"HARDWARE IMPLEMENTATION OF A PROCESSOR"

Project Report Submitted by

Kishan Muchinnaya K.	**4NM12EC068**
Nihal P. Shetty	**4NM12EC087**
Nikhith M. K.	**4NM12EC088**
Prajwal Sharma K.	**4NM12EC101**

UNDER THE GUIDANCE OF

Mr. Mahaveera K.
Assistant Professor

in partial fulfillment of the requirement of the award of the Degree of

Bachelor of Engineering
in
Electronics & Communication Engineering

from
Visvesvaraya Technological University, Belagavi

Department of Electronics & Communication Engineering
N. M. A. M. Institute of Technology
(An Autonomous Institution Affiliated to VTU, Belagavi)
(AICTE approved, ISO 9001:2008 Certified)
Nitte-574110, Udupi District, Karnataka

April-2016

N. M. A. M. Institute of Technology

(An Autonomous Institution Affiliated to VTU, Belagavi)
(AICTE approved, ISO 9001:2008 Certified)
Nitte-574110, Udupi District, Karnataka
Department of Electronics & Communication Engineering

Certificate

Certified that the project work entitled ***"HARDWARE IMPLEMENTATION OF A PROCESSOR"*** *is a bonafide work carried out by* ***Kishan Muchinnaya K. (4NM12EC068), Nihal P. Shetty (4NM12EC087), Nikhith M. K. (4NM12EC088), Prajwal Sharma K. (4NM12EC101)*** *in partial fulfilment of the requirements for the award of* ***Bachelor of Engineering Degree in Electronics & Communication Engineering*** *prescribed by Visvesvaraya Technological University, Belagavi during the year 2015-16.*

It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library.

The project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of Engineering Degree.

Signature of the Guide **Signature of the HOD**

Signature of the Principal

Semester End Viva Voce Examination

Name of the Examiners	**Signature with Date**
1. _____	_____
2. _____	_____

Abstract

Processor design is one of the important step in any hardware design. Any processor is made up of millions of transistors and design is complex. The processor design is critical because it has to be faster, consume less area and power. This project is an attempt to show that the complex processor design can be simplified and designed using the simple NAND gates.

The project involves design of a 16-bit processor. The proposed processor design is based on Harvard Architecture i.e. It has separate Data memory and Instruction memory. The prominent part of the processor is Arithmetic Logic Unit (ALU) and it can perform eighteen operations.

The design is modeled using Verilog HDL. The processor designed is a non-pipelining processor. It uses sequential and combinational circuits as its building blocks.The Instruction Set Architecture (ISA) is divided into two instructions. Firstly, Data Instruction which when decoded controls the ALU operation and data movements. Second instruction is Address Instruction which specifies the address where data has to be stored.

Layouts for some of the blocks are built and output is verified. Delay and area has been optimized and netlist is extracted from symbolic layout. Parasitic capacitance has been extracted and designed rule violation has been checked using design rule checker.

Acknowledgment

We take this opportunity to convey our sincere gratitude to all those who have been kind enough to offer their advice and provide assistance when needed which has lead to the successful completion of this project.

We express my most heartfelt gratitude to **Dr. Niranjan N. Chiplunkar**, Principal, N.M.A.M.I.T., Nitte.

We would like to thank **Dr. K. Rajesh Shetty**, Professor and Head of the Department, Department of Electronics and Communication, N.M.A.M.I.T., Nitte, for permitting us to take up this project and encouraging us to complete it successfully.

We extend our sincere thanks to our guide **Mr. Mahaveera K**, Asst. Professor, Department of Electronics and Communication, N.M.A.M.I.T., Nitte for his kind co operation, valuable guidance and support during the project.

We would like to thank **Mr. Rajeevan K. V.**, System Analyst, N.M.A.M.I.T., Nitte, for his kind support and assistance during the project.

We stand strong today by the knowledge gained from our Professors that we respect and shall oblige to forever.

Lastly we are grateful to our friends, fellow classmates for their inexhaustible support and faith.

