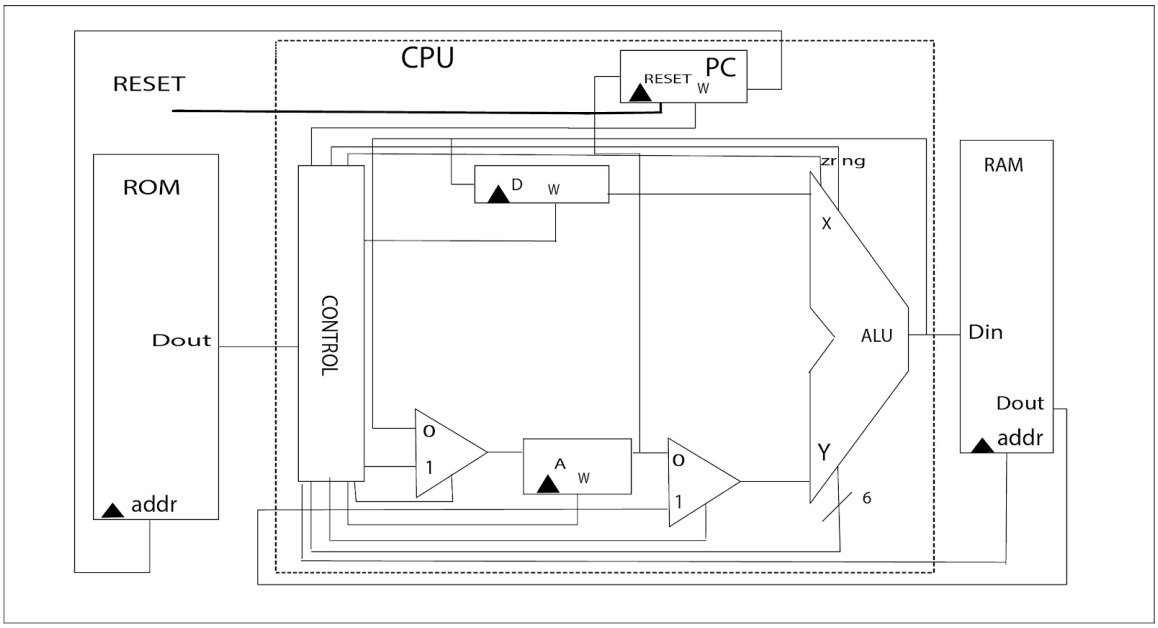

Fig.2.1: Processor Diagram

Following are the different components designed for the project:

- NOT gate

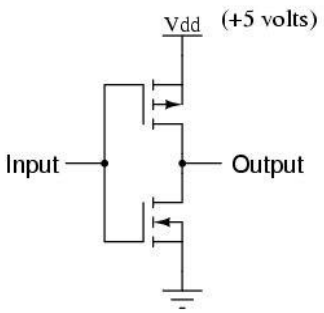

Fig. 2.2: NOT Gate

The above block diagram is of a NOT gate using MOSFETS. It consists of two MOSFETS, one PMOS and one NMOS. The PMOS switches ON and the NMOS switches OFF when the input is HIGH and vice versa when the input is LOW, therefore giving an output LOW and HIGH respectively.

- OR gate

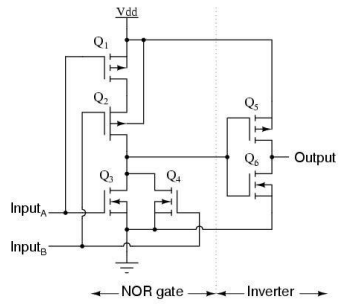

Fig. 2.3: OR Gate

The above block diagram is of a OR gate, consisting of a NOR gate followed by an NOT gate. The OR gates consist of six gates, three PMOS and three NMOS MOSFETs. For the NOR gate the NMOS MOSFETs are connected in parallel and the PMOS MOSFETs are connected in series to perform the OR operations. When these two configurations are connected in series, due to the CMOS effect, the configuration acts as a NOR gate. By connecting a NOT gate, the configuration can be made an OR gate.

- AND gate

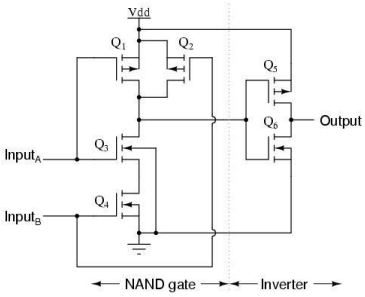

Fig. 2.4: AND Gate

The above block diagram is of a AND gate, consisting of a NAND gate followed by an NOT gate. The AND gates consist of six gates, three PMOS and three NMOS mosfets. For the NAND gate the NMOS mosfets are connected in series and the PMOS mosfets are connected in parallel to perform the AND operations. When these two configurations are connected in series, due to the CMOS effect, the configuration acts like a NAND gate. By connecting a NOT gate, the configuration can be made an AND gate.

- XOR Gate

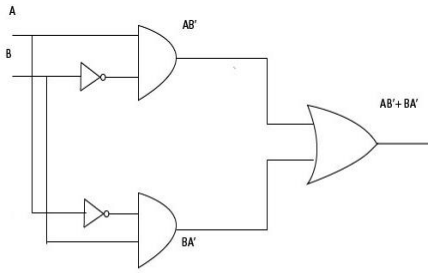

Fig. 2.5: XOR gate

The XOR gate consists of multiple basic gates like AND, OR and NOT with connections as shown in the block diagram. There are basically two Boolean expressions AB` and BA` which go as inputs to the OR gate giving the output of AB`+BA`. This is the Boolean expression of an XOR gate.

- Half Adder Circuit

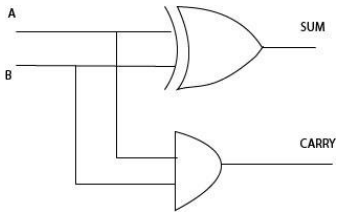

Fig. 2.6: Half Adder Circuit

The half adder circuit consists of an XOR gate and an AND gate. The two single bit inputs A and B are given as inputs to both the XOR gate and AND gate. The output of the XOR gate gives the SUM of the two inputs A and B. The output of the AND gate gives the CARRY output of the two inputs A and B.

- Full Adder Circuit

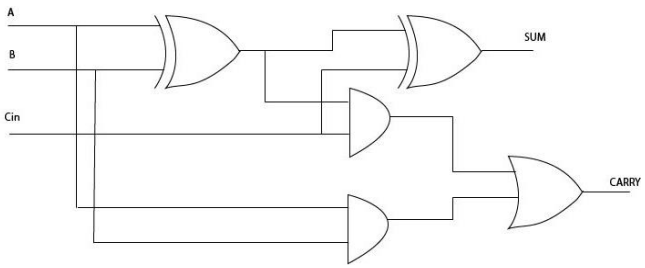

Fig. 2.7: Full Adder Circuit

The full adder circuits consist of two half adder circuits and one OR gate. There are three inputs to the full adder circuit. The full adder circuit has 2 outputs, SUM and CARRY. The two main inputs A and B are given as the inputs to the first half adder circuit. The SUM output of which is given as the input to the next half adder. The carry is given to the OR gate as shown in the block diagram. The CARRY IN input is given as the input to the second half adder circuit. The CARRY of the second half adder circuit is given as the input to the OR gate. The SUM of the second half adder is the SUM of the full adder circuit. The output of the OR gate is the CARRY output of the full adder circuit.

- 16-Bit AND Gate

Fig. 2.8: 16 Bit AND gate

The 16bit AND gate is a collection of 15 AND gates connected in a cascaded manner. The least significant bits are given as input to the first AND gate, the output of which is given as the input to the next AND gate. This is how the AND gates are cascaded. The output of the 15th AND gate is the output of the 16 Bit AND gate configuration. This AND gate performs the AND operation on all the 16 bits. The 16 bit AND gate gives an output of 1 (HIGH) only if all the 16 inputs are 1 (HIGH).

- 16-Bit XOR Gate

Fig. 2.9: 16 Bit XOR gate

The 16bit XOR gate is a collection of 15 XOR gates connected in a cascaded manner. The least significant bits are given as input to the first XOR gate, the output of which is given as the input to the next XOR gate. This is how the XOR gates are cascaded. The output of the 15th XOR gate is the output of the 16 Bit XOR gate configuration. This XOR gate performs the operation of XOR on all the 16 bits. The 16 bit XOR gives an output of 1 (HIGH) only if there are an odd number of 1 (HIGH) in the 16-Bit input.

- 16-Bit OR Gate

Fig. 2.10: 16 Bit OR gate

The 16-Bit OR gate is a collection of 15 OR gates connected in a cascaded manner. The least significant bits are given as input to the first OR gate, the output of which is given as the input to the next OR gate. This is how the OR gates are cascaded. The output of the 15th OR gate is the output of the 16 Bit OR gate configuration. This OR gate performs the operation of OR on all the 16 bits. The 16 bit OR gives an output of 1 (HIGH) if there is at least one 1 (HIGH) in the 16-bit input.

- 16-Bit NOT Gate

Fig. 2.11: 16 Bit NOT gate

The 16bit NOT gate is a collection of 16 NOT gates. The 16 Bit NOT gate is used to perform 1's complement of the 16 Bit input. It gives a 16-bit output for a 16-bit input.

- 16-Bit Full Adder Circuit

Fig. 2.12: 16 Bit Full Adder Circuit

The 16-bit full adder is the collection of 16 full adder circuits whose CARRY OUT are connected in a cascading manner. The least significant bits of the two 16 bit inputs are given as the inputs to the first full adder circuit. The sum is passed on to the output as the least significant bit of the SUM. The carry is cascaded into the next full adder circuit as the CARRY IN. The 16-bit full adder adds two 16 bit numbers and a CARRY IN. the output of the 16 Bit Full Adder circuit is a 16 Bit SUM and a one Bit CARRY OUT.

- 16-Bit 2:1 Multiplexer

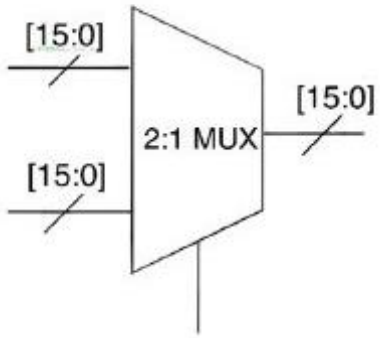

Fig. 2.13: 16 Bit 2:1 Multiplexer

A 2:1 Multiplexer consists of 4 MOSFETS and a NOT gate. The four MOSFETS consist of two PMOS and 2 NMOS MOSFETS. The $ MOSFETS are used to make 2 transmission gates and they are connected in series connecting the gates of a PMOS with the gate of the NMOS. The two inputs are given as the individual inputs to the transmission gates. The select line is given to the gates of the remaining gates of the transmission gates. The select line also is given as the input to the NOT gate, the output of which is given to the connecting gates. The outputs of the transmission gates are connected to each other and are taken as the output of the multiplexer. When the select line is high the upper transmission gate is conducting while the lower transmission gate is not. This gives the first input at the output. The opposite happens when the select line is low. When 16 of these Multiplexers are take the inputs of two 16 bit inputs, the arrangement works like a 16 bit 2:1 Multiplexer

- 16 Bit 4:1 Multiplexer

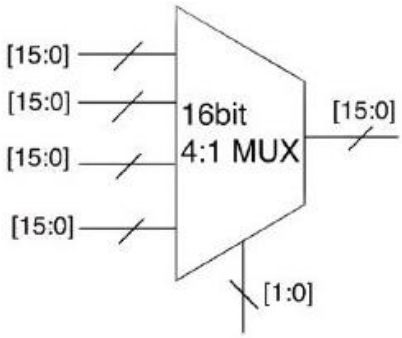

Fig. 2.14: 16 Bit 4:1 Multiplexer

A 4:1 Multiplexer is nothing but a combination of three 2:1 multiplexer in a particular connection configuration. The four inputs are given to the initial two Multiplexers with input1 and input2 being given to the multiplexer 1 and input 3 and input 4 being given to the second Multiplexer. The outputs of the first two Multiplexers are given as the inputs to the third Multiplexer. The select lines are given in the following way. The initial two multiplexers are given the same LSB of the select line, the MSB is given as the select line for the third multiplexer. When the select lines are 00, 01, 10, 11 the outputs are obtained as follows: input1, input2, input3 and input4 respectively. When 16 of suck multiplexers are brought in a configuration similar to 16 Bit 2:1 Multiplexer we get a 16 Bit 34:1 Multiplexer.

- 16-Bit 1:4 Demultiplexer

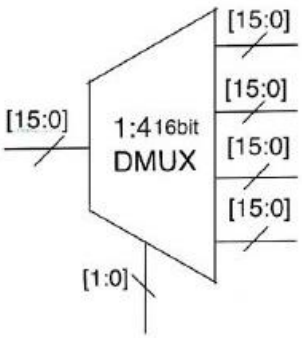

Fig. 2.15: 16 Bit 1:4 Demultiplexers

The 16 Bit 1:4 De-multiplexer is a combination of 16 individual 4 Bit 1:4 De-multiplexers. The De-multiplexer has a 16 Bit input, a 2 Bit select line and 4 - 16 Bit outputs which can be activated using the select line. Depending on the input of the select line, the input is propagated to one of the 4 outputs. The select line can assume one of the four following values - 00, 01, 10, 11. When the select line is 00, the input is propagated to the first output,

while the other inputs are deactivated. When the input is 01, the input is propagated to the second output. When the input is 10, the input is propagated through the third output. When the select line is 11, the input is propagated through the fourth output. This is the working of the 16 Bit 1:4 De multiplexer.

- 16-Bit 1:8 Demultiplexers

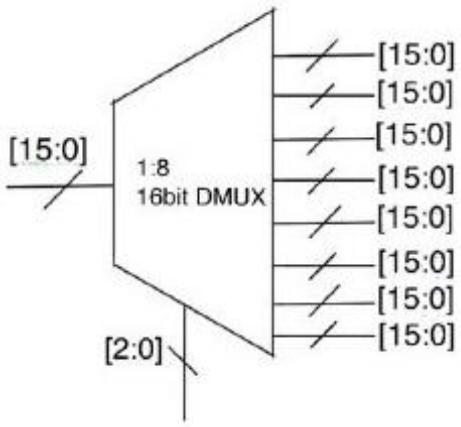

Fig. 2.16: 16 Bit 1:8 Demultiplexer

The 16 Bit 1:8 Demultiplexer is a combination of 16 individual 8 Bit 1:8 Demultiplexer. The Demultiplexer has a 16 Bit input, a 3 Bit select line and 8 - 16 Bit outputs which can be activated using the select ine. Depending on the input of the select line, the input is propagated to one of the 4 outputs. The select line can assume one of the eight following values - 000, 001, 010, 011…., 111. When the select line is 000, the input is propagated to the first output, while the other inputs are deactivated. When the input is 001, the input is propagated to the second output. When the input is 010, the input is propagated through the third output. When the select line is 011, the input is propagated through the fourth output and so on for the other values of the select line. This is the working of the 16 Bit 1:8 Demultiplexer.

- Register

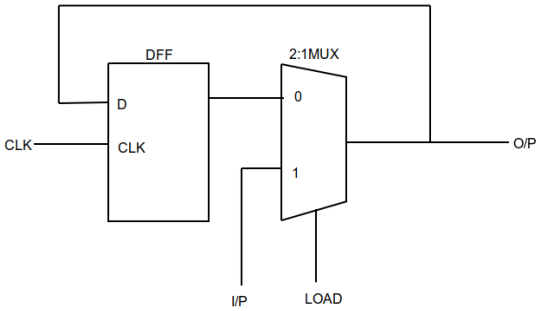

Fig. 2.17: Register

The register consists of a D-FlipFlop and a 2:1 multiplexer. When these registers are used in a collection of 16, having a common clk and load they form 16 bit registers. These registers store data into them when the clk and the load are high. The data of the register does not change even if one of the mentioned is low. These 16 bit registers are used as the A register and D register.
The A register is unique and determines the data that goes in as the input to the ALU, the value of Program Counter that points to the next instruction to be executed and the address location to which the CPU output gets loaded to the RAM. The D register is connected to the other input of the ALU.

- Decoder

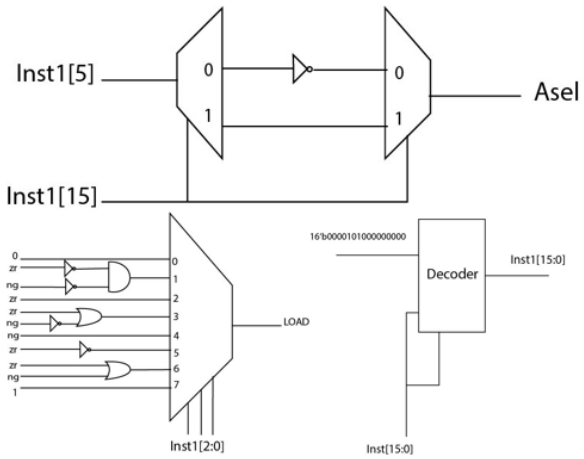

Fig. 2.18: Decoder

The decoder uses combinational logic coupled with multiplexer logic to decipher the instruction input from the ROM.
The decoder's output is given to the select lines of the multiplexers and sub-components of the CPU.
The decoder consists mainly of three primary components. The first component differentiates between the C instruction and A instruction. This part of the decoder is a 16 Bit 2:1 Mutlplexer with the MSB of the instruction acting as the select line. Once the instruction has been classified between a C instruction and an A instruction the 'c' bits of the instruction go to the ALU, the 'd' instruction bits go to the 'write enable' pins of the general purpose registers and the RAM. The 'j' bits are given to the Program Counter input. The next part of the decoder is responsible for the activation of the A register. When the instruction is an A instruction, the 'Asel' is used to write into the A register. The third and the final component of the decoder is responsible for activating the load pin of the program counter. Based on the output of the ALU, the 'zr' and 'ng' values are modified and passed to a 8:1 Multiplexer. The select lines of the multiplexer are the least significant 3 bit of the instruction (aka jum bits). The 'Load' line is given to the Program Counter.

Load is made high depending on the output of the ALU. The state of the two flag bits of the ALU, 'zr' and 'ng' determine if the Load line is to be made high or not.

- Program Counter

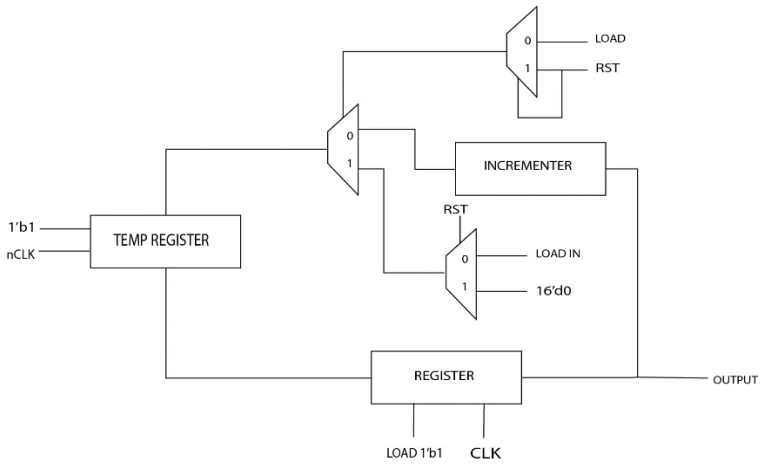

Fig. 2.19: Program counter

The program counter consists of two 16 bit registers, a 16bit adder being used as an incrementer, two 16 Bit 2:1 multiplexer and a single 2:1 multiplexer.
The Program Counter is used to determine which instruction of the ROM is executed next by the CPU. When the Program Counter is reset, it gets loaded with 0s. If the 'j' bits are 0, the PC increments by one. If the 'j' bits are non-zero, the value stored in A gets loaded into the PC. The figure below gives a visual description of the Program Counter.

- Arithmetic Logic Unit

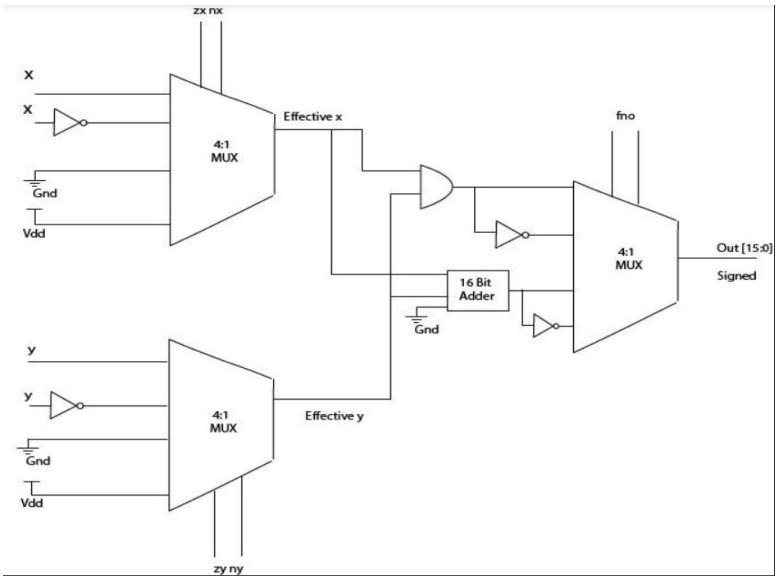

Fig. 2.20: ALU

The ALU consists of three 16 Bit 4:1 Multiplexers, two for the inputs X and Y and one for the output. It also has a 16 Bit and a couple of 16 Bit basic gates. The ALU has two 16 bit input busses and 6 control bits which are responsible for determining the kind of computation to be done on the two 16-bit bus inputs. The inputs X and Y can be taken in four different ways, X/Y, notX/notY, Vdd, Gnd. The input is selected based on the select lines of the respective multiplexer. The operations performed on these inputs can be of four types - X and Y, not(X and Y), X+Y and not(X+Y). The operation performed depends on the select lines of the output multiplexer.
The outputs of the ALU are of two types, one is the 16 But output from the multiplexer while the other output is a 2 Bit output indicating the character of the output. One Bit indicating if the output is a zero and the other bit indicating if the output is negative.
The ALU can perform 18 different operations using these components.
The function of the control bits is given below. The control pins 'zx', 'nx', 'zy', 'ny', 'f', 'no' determine what computation is to be done with the two 16 bit inputs. 'zx' makes the 'x' input 0, 'nx' complements the input. 'Zy' and 'ny' perform similar actions on the 'y' input. 'f' and 'no' are responsible for 'zr' and 'ng' are output flag bits that determine if the output is zero and if it is negative respectively.

- CPU

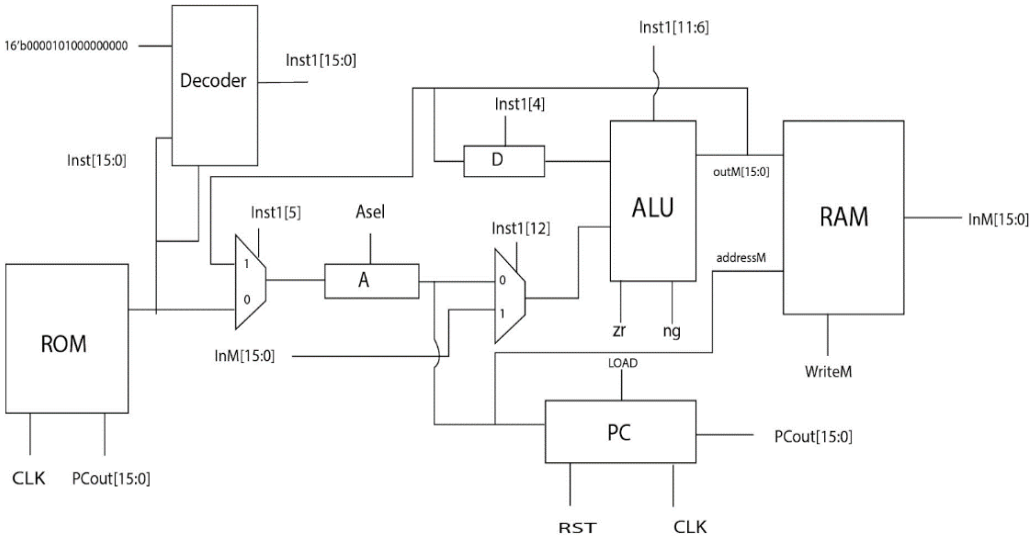

Fig. 2.20: CPU

The computer architecture is designed keeping in mind the hack language. The entire computer architecture consists of three major blocks, the RAM, ROM and CPU. The CPU deals with 16 bit operands at every positive edge instance of a clock. The programs written in the hack language are converted into binary values where sets of 16 bit binary numbers represent an instruction. These instructions are stored in the ROM. The RAM is initially

empty and is capable of storing the CPU's output. The RAM also gives an input to the CPU depending on the instruction given to the CPU by the ROM. The CPU's program counter is responsible for selecting which instruction is to be executed in the next machine cycle of the CPU. The RAM has a write enable pin that decides if the CPU's output is to be storedIn the RAM or Not. The CPU consists of a decoder, two general purpose registers A and D, an ALU and a Program Counter. All CPU components are designed using multiplexer logic. There are multiplexers present in between the fundamental components of the CPU which are controlled by the decoder. The decoder uses combinational logic coupled with multiplexer logic to decipher the instruction input from the ROM. The decoder distinguishes the input instruction into the A type or the C type. The decoder's output is given to the select lines of the multiplexers and sub components of the CPU. The 'c' bits of the instruction go to the ALU, the 'd' instruction bits go to the 'write enable' pins of the general purpose registers and the RAM. The 'j' bits are given to the Program Counter input. 'Asel' is used to write into the A register. The 'Load' line is given to the Program Counter. Load is made high depending on the output of the ALU. The state of the two flag bits of the ALU, 'zr'and 'ng' determine if the Load line is to be made high or not. The A register is unique and determines the data that goes in as the input to the ALU, the value of Program Counter that points to the next instruction to be executed and the address location to which the CPU output gets loaded to the RAM. The D register is connected to the other input of the ALU.The Program Counter is used to determine which instruction of the ROM is executed next by the CPU. When the Program Counter is reset, it gets initially loaded with 0s. If the 'j' bits are 0, the PC increments by one. If the 'j' bits are non-zero, the value stored inA gets loaded into the PC. The figure below gives a visual description of the Program Counter. The ALU has two 16 bit input busses and 6 control bits which are responsible for determining the kind of computation to be done on the two 16-bit bus inputs. The function of the control bits is given below. The control pins 'zx', 'nx', 'zy', 'ny', 'f', 'no' determine what computation is to be done with the two 16 bit inputs. 'zx' makes the 'x' input 0, 'nx' complements the input. 'Zy' and 'ny' perform similar actions on the 'y' input. 'f' and 'no' are responsible for 'zr' and 'ng' are output flag bits that determine if the output is zero and if it is negative respectively.

CHAPTER 3

HARDWARE/ SOFTWARE TOOLS INTERFACING

Software Tools:

- Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991.Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.
Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.
Python was conceived in the late 1980s as a successor to the ABC language. Python 2.0, released in 2000, introduced features like list comprehensions and a garbage collection system capable of collecting reference cycles. Python 3.0, released in 2008, was a major revision of the language that is not completely backward-compatible. Python interpreters are available for many operating systems. A global community of programmers develops and maintains Python, an open source reference implementation.