Kishan Muchinnaya K.

Nihal P. Shetty

Nikhith M. K.

Prajwal Sharma K.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

All the computation and storage in a computer is done using a processor of required specification. The processor consists of Central Processing Unit (CPU), Random Access Memory (RAM) and Read Only Memory (ROM). The CPU performs arithmetic and logical operations and hence called the brain of the computer. The CPU also gives the address as output to fetch the data from the memory.

To design the processor the Instruction Set is divided into two sets: one is Address Instruction and other is Data Instruction. Using the Data Instruction the operations to be performed by the Arithmetic Logic Unit (ALU) can be specified. The proposed processor is built using NAND gates.

Architecture of the processor was designed by referring to Harvard Architecture. The prominent part of the processor is Arithmetic Logic Unit (ALU), it can perform eighteen operations. All the computation and storage in a computer is done using a processor of required specification. The processor has an addressable 24 K, 16-bit Data Memory which is also called as RAM. It also has an addressable 16-bit Instruction Memory which is also called ROM.

Layout for some of the blocks are built using Alliance VLSI CAD tool. This tool has a command line interface. Using various commands and options available in this tool we have optimized delay and area. Parasitic capacitance and resistance are also extracted and then the layout is checked for violation of design rules using Design Rule Checker.

1.1 Motivation

Processor design is one of the important step in any hardware design. Any processor is made up of millions of transistors and design is complex. NAND gate is a universal gate

which can be used in all digital designs. Combinational and Sequential blocks can be built using NAND gates. The Combinational Logic Circuits and Sequential Logic Circuits form the basic building of Arithmetic Logic Unit and Memory Unit respectively. It can be inferred that the processor can be built entirely using NAND gates. This project is an attempt to show that the complex processor design can be simplified and designed using the simple NAND gates.

Chapter 2

Literature Survey

2.1 CISC Architecture

Complex Instruction Set Computing (CISC) Architecture is a type of processor design and a high level code which executes single instructions of variable length consisting of several low level operations. The several operations performed are load from memory, arithmetic operations and memory storage. These instruction sets are capable of multi-step operations and each instruction utilizes more number of clock[3].

2.2 RISC Architecture

Reduced Instruction Set Computer (RISC) Architecture uses simple instructions to perform low level operations which can execute in single clock cycle. These instructions are divided into multiple instructions. RISC Architecture emphasises on software and requires less hardware space. Pipelining is made possible in this architecture[3].

2.3 Combinational Logic

Combinational logic is a type of digital logic where the output function depends on the present input only. This implies that combinational logic does not contain memory. It is used to perform boolean logic operations on input signals. The Arithmetic Logic Unit (ALU) is built using Combinational Logic circuits like Half Adder, Full Adder, Multiplexer, De-Multiplexer.

Combinational Logic is done using:

1) Sum of Products (SOP): SOP expressions consists of two or more AND terms that are

ORed together. Each AND term consists of one or more variables individually appearing in either complemented or uncomplemented form.
2) Product of Sum (POS): POS consists of two or more OR terms that are ANDed together. Each OR term contains one or more variables in complemented or uncomplemented form[4].

2.4 Sequential Logic

Sequential Logic Circuits are function of not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A synchronous sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an asynchronous sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change[4].

2.5 Harvard Architecture

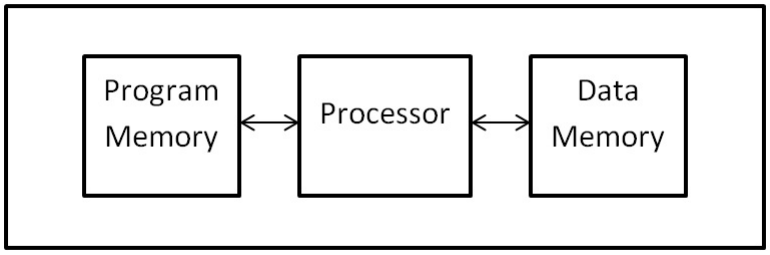

Figure 2.1: Block Diagram of Harvard Architecture

Harvard Architecture consists of two separate memory, one for storing data and the other is known as program memory as shown in Figure 2.1. With appropriate pipelining, processor can complete an instruction in one cycle. The first stage of pipeline is to read the instruction from program memory that has to be executed, the second stage involves reading data from data memory by decoding instruction or address. In this architecture the stages of pipeline varies from system to system[1].