- Xilinx Vivado HLx

The Vivado Design Suite HLx edition supplies design teams with the tools and methodology needed to leverage C-based design and optimized reuse, IP sub-system reuse, integration automation and accelerated design closure. It is proven to accelerate productivity by enabling designers to work at a high level of abstraction while facilitating design reuse.
Vivado also provides:

- Software-defined IP Generation with Vivado High-Level Synthesis
- Block-based IP Integration with Vivado IP Integrator
- Vivado Logic Simulation
- Integrated Mixed Language Simulator
- Integrated & Standalone Programming and Debug Environments
- Verification IP
- 4X Faster Implementation
- Up to 3-Speedgrade Performance Advantage for the low-end & mid-range and 35% Power Advantage in the high-end

- Xilinx ISE

The ISE Design Suite: Embedded Edition includes Xilinx Platform Studio (XPS), Software Development Kit (SDK), large repository of plug and play IP including MicroBlaze™ Soft Processor and peripherals, and a complete RTL to bit stream design flow. Embedded Edition provides the fundamental tools, technologies and familiar design flow to achieve optimal design results. These include intelligent clock gating for dynamic power reduction, team design for multi-site design teams, design preservation for timing repeatability, and a partial reconfiguration option for greater system flexibility, size, power, and cost reduction.

- Quartus Prime

The revolutionary Intel Quartus Prime Design Software includes everything you need to design for Intel FPGAs, SoCs, and complex programmable logic device (CPLD) from design entry and synthesis to optimization, verification, and simulation. Dramatically increased capabilities on devices with multi-million logic elements are providing designers with the ideal platform to meet next-generation design opportunities.

Hardware Tools :

- Xilinx Spartan 6 FPGA

Spartan®-6 devices offer industry-leading connectivity features such as high logic-to-pin ratios, small form-factor packaging, MicroBlaze™ soft processor, and a diverse number of supported I/O protocols. Ideally suited for a range of advanced bridging applications found in consumer, automotive infotainment, and industrial automation.
Specifications

Programmable System Integration	• High pin-count to logic ratio for I/O connectivity • Over 40 I/O standards for simplified system design • PCI Express® with integrated endpoint block
Increased System Performance	• Up to 8 low power 3.2Gb/s serial transceivers • 800Mb/s DDR3 with integrated memory controller
BOM Cost Reduction	• Cost-optimized for system I/O expansion • MicroBlaze™ processor soft IP to eliminate external processor or MCU components

Total Power Reduction	• 1.2V core voltage or 1.0V core voltage option • Zero power with hibernate power-down mode
Accelerated Design Productivity	• Enabled by ISE® Design Suite—a no-cost, front-to-back FPGA design solution for Linux and Windows • Fast design closure using integrated wizards

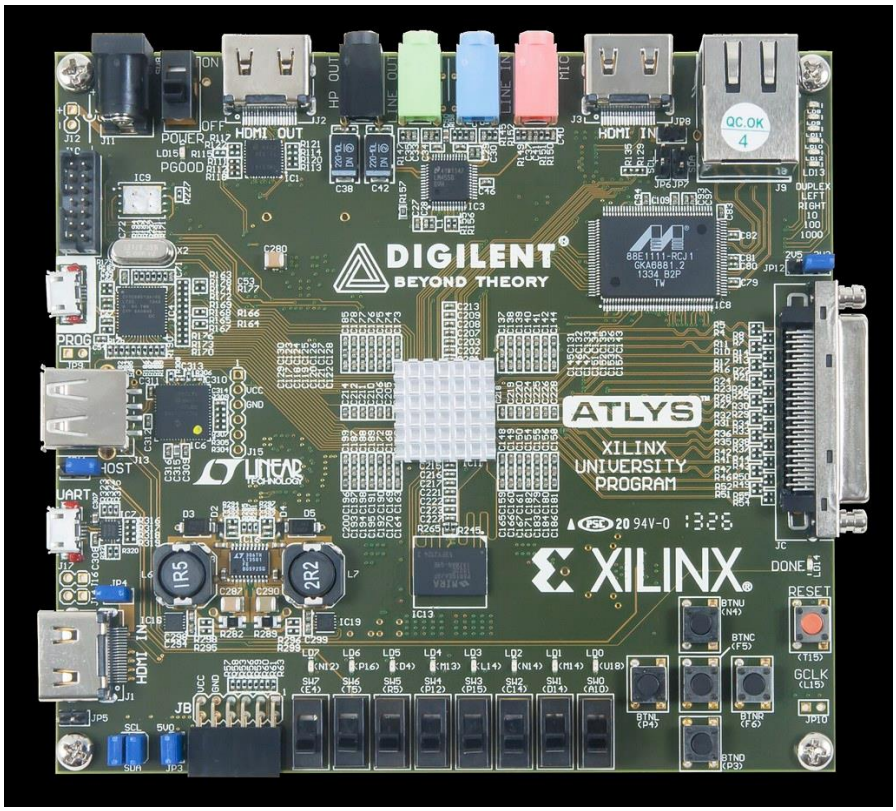

Fig. 3.1: Xilinx Spartan 6 FPGA

Software tools created and used:

To compile the assembly level code and create its equivalent machine code we created a compiler in the Python programming language.
The assembler takes a program source code file written in the Hack Assembly Language, which is a *.asm text file, and then assembles it into binary machine code. The assembled machine code program is then written to a new *.hack text file with the same name. The content is then written into the rom.v so as to be executed by the CPU.
The Assembling process is implemented in two passes. The first pass scans the whole program, registering the labels only in the Symbol Table. The second pass scans the whole

program again, registering all variables in the Symbol Table, substituting the symbols with their respective memory and/or instruction addresses from the Symbol Table, generating binary machine code and then writing the assembled machine code to the new *.hack text file.
Source code is organized into several components, the decisions for their names, interfaces and APIs were already specified in the book as sort of a specification-implementation contract. All components of the Assembler reside in the **/Assembler** directory, as follows:

1. **Assembler.py**: Main module. Implements the two passes and glues the other components together.
2. **Parser.py**: Simple Parser. Parses the instructions by looking ahead 1 or 2 characters to determine their types and structures.
3. **Lex.py**: A simple Lexer which is used by the Parser to break an instruction to smaller parts and structure it in a way that makes it easy to convert it to machine code.
4. **Code.py**: Generates binary machine code for instructions. For C-Instructions, it generates machine code for its constituting parts and then merges them back altogether.
5. **SymbolTable.py**: Implements a lookup table which is used to register symbols (labels and variables) and look up their memory addresses.

How to use the compiler program :

Write the assembly code in a *.asm file
Run the python asse_to_v.py script using the following command
python3 asse_to_v.py <assembly_file_name>

Assembly program	Machine code
@R0	0000000000000000
D=M	1111110000010000
@R1	0000000000000001
D=D-M	1111010011010000
@OUTPUT_FIRST	0000000000001010
D;JGT	1110001100000001
@R1	0000000000000001
D=M	1111110000010000
@OUTPUT_D	0000000000001100
0;JMP	1110101010000111
	0000000000000000
(OUTPUT_FIRST)	1111110000010000
	0000000000000010

@R0 D=M (OUTPUT_D) @R2 M=D (INFINITE_LOOP) @INFINITE_LOOP 0;JMP	1110001100001000 0000000000001110 1110101010000111

The Hack Assembly Language is minimal, it mainly consists of 3 types of instructions. It ignores whitespace and allows programs to declare symbols with a single symbol declaration instruction. Symbols can either be labels or variables. It also allows the programmer to write comments in the source code, for example: `// this is a single line comment`.
Predefined Symbols

- **A**: Address Register.
- **D**: Data Register.
- **M**: Refers to the register in Main Memory whose address is currently stored in **A**.
- **SP**: RAM address 0.
- **LCL**: RAM address 1.
- **ARG**: RAM address 2.
- **THIS**: RAM address 3.
- **THAT**: RAM address 4.
- **R0**-**R15**: Addresses of 16 RAM Registers, mapped from 0 to 15.
- **SCREEN**: Base address of the Screen Map in Main Memory, which is equal to 16384.
- **KBD**: Keyboard Register address in Main Memory, which is equal to 24576.

Types of Instructions:

1. A-Instruction: Addressing instructions.
2. C-Instruction: Computation instructions.
3. L-Instruction: Labels (Symbols) declaration instructions.

A-INSTRUCTIONS:
Symbolic Syntax:
`@value`, where value is either a decimal non-negative number or a Symbol.

Examples:

- `@21`
- `@R0`
- `@SCREEN`

Binary Syntax:
`0xxxxxxxxxxxxxx`, where `x` is a bit, either 0 or 1. A-Instructions always have their MSB set to 0.
Examples:

- `0000000000001010`
- `0111111111111111`

Effects:
Sets the contents of the **A** register to the specified value. The value is either a non-negative number (i.e. 21) or a Symbol. If the value is a Symbol, then the contents of the **A** register is set to the value that the Symbol refers to but not the actual data in that Register or Memory Location.

L-INSTRUCTIONS:
Symbols can be either variables or lables. Variables are symbolic names for memory addresses to make remembering these addresses easier. Labels are instruction addresses that allow multiple jumps in the program easier to handle. Symbols declaration is not a machine instruction because machine code doesn't operate on the level of abstraction of that of labels and variables, and hence it is considered a pseudo-instruction.

Declaring Variables:
Declaring variables is a straightforward A-Instruction, example:
@i
M=0
The instruction `@i` declares a variable "i", and the instruction `M=0` sets the memory location of "i" in Main Memory to 0, the address "i" was automatically generated and stored in **A** Register by the instruction.
Declaring Labels:
To declare a label we need to use the command `(LABEL_NAME)`, where "LABEL_NAME" can be any name we desire to have for the label, as long as it's wrapped between parentheses. For example:

```
(LOOP)
  // ...
  // instruction 1
  // instruction 2
  // instruction 3
  // ...
```

```
@LOOP
0;JMP
```

The instruction `(LOOP)` declares a new label called "LOOP", the assembler will resolve this label to the address of the next instruction (A or C instruction) on the following line.
The instruction `@LOOP` is a straight-forward A-Instruction that sets the contents of **A** Register to the instruction address the label refers to, whereas the `0;JMP` instruction causes an unconditional jump to the address in **A** Register causes the program to execute the set of instructions between `(LOOP)` and `0;JMP` infinitely.
C-INSTRUCTIONS:
Symbolic Syntax:
dest = *comp* ; *jmp*, where:

1. *dest*: Destination register in which the result of computation will be stored.
2. *comp*: Computation code.
3. *jmp*: The jump directive.

Examples:

- `D=0`
- `M=1`
- `D=D+1;JMP`
- `M=M-D;JEQ`

Binary Syntax:
`1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`, where:

- `111` bits: C-Instructions always begin with bits `111`.
- `a` bit: Chooses to load the contents of either **A** register or **M** (Main Memory register addressed by **A**) into the ALU for computation.
- Bits `c1` through `c6`: Control bits expected by the ALU to perform arithmetic or bitwise logic operations.

* Bits `d1` through `d3`: Specify which memory location to store the result of ALU computation into: **A**, **D** or **M**.
* Bits `j1` through `j3`: Specify which JUMP directive to execute (either conditional or unconditional).

CHAPTER 4

ALGORITHM AND FLOWCHART

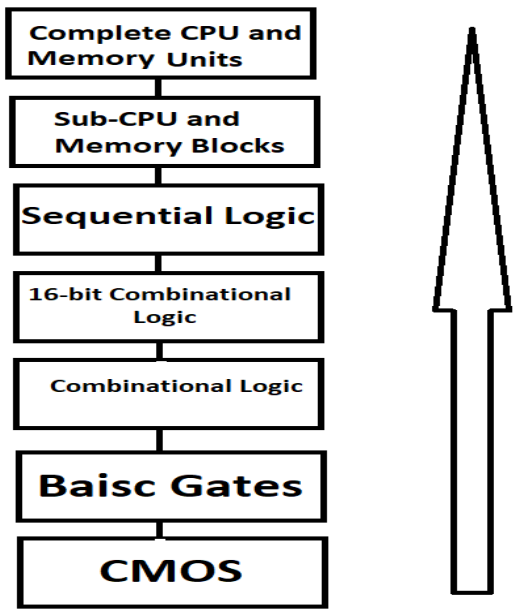

Fig. 4.1: Bottom-Up Approach used in the computer design

A bottom-up approach is used to design the computer. The most basic unit of the computer is a CMOS. The entire computer is designed using CMOS logic. The CMOS is designed using 45nm MOSFETs. The bottom up approach involves the creation of larger logic units from the CMOS. The CMOS is used to make Basic Gates. Basic gates are used to make combinational logic such as half adders and full adders. The 16- bit version of these combinational logic circuits are used for the CPU design as the CPU deals with 16 bit instructions. Sequential Logic is then designed using basic gates. The use of both Sequential and Combinational Logic is required to design the sub CPU and memory blocks such as the RAM, Program Counter, ALU, etc. The sub CPU and Memory blocks are combined to create the 16-Bit hack computer.

CHAPTER 5

RESULTS AND DISCUSSIONS

In this project, we describe the design of the processor by laying out and explaining how it is built using different low-level chips. The basic gates required are initially designed using CMOS logic and using those designs, the higher abstractions have been made. All the components are designed and are simulated using Xilinx ISE. The Verilog codes are compatible with FPGA (Spartan -6).

@2	0000000000000010
D=A	1110110000010000
@7	0000000000000111
D=D+A	1110000010010000

Table 5.1: Addition of two numbers

Let us consider the small snippet of code given above and try to understand the functioning of the Hack Processor. Initially, the Hack code is written and stored in an assembly file. The ".asm" file is run through an assembler to get the binary values that represent each instruction of the program as shown in the table above. These sets of 16-bit binary values are stored in each register of the ROM. The Program Counter is initially reset to point to zero. The instruction in ROM[0] is initially read with the binary value representing the instruction "@2". This instruction denotes an A-Instruction where the value 16'd2 is to be stored into the A register. The binary value of this instruction first enters the decoder. The decoder outputs a value of 16'b0000101000000000 which disables the "Load" pin of the Program Counter, enables the "Write" pin of A register, disables the "Write" pin of D register and RAM and causes the ALU to output a value of zero by enabling the "zx" and "zy" pins. This causes the A pin to load the value 2 into it.

For the second instruction to be executed, the Program Counter must point to ROM[1]. In the previous instruction since, the "load" pin of the Program Counter is disabled, it by default increments by 1. In the second instruction, D=A implies that the value 2, stored in

the A register is to be stored in the D register. This is an instruction of the C type. The decoder receives this instruction from the ROM and leads to the A register getting disabled, the Program counter to increment by a value of one, the D register's "Write" pin is enabled, the RAM is disabled and the ALU's output is made to transfer the data in the A register to the D register. Just like the first instruction, the third instruction is of the A type. The behaviour of the CPU is similar to that of the first instruction. Here, the decimal value '7' is stored in the A register and the Program Counter is incremented by one.

The fourth instruction is C instruction. Here the value stored in the A register is to be added with the one in the D register. The output of this computation is available at the output of the ALU and this output is fed back to the D register.

Some of the logic gates presented here are typically referred to as "elementary" or "basic." At the same time, every one of them can be composed from NAND gates alone. Therefore, they need not be viewed as Primitive.

Truth tables of the different basic gates:

NOT

x	x'
0	1
1	0

AND

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

Table 5.2: Truth Tables

- BASIC COMPONENTS

1. AND GATE

The And function returns 1 when both its inputs are 1, and 0 otherwise.

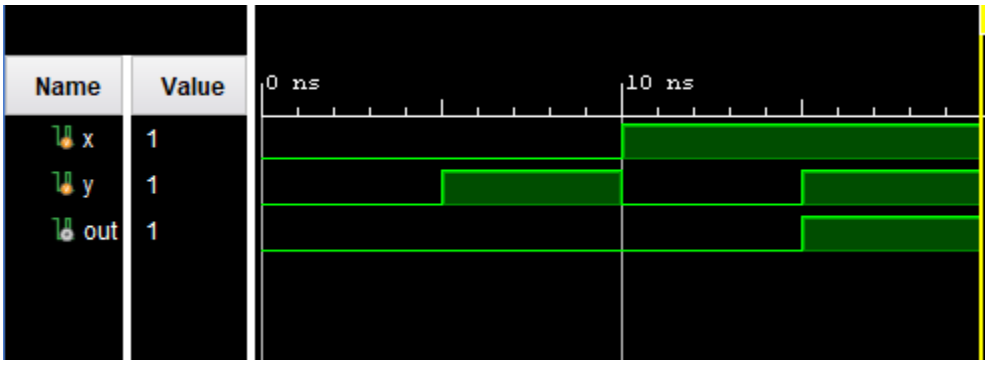

Fig 5.1: Waveform of AND GATE

2. OR GATE

The Or function returns 1 when at least one of its inputs is 1, and 0 otherwise.

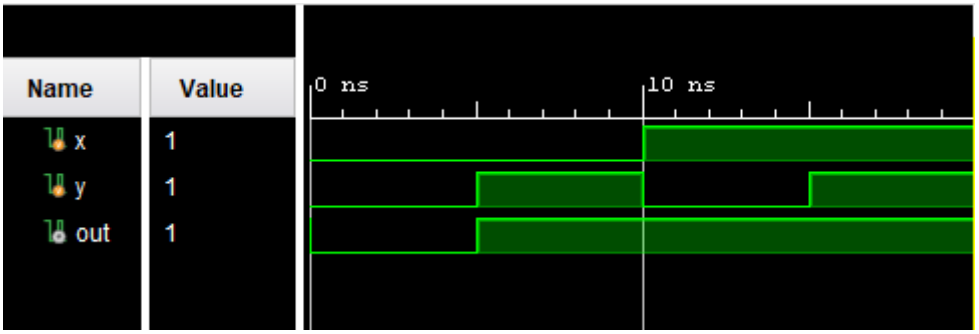

Fig. 5. 2: Waveform of OR Gate

3. NOT GATE

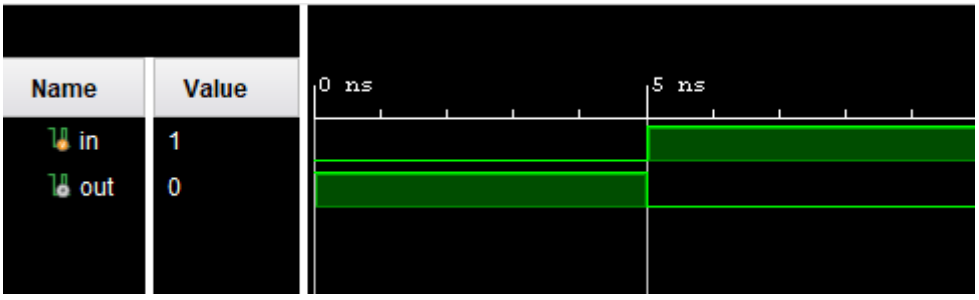

Fig. 5.3: Waveform of NOT GATE

4. XOR GATE

The Xor function, also known as "exclusive or," returns 1 when its two inputs have opposing values and 0 otherwise.

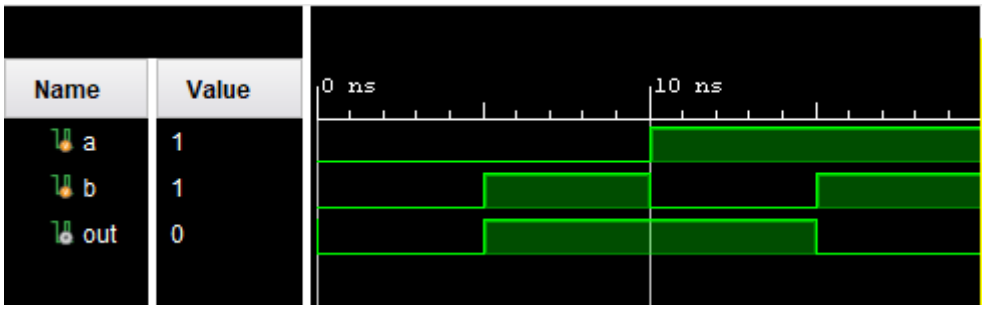

Fig. 5.4: Waveform of XOR GATE

5. Multiplexer: Multiplexer is a three-input gate that uses one of the inputs, called "selection bit," to select and output one of the other two inputs, called "data bits." Thus, a better name for this device might have been selector. The name multiplexor was adopted from communications systems, where similar devices are used to serialize (multiplex) several input signals over a single output wire.

a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

sel	out
0	a
1	b

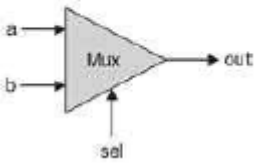

Table 5.3: Multiplexer

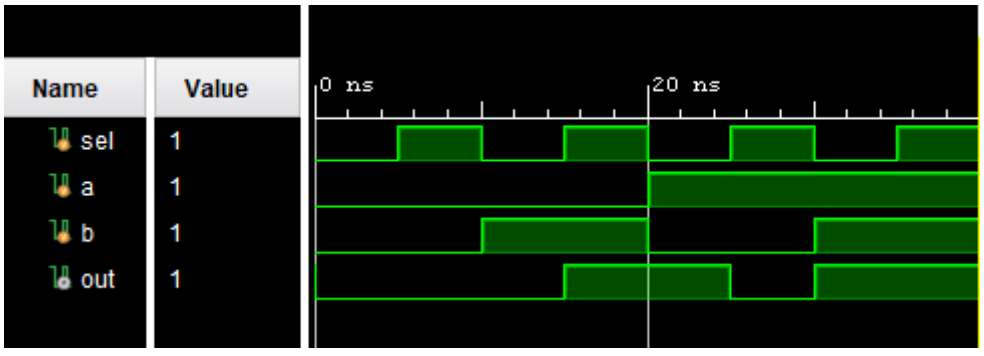

Fig 5.5: Waveform of 2:1 MUX

6. Demultiplexer

sel	a	b
0	in	0
1	0	in

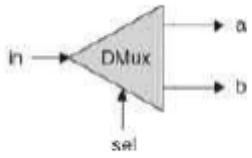

Table 5.5: Demultiplexer

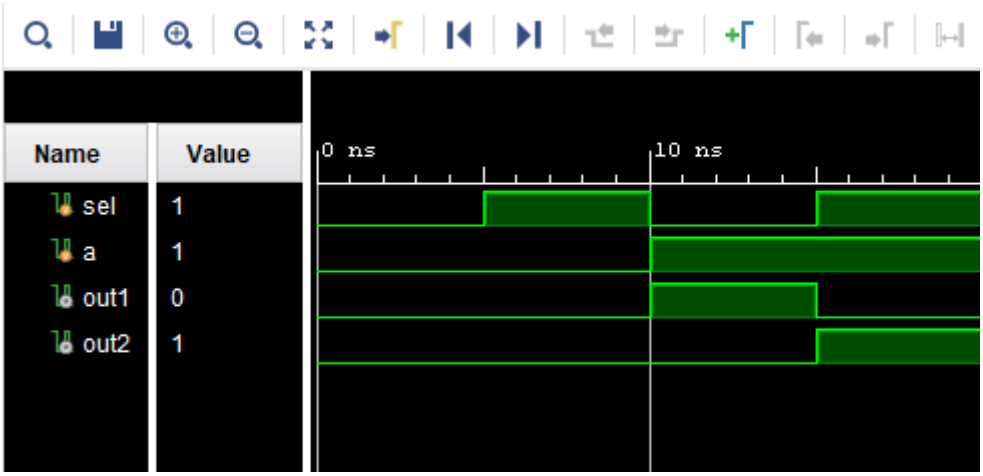

Fig 5.6: Waveform of 1:2 DEMUX

Multi Bit Versions of the above Components: Computer hardware is typically designed to operate on multi-bit arrays called "buses." For example, a basic requirement of a 32-bit computer is to be able to compute (bit-wise) an AND function on two given 32-bit buses. To implement this operation, we can build an array of 32 Binary AND gates, each operating separately on a pair of bits. In order to enclose all this logic in one package, we can encapsulate the gates array in a single chip interface consisting of two 32-bit input buses and one 32-bit output bus. This section describes a typical set of such multi-bit logic gates, as needed for the construction of a typical 16-bit computer. We note in passing that the architecture of n-bit logic gates is basically the same irrespective of n's value. When referring to individual bits in a bus, it is common to use an array syntax. For example, to refer to individual bits in a 16-bit bus named data, we use the notation data [0], data [1],..., data[15].

1. 16-Bit Not:

An n-bit Not gate applies the Boolean Operation Not to every one of the bits in its n-bit input bus.

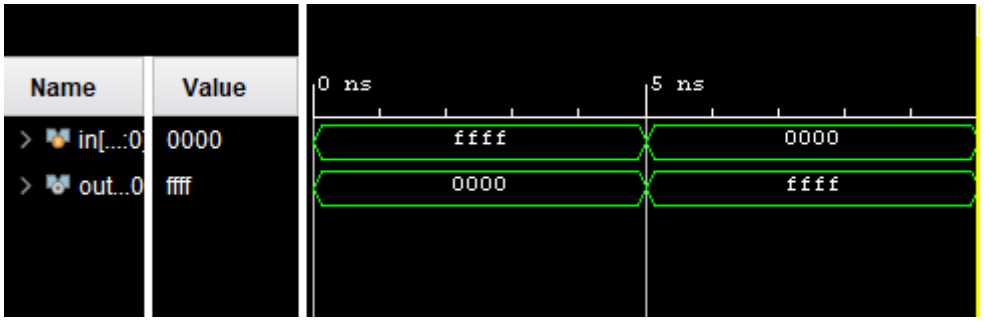

Fig.5.7: Waveform of 16-Bit NOT GATE

2. 16-Bit AND: An n-bit AND gate applies the Boolean Operation AND to every one of the n bit-pairs arrayed in its two n-bit input buses

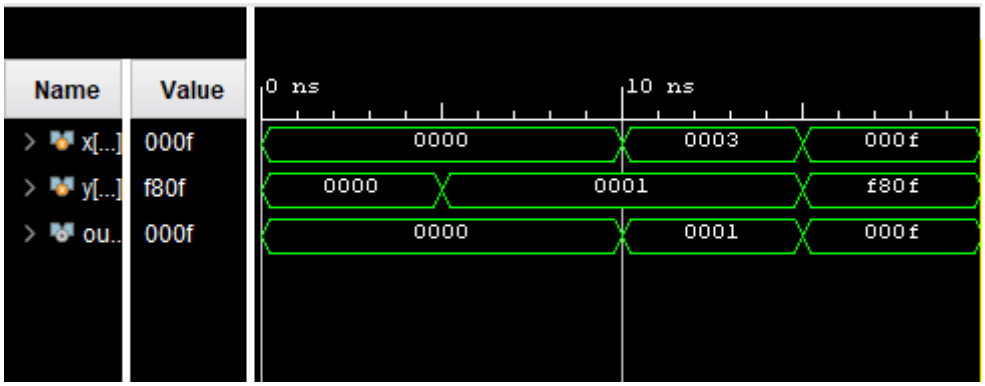

Fig.5.8: Waveform of 16-Bit AND GATE

3. OR Gate:

- OR 16

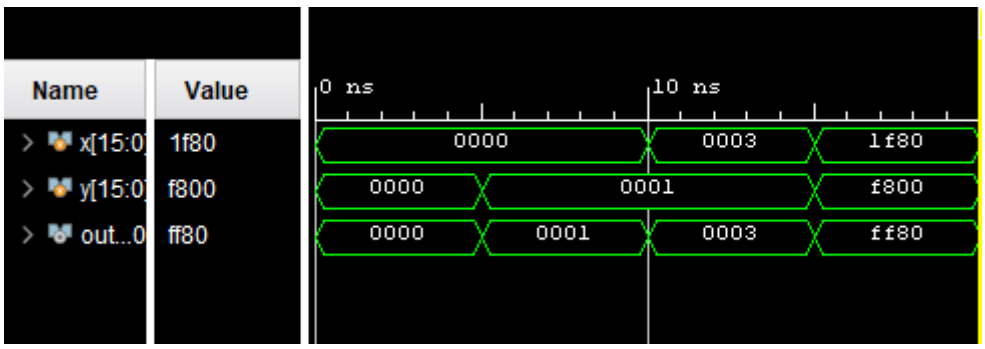

Fig. 5.9: 16-Bit OR Gate

- OR 8

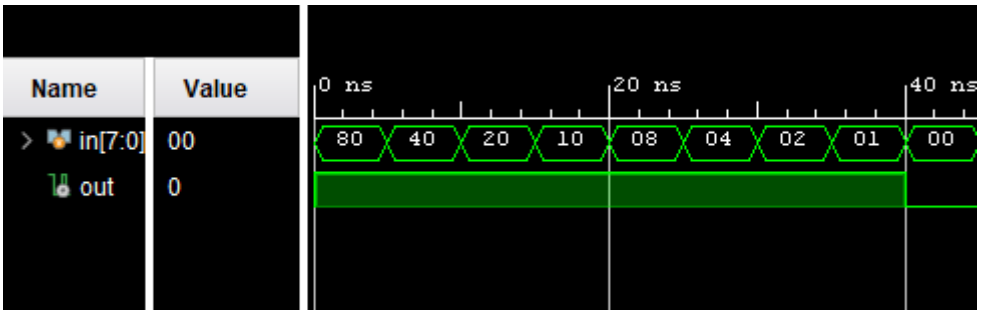

Fig. 5.10: Waveform of 8-Bit OR Gate

4. Multiplexer: An n-bit multiplexor is exactly the same as the binary multiplexor described in, except that the two inputs are each n-bit wide; the selector is a single bit.

Multi-Way/Multi-Bit Multiplexor: An m-way n-bit multiplexor selects one of m n-bit input buses and outputs it to a single n-bit output bus. The project requires two variations of this chip: A 4-way 16-bit multiplexor and an 8-way 16-bit multiplexor:

sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

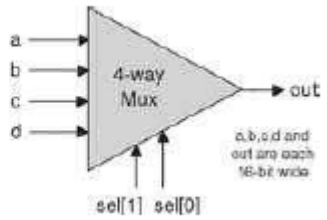

Table 5.6: 4:1 Multiplexer

- 16-Bit 4:1 Multiplexer

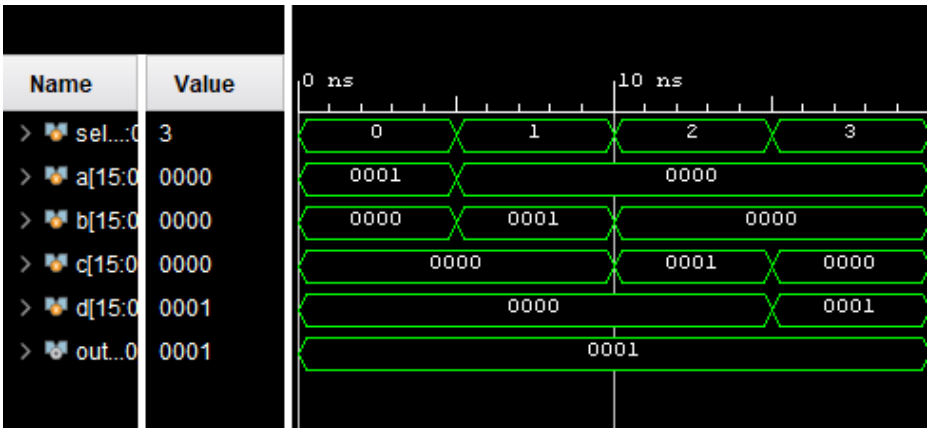

Fig. 5.11: 16-Bit 4:1 Multiplexer

- 16-Bit Multiplexer

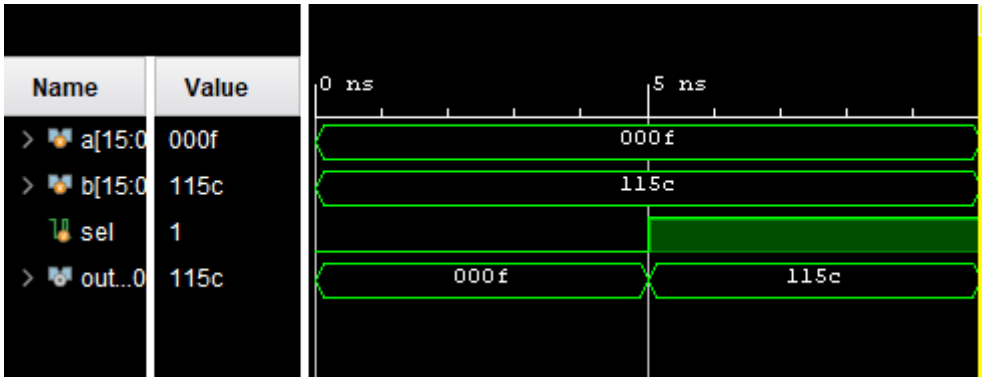

Fig. 5.12: 16-Bit Multiplexer

- 8-Bit Multiplexer

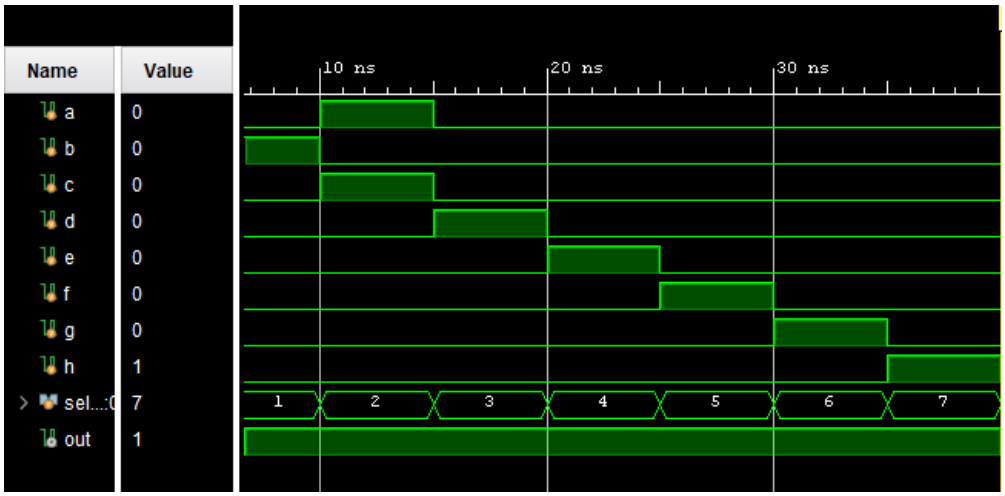

Fig. 5.13: 8-Bit Multiplexer

5. Demultiplexer: An m-way n-bit demultiplexer channels a single n-bit input into one of m possible n-bit outputs.

sel[1]	sel[0]	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

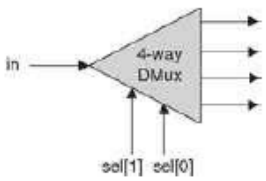

Table 5.5: 1:1 Demultiplexer

- 16-Bit 1:4 Demultiplexer

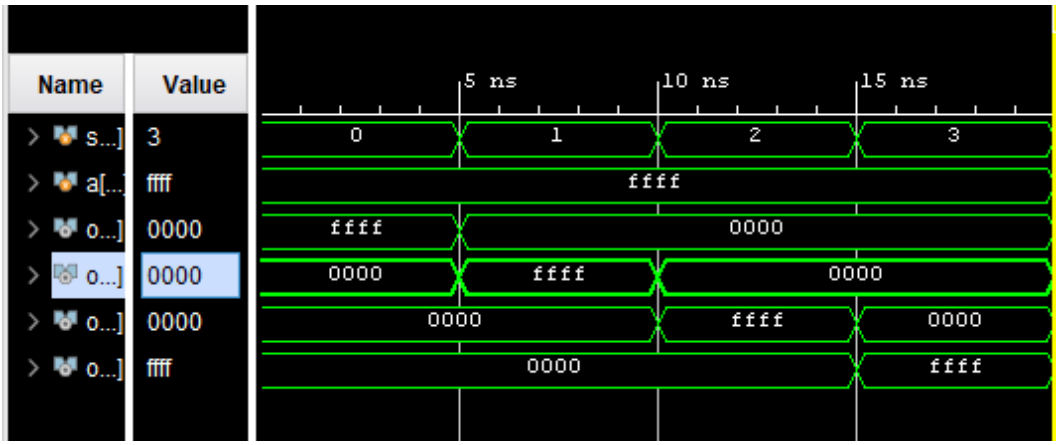

Fig. 5.14: 16-Bit 1:4 Demultiplexer

- 16-Bit 1:8 Demultiplexer

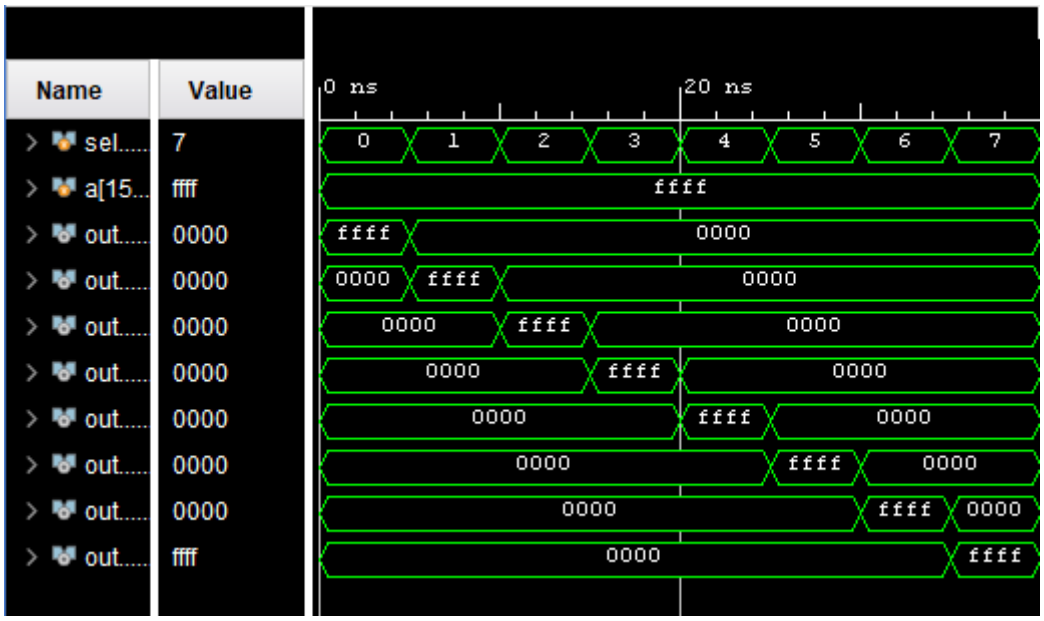

Fig. 5.15: 16-Bit 1:8-Bit Demultiplexer

- 16-Bit Demultiplexer

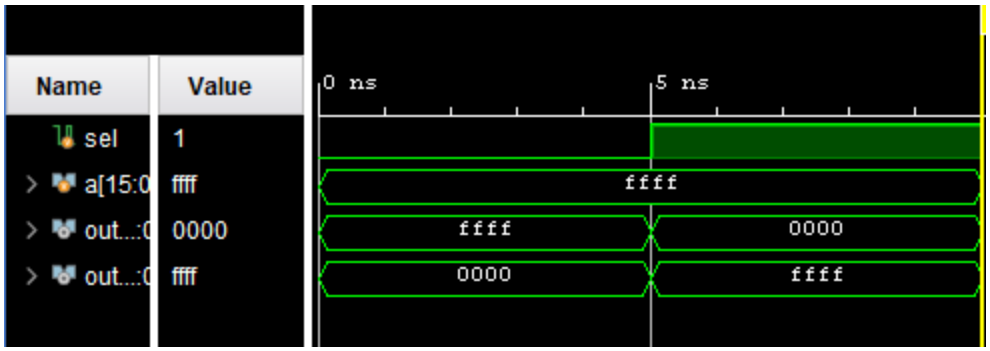

Fig. 5.17: 16-Bit Demultiplexer

6. D-Flip Flop: The D flip-flop tracks the input, making transitions with match those of the input D. The D stands for "data"; this flip-flop stores the value that is on the data line. It can be thought of as a basic memory cell.

- D Flip Flop

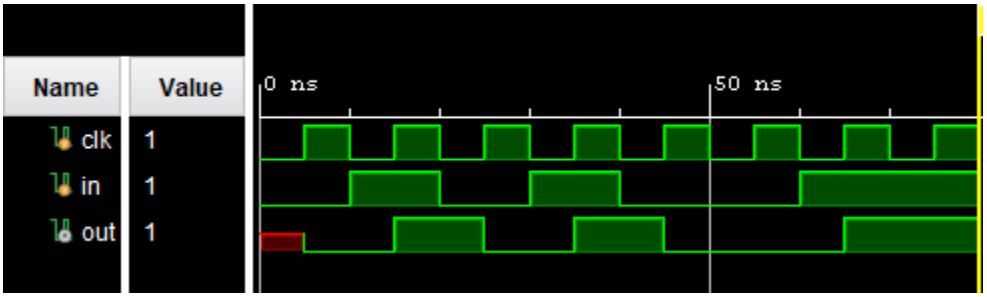

Fig. 5.18: D Flip Flop

- 16-Bit D Flip Flop

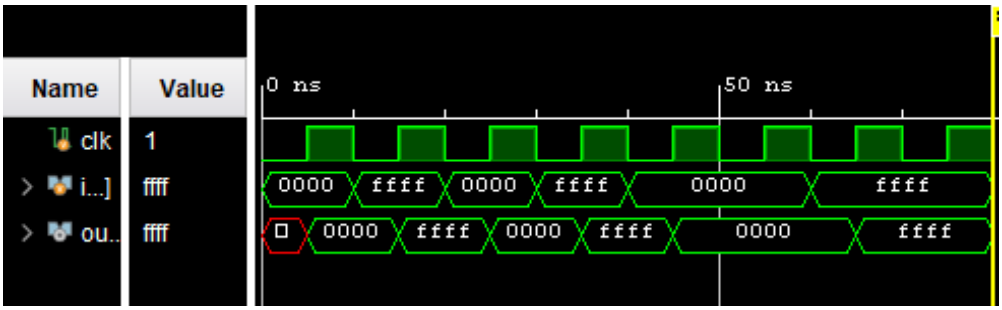

Fig. 5.19: 16-Bit D Flip Flop

6. Adders: An adder is a digital circuit that performs addition of numbers. In many computers and other kinds of processors adders are used in the arithmetic logic units or ALU.

The half adder adds two binary digits called as augend and addend and produces two outputs as sum and carry; XOR is applied to both inputs to produce sum and AND gate is applied to both inputs to produce carry.

- Half adder

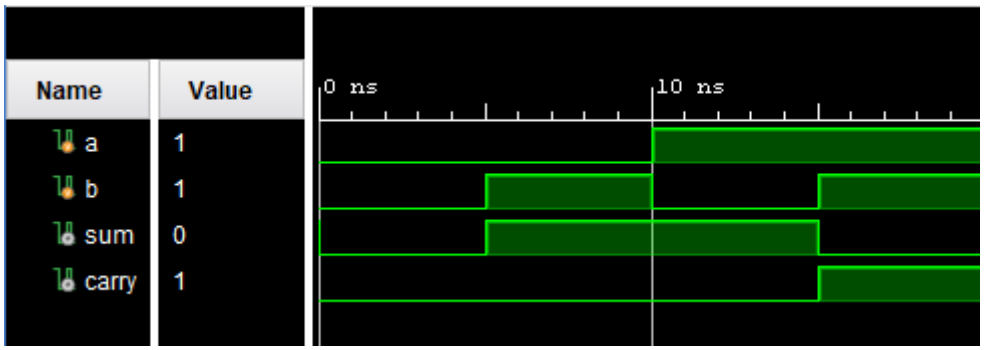

Fig. 5.20: Half Adder

Full Adder is the adder which adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM.

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

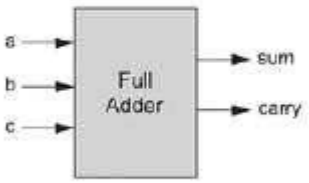

Table 5.8: Full Adder

- FULL ADDER

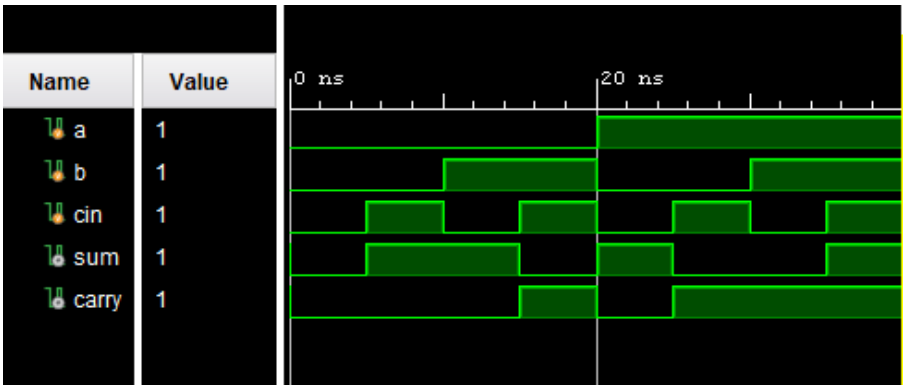

Fig. 5.21: Full Adder

- 16-BIT FULL ADDER

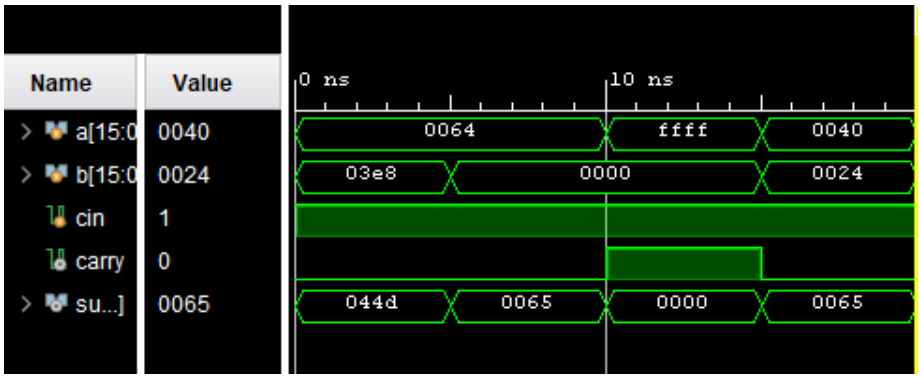

Fig. 5.22: 16-Bit Full Adder

OTHER COMPONENTS:

- Incrementer: Used in the program counter to increment by one position.

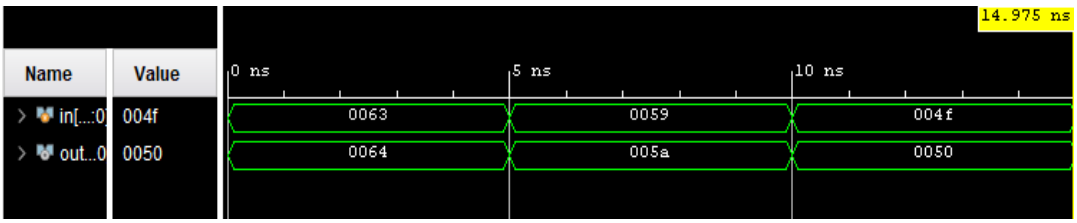

Fig. 5.23: Incrementer

- IS EQUAL TO: This component checks if the two data input values are equal to each other or not.

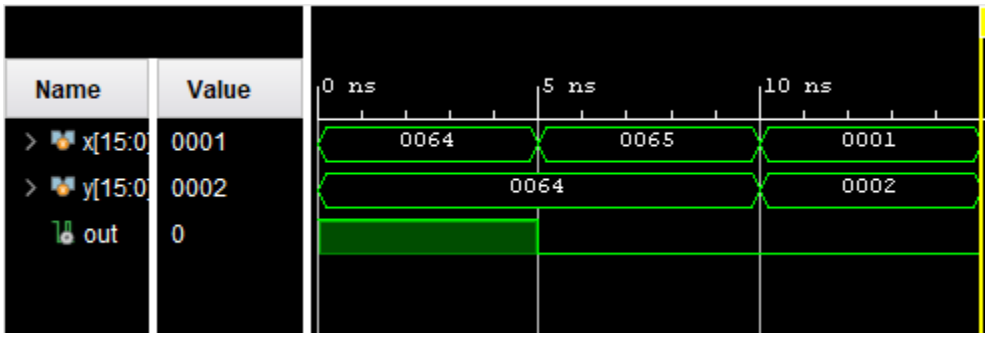

Fig. 5.24: Is Equal To

- LOAD

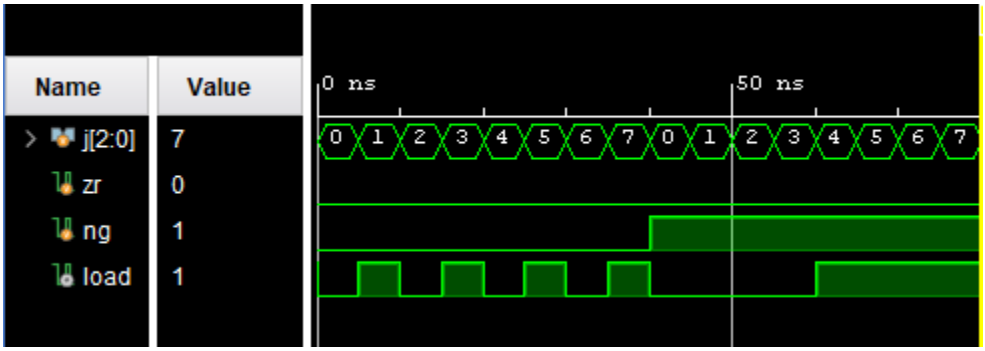

Fig. 5.25: Load

After all the above components were designed, they were integrated in order to form different components such as ALU, CPU, Program Counter and Decoder.

- Arithmetic Logical Unit

Basic components such as adders and logical gates are used in order to create the ALU. The ALU is capable of handling different arithmetic and Boolean operations such as addition, subtraction multiplication, AND, OR, NOR, NAND, XOR.

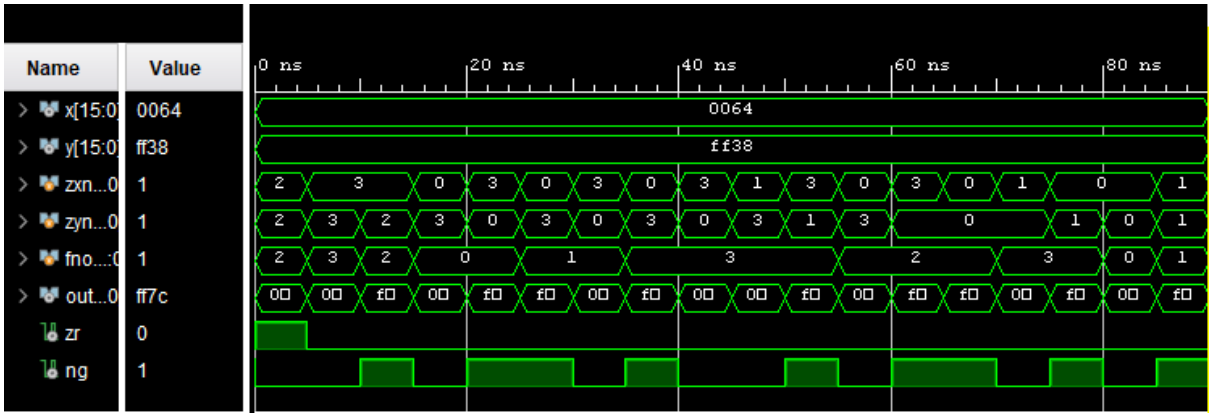

Fig. 5.26: Result of Arithmetic Logical Unit

- Program Counter

Out of the several registers present in the processor, PC is the register present in the processor which holds the address location of the instruction being executed and gets incremented to the next address location as the next instruction gets fetched for execution.

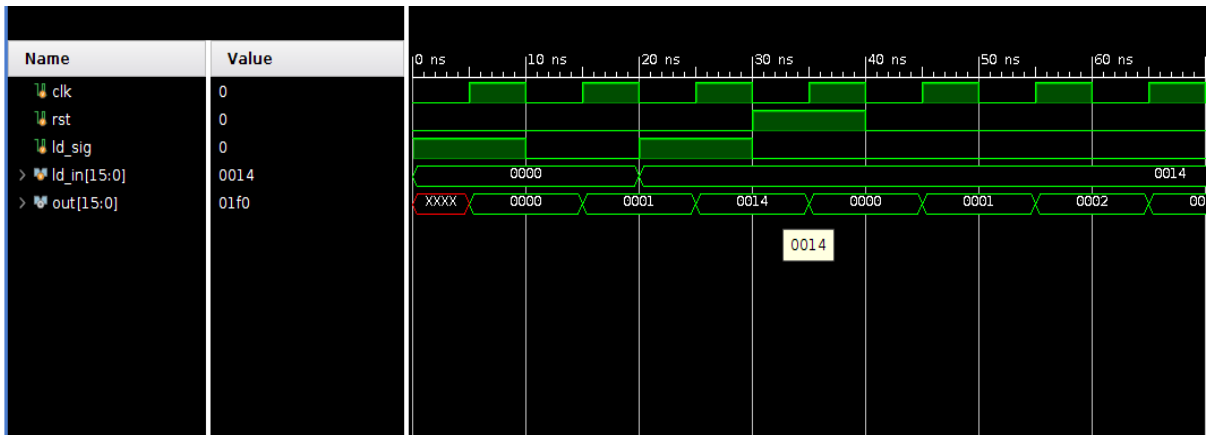

Fig. 5.27: Result of Program Counter

- Decoder

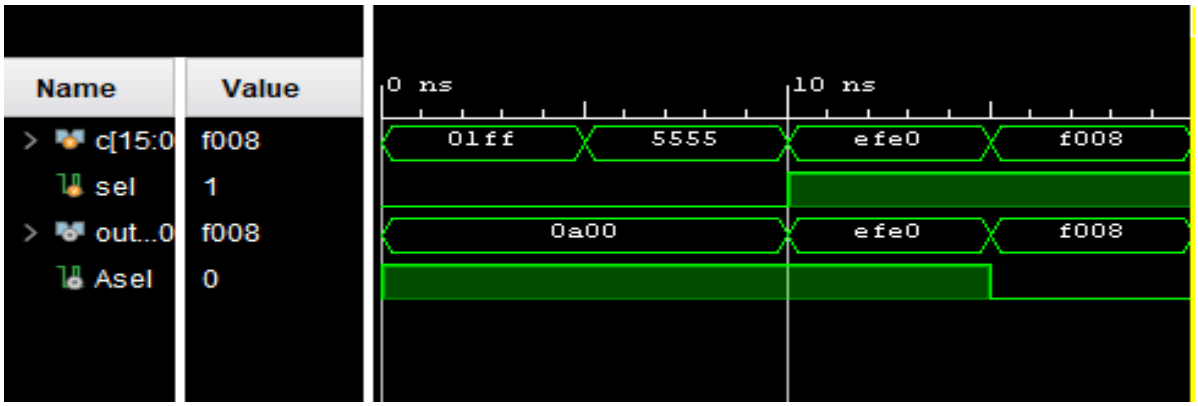

Fig. 5.28. Result of Decoder

- Central Processing Unit

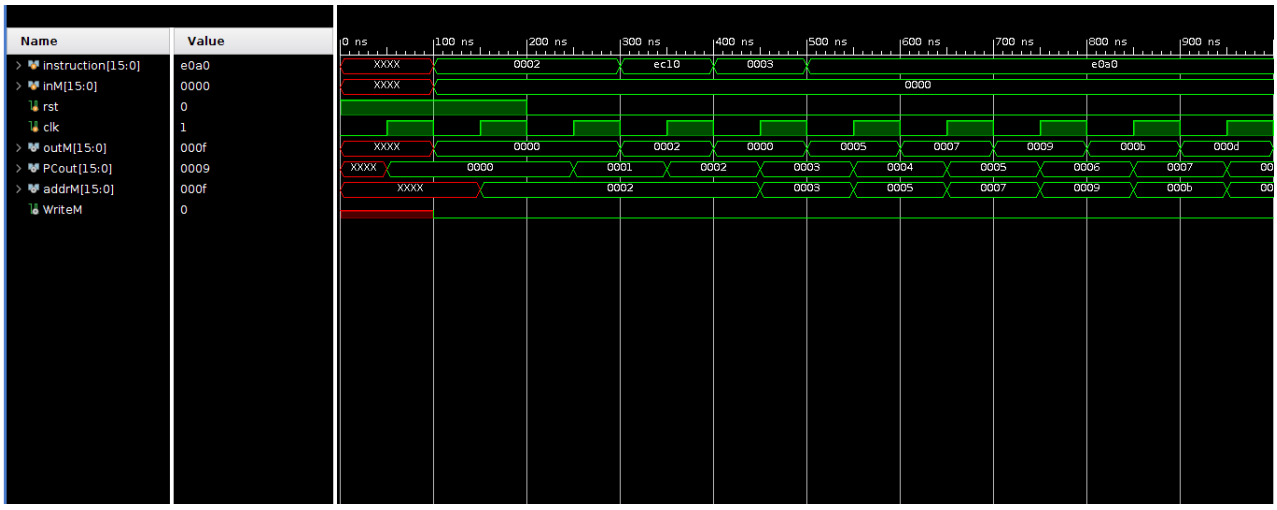

Fig5.29: Result of Central Processing Unit

- Simulation of program - maximum of two numbers

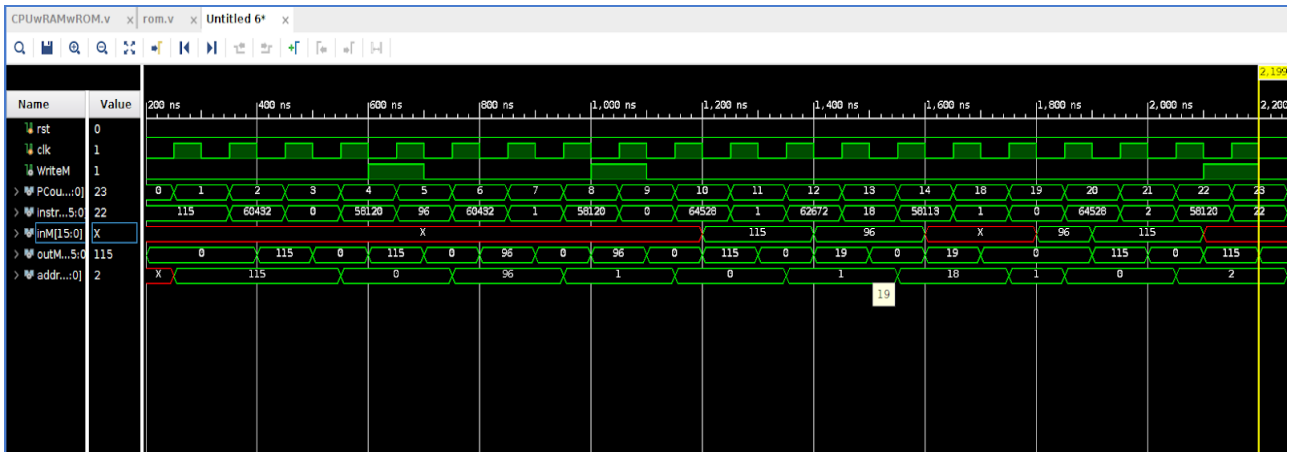

Fig.5.30: Results of the program - Maximum of two numbers

The above simulation shows the working of the CPU when the "maximum of two numbers" program is loaded. As an overview two numbers are compared and the bigger of the two numbers is stored in a particular RAM location.

CHAPTER 6

APPLICATIONS, ADVANTAGES AND LIMITATIONS

Advantages:

- Since the processor is completely modular, each and every component in the layers of abstraction can be customised according to our requirements. This enables us to improve our solutions and it leads to efficiency in development through reusing this code. We can also improve the efficiency of the code using this approach.
- Since the processor is modular it can be easily understood and can be used to teach computer architecture and FPGA implementation to the masses.
- Since the processor is simpler it is much more reliable and data validation is also easier in a 16-bit CPU
- In cases where much computing power is not needed then a 16-bit CPU is a better choice as the price of a 16-bit CPU is far cheaper than that of a 64-bit CPU. for example, a 16 bit cpu costs around 3 dollars whereas a 64 bit cpu costs around 300 dollars
- Since each bit Flip involves moving charge and moving charge involves moving current, a 16-bit CPU is much more power efficient than compared to a 32-bit CPU.
- 16 bit processors are much easier to integrate into a device. Their hardware requirements are much more relaxed. Everything is much cheaper, the interfaces, power supply, all additional circuitry
- The software development for 16-bit CPU is much easier than compared to CPU's with higher bits

Applications:

- This can be used as a model to teach and experiment in schools and colleges for the students to learn computer architecture.
- Also since the processor is modular and is highly customisable, it can be easily modified to meet the custom requirements as and when required.
- This processor can be used in places where there are low processing power requirements to mitigate the high cost of processors when an FPGA is readily available.
- Can be used in places where reliability is an important factor rather than performance as 16-bit processors are more reliable.

Limitations:

Some limitations of the CPU are as follows

- The CPU does not have extra subsystems because of which we cannot execute complex instructions. The computation is limited to the execution of simple instructions.
- The CPU does not support overflow conditions hence we cannot perform operations whether overflow condition is required.
- The CPU fetches data in one clock cycle and the execution of the instruction happens in the next clock cycle. This slows down the execution of the program.
- When it comes to high performance computation the 16-bit processor fails to deliver as it can only process 16 bits at a time which is a huge drawback.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

One might assume that the 16-bit processor is obsolete, but that is not the case in the present world. The 16-bit processor plays a key role in many important applications for example satellites still use 16-bit processors as they do not need much computation power, power consumption is an important factor here and 16 bit processors use significantly less power than a 32-bit processor. 16-bit processors are also very reliable which make them suitable for such applications.

This project has explained the methodology of designing a 16-bit processor from scratch. The bottom up approach has been used in the designing of the processor. First the basic gates were defined using the CMOS logic. Then the other components like adders, multiplexers etc were created using the designed basic gates. These were then integrated to form the ALU, program counter and the decoder, all these components were then integrated to make the processor. The advantage of using the bottom-up approach in the designing of this processor was that the independent modules could first be created and then they could be integrated together to create the CPU, this made it possible to modify the modules easily without much modification to the CPU as a whole.

This project also explains the translation of the Hack language to machine code.The program that is in the Hack language is first converted into the machine code. The machine code is then stored in the memory which the processor can then access during the execution of the program. The execution happens in a line by line manner until a jump instruction is encountered. The software implementation of the processor has been explained along with the simulation results of all the components that are present in the CPU. The simulation results have been tested and verified manually to ensure the proper working of the components.

For future work, the modules present in the CPU can be removed and new modules can be added to meet the requirements of the user and to create customized features without affecting the functionality of the CPU. The swapping of the modules can also result in performance benefits for the CPU. Since some bits are not used in the instruction set, new functionality can be mapped and added to the instruction set to further increase the functionality of the CPU. Also new software's can be developed and implemented according to the architecture and the functioning of the CPU. This will allow the CPU to be used in various new applications and opens up a whole new world of possibilities.

REFERENCES

[1] Samir Palnitkar, *Verilog HDL- A Guide to Digital Design and Synthesis,* 2nd ed.,1996.

[2] Noam Nisan and Shimon Schocken, The *Elements of Computing System: Building a Modern Computer from First Principles,* 2005.

[3] Iftach Amit, *Hack Project (Formerly: The I Computer)*, The Interdisciplinary Center, Herzlya

[4] Bertrand Russell, Computer *Architecture: A Software Perspective.*

[5] Kai Hwang, *Advanced Computer Architecture*,2001.

[6] Frank Vahid and Tony Givragis, *Embedded System Design: A Unified Hardware/Software Approach*,1999.

[7] Raj Kamal, *Embedded System: Architecture, Programming and Design*, 2nd ed.

[8] David A. Patterson, David R. Ditzel, *The Case for the Reduced Instruction Set Computer.*

[9] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, Arvind, *Composable Building Blocks to Open up Processor Design.*

[10] Shraddha M Bhagat, Sheetal U Bhandari. *Design and Analysis of 16-bit RISC Processor*.

[11] Supraj Gaonkar, Anitha M, *Design of 16-bit RISC Processor*.

[12] Abdullah Yıldız, H. Fatih Ugurdag, Barış Aktemur, Deniz İskender, Sezer Gören. *CPU design simplified*.

[13] DESIGN AND APPLICATION OF RISC PROCESSOR, Mohammad Zaid, Prof. Pervez Mustajab,Department of Electronics Engineering Zakir Hussain College of Engineering & Technology Aligarh Muslim University, Aligarh. U.P. India,May 2018

[14] Single Cycle RISC-V Micro Architecture Processor and its FPGA Prototype,Don Kurian Dennis, Ayushi Priyam, Sukhpreet Singh Virk, Sajal Agrawal, Tanuj Sharma, Arijit Mondal and Kailash Chandra Ray Indian Institute of Technology Patna,March 2018

[15] Embedded multi-core systems for mixed-critical applications with RPMsg protocol based on xilinx ZYNQ-7000,Mustafa O. E. Aboelhassan,Ondrej Bartik,Marek Novak,February 2018

[16] The history and use of pipelining computer architecture: MIPS pipelining implementation,Iro Pantazi – Mytarelli New York Institute of Technology Old Westbury, New York, 11568,August 2013

[17] The Evolution of RISC Technology at IBM,John Cocke and V Markstein,January 1990

[18] A 32-bit RISC-V AXI4-lite bus-based Microcontroller with 10-bit SAR ADC,Ckristian Duran, Luis Rueda D., Giovanny Castillo, Anderson Agudelo, Camilo Rojas, Luis Chaparro, Harry Hurtado, Juan Romero, Wilmer Ramirez, Hector Gomez, Javier Ardila, Luis Rueda, Hugo Hernandez, Jose Amaya and Elkim Roa Design Group of Integrated Systems CIDIC, Universidad Industrial de Santander, Bucaramanga, Colombia,April 2016

CERTIFICATES

IEEE

VAIGAI
VAIGAI
COLLEGE OF ENGINEERING

CERTIFICATE OF PRESENTATION

This certificate is awarded to

Aldrich Shawn Lewis

has successfully presented a paper entitled

Implementation of 16-Bit Hack CPU on FPGA

in the **International Conference on Intelligent Computing and Control Systems (ICICCS 2020)**
organised by Vaigai College of Engineering during 13-15, May 2020 at Madurai, India.

SESSION CHAIR

ORGANIZING SECRETARY
Prof. R. Rajesh Kumar

CONFERENCE CHAIR
Prof. P. Sugumaran

IEEE XPLORE ISBN: 978-1-7281-4876-2

IEEE

VAIGAI
VAIGAI
COLLEGE OF ENGINEERING

CERTIFICATE OF PRESENTATION

This certificate is awarded to

Paarthvi Sharma

has successfully presented a paper entitled

Implementation of 16-Bit Hack CPU on FPGA

in the **International Conference on Intelligent Computing and Control Systems (ICICCS 2020)**
organised by Vaigai College of Engineering during 13-15, May 2020 at Madurai, India.

SESSION CHAIR

ORGANIZING SECRETARY
Prof. R. Rajesh Kumar

CONFERENCE CHAIR
Prof. P. Sugumaran

IEEE XPLORE ISBN: 978-1-7281-4876-2

IEEE

VAIGAI
VAIGAI
COLLEGE OF ENGINEERING

CERTIFICATE OF PRESENTATION

This certificate is awarded to

Prajwal Shenoy KP

has successfully presented a paper entitled

Implementation of 16-Bit Hack CPU on FPGA

in the **International Conference on Intelligent Computing and Control Systems (ICICCS 2020)**
organised by Vaigai College of Engineering during 13-15, May 2020 at Madurai, India.

SESSION CHAIR | **ORGANIZING SECRETARY** Prof. R. Rajesh Kumar | **CONFERENCE CHAIR** Prof. P. Sugumaran

IEEE XPLORE ISBN: 978-1-7281-4876-2

IEEE

VAIGAI
VAIGAI
COLLEGE OF ENGINEERING

CERTIFICATE OF PRESENTATION

This certificate is awarded to

Sanath S. Naik

has successfully presented a paper entitled

Implementation of 16-Bit Hack CPU on FPGA

in the **International Conference on Intelligent Computing and Control Systems (ICICCS 2020)**
organised by Vaigai College of Engineering during 13-15, May 2020 at Madurai, India.

SESSION CHAIR | **ORGANIZING SECRETARY** Prof. R. Rajesh Kumar | **CONFERENCE CHAIR** Prof. P. Sugumaran

IEEE XPLORE ISBN: 978-1-7281-4876-2

Implementation of 16-Bit Hack CPU on FPGA

Druva Kumar S, Paarthvi Sharma, Prajwal Shenoy KP, Sanath S. Naik, Aldrich Shawn Lewis

Electronics and Communication, Dayananda Sagar College of Engineering – Bangalore 560078

druva-ece@dayanandasagar.edu, paarthvi.sharma1998@gmail.com

prajwalkpshenoy@gmail.com, sanath11598@gmail.com, aldrichshawn15@gmail.com

***Abstract*—This paper presents the design and implementation of a 16-Bit hack CPU which is a modular processor. The paper is intended to showcase the process involved in building a complex circuit capable of performing real world computations, from the most basic component used for digital data representation that is the CMOS. The design methodology used is a bottom up approach, this starts with the construction of basic gates and moving up to major components like the program counter, ALU, etc. and ends with the complete construction of the CPU using the previously built components in a modular manner. The aim of this paper is to give the reader a complete understanding of the functioning of a simple computer in a digital electronics abstract. This paper will also give an idea about the data flow in a CPU triggered by a CPU. An idea regarding the way a low level programming language controls this data flow can also be understood. This CPU design is easily implementable on an FPGA and is hence a great tool to teach students about the basics of Computer Architecture and Digital System Design. The CPU reads instructions from the ROM and performs operations using the A register, D register, or the RAM memory units based on the instruction type. There are mainly 2 types of instructions, A instructions and C instructions. The A instructions have the sole purpose of storing values into the A register while the C instruction can perform multiple operations.**

***Keywords*— Arithmetic Logical Unit(ALU), Program Counter(PC), Central Processing Unit(CPU).**

I. Introduction

Processors are small silicon chips which execute instructions to complete tasks. These instructions are written in the storage device connected to the processor. These instructions are executed one at a time in a logical manner by the processor. These instructions can be of varied sizes. The unit of an instruction is a bit, a digital value which can represent 2 states 1 (logic HIGH and 0 logic LOW). Some of the most common instruction sizes are 8bit, 16bit, 32bit and 64bits. An instruction of size 16bits can represent a total of 2^{16} unique instructions. These instructions decide which type of operation the processor should execute. The instructions are designed such that each bit has a specific function and controls a specific part of the processor. The 16bit processor is designed in a modular manner such that any individual component could be swapped to a different component to observe performance changes without the need to redesign the whole processor again. The processor has also been designed in a bottom up approach (starting by creation of the logic gates using MOSFETS) so that any changes made in any level reflect upwards without any redesigning. The method of abstraction has also been implemented here such that an architect does not have to worry about the underlying core components present.

II. Related Work

The machine language is adopted from the book The Elements of Computing system by Noam Nisan and Shimon Schocken [2] which provides us with the knowledge of designing a processor from scratch. The book uses custom HDL (hardware description language) to design the components and the integration is done using those components. The assembler is also created by referring to the same book [2]. For further understanding of how the different components of the processor works, we have referred to a few computer architectures and embedded system designing books [4], [5], [6], [7]. We learnt how to design the custom instruction set and how to reduce the number of components required to execute these instructions from the past work of some authors. [8], [9], [10], [11]. The components are built using standard design in Verilog. Verilog coding methodology and design flow were referred from the book Verilog HDL-A guide to digital design and synthesis by Samir Palnitkar [1]. Similar kind of work was done by an author [3] but the drawback of that processor is that it's not modular. In another reference paper, a simple CPU is programmed using C language and assembly language which aims to teach the designing basics of a simple and customisable CPU [12]. Similar approach has been considered while designing our processor.

III. Translation

The program to be translated is stored is an .asm file. The assembler translates the .asm file into the binary equivalent. These binary instructions are written onto the ROM. Each command is translated individually.

Each field of the assembly command is translated into machine code in accordance to the tables given above. To translate the assembly code to machine code the assembler first checks if the instruction is an A instruction or C instruction.

			Computation							Destination			Jump		
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d2	j1	j2	j3

Table.1. C-Instruction Format

Comp when a = 0	c1	c2	c3	c4	c5	c6	Comp when a = 1
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

Table.2. Computation

Jump	j1	j2	j3
null	0	0	0
JGT	0	0	1
JEQ	0	1	0
JGE	0	1	1
JLT	1	0	0
JNE	1	0	1
JLE	1	1	0
JMP	1	1	1

Table.3. Jump Table

Destination	d1	d2	d3
Null	0	0	0
M	0	0	1
D	0	1	0
MD	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
AND	1	1	1

Table.4. Destination Table

The A instruction will always start with a zero. It is used to set the A-register to a desired 15- bit value. The A instruction is the only way that we can enter data into the computer while the program is running. Also we can jump to certain memory locations to manipulate the data present at that location. Also it can be used to control the jump instruction by loading the jump address to the A register. When we use @constant we load an unsigned value to the A register. This can then be used to perform computation or any other operation that requires the constant.

If it is a C instruction, then the assembler checks the A-bit. If the A bit is 1, the ALU performs operations between the Memory and the D register. If the A bit is 0, the ALU performs operations between the A and D registers. Based on the instruction the binary value for the bits c1-c6 are assigned. The next three bits are the destination bits (d1, d2, d3) which are used to indicate where the data must be stored (RAM, A reg, D reg). The next 3 bits are the jump bits (j1, j2, j3). Jump bits are used only when there is a jump instruction. There are 7 jump instructions (6 compare and jump, 1 simple jump).

IV. Architecture

The computer architecture is designed keeping in mind the hack language. The entire computer architecture consists of three major blocks, the RAM, ROM and CPU. The CPU deals with 16 bit operands at every positive edge instance of a clock. The programs written in the hack language are converted into binary values where sets of 16 bit binary numbers represent an instruction. These instructions are stored in the ROM. The RAM is initially empty and is capable of storing the CPU's output. The RAM also gives an input to the CPU depending on the instruction given to the CPU by the ROM. The CPU's program counter is responsible for selecting which instruction is to be executed in the next machine cycle of the CPU. The RAM has a write

enable pin that decides if the CPU's output is to be stored in the RAM or not.

The CPU consists of a decoder, two general purpose registers A and D, an ALU and a Program Counter. All CPU components are designed using multiplexer logic. There are multiplexers present in between the fundamental components of the CPU which are controlled by the decoder.

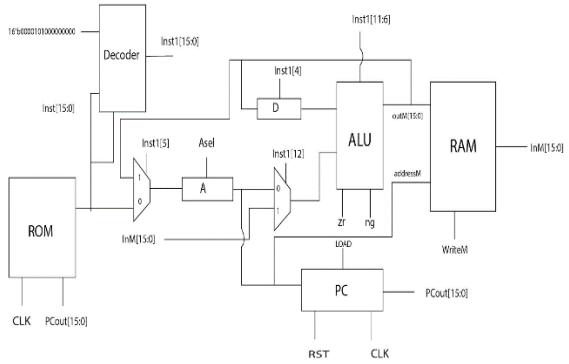

Fig.1.Processor Diagram

The decoder uses combinational logic coupled with multiplexer logic to decipher the instruction input from the ROM. The decoder distinguishes the input instruction into the A type or the C type. The decoder's output is given to the select lines of the multiplexers and sub components of the CPU. The 'c' bits of the instruction go to the ALU, the 'd' instruction bits go to the 'write enable' pins of the general purpose registers and the RAM. The 'j' bits are given to the Program Counter input. 'Asel' is used to write into the A register. The 'Load' line is given to the Program Counter. Load is made high depending on the output of the ALU. The state of the two flag bits of the ALU, 'zr' and 'ng' determine if the Load line is to be made high or not.

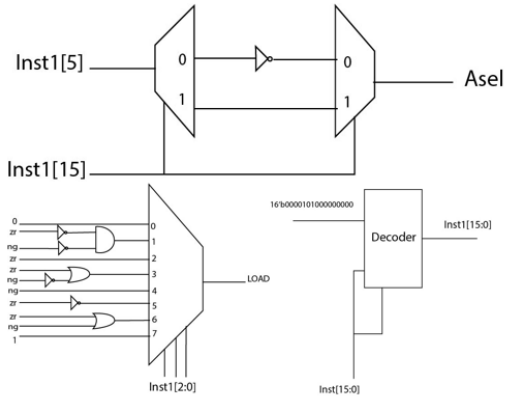

Fig.2. Decoder

The A register is unique and determines the data that goes in as the input to the ALU, the value of Program Counter that points to the next instruction to be executed and the address location to which the CPU output gets loaded to the RAM. The D register is connected to the other input of the ALU.

The Program Counter is used to determine which instruction of the ROM is executed next by the CPU. When the Program Counter is reset, it gets initially loaded with 0s. If the 'j' bits are 0, the PC increments by one. If the 'j' bits are non-zero, the value stored in A gets loaded into the PC. The figure below gives a visual description of the Program Counter.

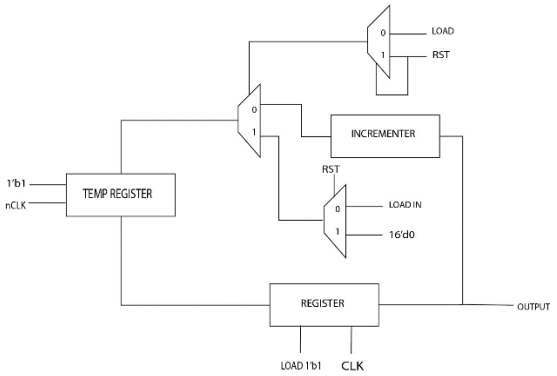

Fig.3.Program Counter

The ALU has two 16 bit input busses and 6 control bits which are responsible for determining the kind of computation to be done on the two 16-bit bus inputs. The function of the control bits is given below. The control pins 'zx', 'nx', 'zy', 'ny', 'f', 'no' determine what computation is to be done with the two 16 bit inputs. 'zx' makes the 'x' input 0, 'nx' complements the input. 'Zy' and 'ny' perform similar actions on the 'y' input. 'f' and 'no' are responsible for 'zr' and 'ng' are output flag bits that determine if the output is zero and if it is negative respectively.

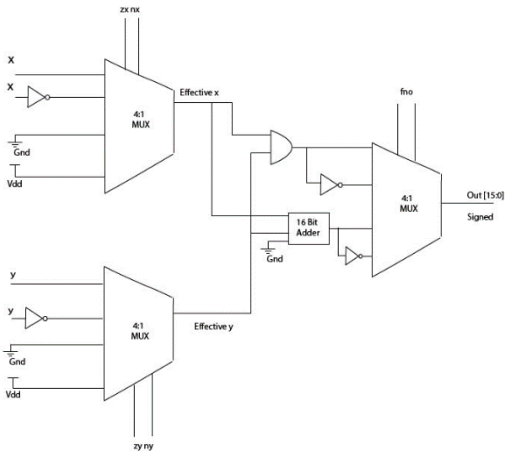

Fig.4. Arithmetic Logical Unit

V. Implementation and Results

In this paper we describe the design of the processor by laying out and explaining how it is built using different low level chips. The basic gates required are initially designed using CMOS logic and using those designs, the higher abstractions have been made All the components are designed and are simulated using

Xilinx ISE. The Verilog codes are compatible with FPGA (Spartan -6).

@2	0000000000000010
D=A	1110110000010000
@7	0000000000000111
D=D+A	1110000010010000

Fig.5. Addition of two numbers

1.Arithmetic Logical Unit

Basic components such as adders and logical gates are used in order to create the ALU. The ALU is capable of handling different arithmetic and Boolean operations such as addition, subtraction multiplication, AND, OR, NOR, NAND, XOR.

Fig.5. Result of Arithmetic Logical Unit

2. Program Counter

Out of the several registers present in the processor, PC is the register present in the processor which holds the address location of the instruction being executed and gets incremented to the next address location as the next instruction gets fetched for execution.

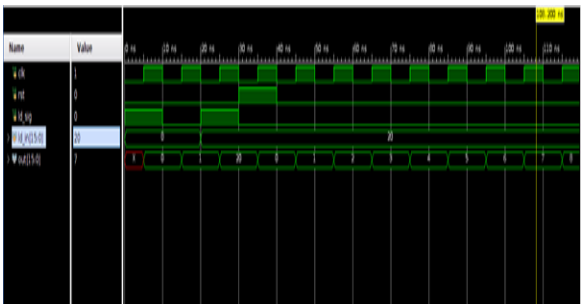

Fig.6. Result of Program Counter

3.Decoder

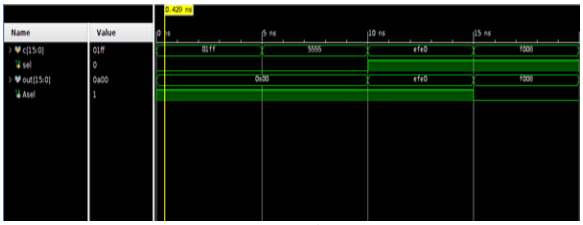

Fig.7. Result of Decoder

4.Central Processing Unit

Fig.8. Result of Central Processing Unit

VI. Conclusion

This paper has explained the methodology of designing the 16bit processor, explaining the bottom up approach. This paper has also explained the translation of the Hack language to machine code which the processor can follow the instruction stored in storage. The implementation of the processor has also been explained along with the simulation results of the major components. From these simulations and the example program mentioned in the previous sections, this paper explains the implementation of a 16bit processor. In addition to this, creation of additional modules which can be swappable with the present existing modules such that ready customisation can be easy. Examples to this would be creation of different addition modules and swapping between them to check the performance factors of the processor. Future work for this project would be the creation of multiple swappable modules to swap out existing modules with the customised features for performance benefits, space constraints, etc.

References

[1] Samir Palnitkar, *Verilog HDL- A Guide to Digital Design and Synthesis*, 2nd ed.,1996.

[2] Noam Nisan and Shimon Schocken, The *Elements of Computing System: Building a Modern Computer from First Principles,* 2005.

[3] Iftach Amit, *Hack Project (Formerly: The I Computer)*, The Interdisciplinary Center, Herzlya

[4] Bertrand Russell, Computer *Architecture: A Software Perspective.*

[5] Kai Hwang, *Advanced Computer Architecture*,2001.

[6] Frank Vahid and Tony Givragis, *Embedded System Design: A Unified Hardware/Software Approach*,1999.

[7] Raj Kamal, *Embedded System: Architecture, Programming and Design*, 2nd ed.

[8] David A. Patterson, David R. Ditzel, *The Case for the Reduced Instruction Set Computer*.

[9] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, Arvind, *Composable Building Blocks to Open up Processor Design.*

[10] Shraddha M Bhagat, Sheetal U Bhandari. *Design and Analysis of 16-bit RISC Processor*.

[11] Supraj Gaonkar, Anitha M, *Design of 16-bit RISC Processor*.

[12] Abdullah Yıldız, H. Fatih Ugurdag, Barış Aktemur, Deniz İskender, Sezer Gören. *CPU design simplified*.