2.6 Von Neumann Architecture

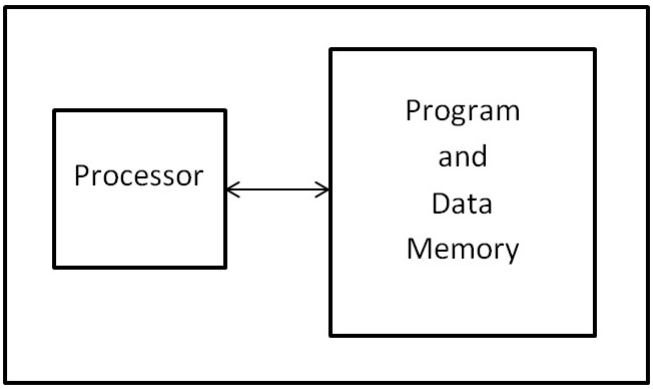

Figure 2.2: Block Diagram of Von Neumann Architecture

Von Neumann Architecture consists of single storage memory for both stored data and for the program to be executed as shown in Figure 2.2. Using this Architecture pipelining of instruction is not possible and it requires two clock cycles to complete an instruction. In the first clock cycle, the processor gets the instruction from the memory and decodes it. In the next clock cycle the data is taken from the memory[1].

2.7 NAND versus NOR

Non-volatile memory can be divided into two categories: NAND and NOR type memory. The most distinguishing feature between NAND and NOR type memory is the initial access time. The initial access time of NAND is 25 μs and that of NOR depends on its reading mode, is approximately 50 - 100 ns. This is approximately a 250 to 500 fold difference between the two. After the initial access, both NAND and NOR flash take a similar amount of time for sequential data reading. NOR flash memory is mainly used to store instruction codes for operation, while NAND is used for data storage. However, NAND does have more economical benefits[5].

2.8 Read Only Memory

The Read Only Memory (ROM) is a type of semiconductor memory designed to hold data that are either permanent or will not change frequently. During normal operation, no new data can be written into a ROM, but can be read from ROM. The process of entering data is called programming or burning the ROM. Some ROMs cannot have their data changed

once they have been programmed, others can be erased or reprogrammed. Since ROMs are nonvolatile, these programs are not lost when the electrical power is turned off. When the microcomputer is turned on, it can immediately begin executing the program stored in ROM. The different types of ROM are Mask Programmed ROM, Programmable ROMs, Erasable Programmable ROMs and Electrically Erasable PROMs[4].

2.9 Random Access Memory

Random Access Memory (RAM) is of two types: Static RAM (SRAM) and Dynamic RAM (DRAM).
1) SRAM consists of internal latches that store binary information. The stored information remains valid until power is applied to the memory.
2) DRAM stores the binary information in terms of electric charge in a capacitor. The stored charge in a capacitor discharges with time and hence DRAM has to be periodically refreshed. DRAM is made using one MOSFET (Metal Oxide Semiconductor Field Effect Transistor) and a capacitor[4].

2.10 Alliance CAD

The set of tools provided by Alliance lets us design and test a circuit from its specification to its layout form and many of its intermediate formats. Alliance provides a symbolic cell library that makes the design of circuits independent of the technology used in their fabrication step. The libraries include a standard cell library and several specific purpose cells for memory and data path logic. Many of the tools in Alliance can be used independently as command line tools. Others have a graphic user interface that requires Motif and the X11 library used in many variants of UNIX and Linux. If we provide an adequate technology file the design obtained with Alliance can be converted to CIF or GDSII format for silicon fabrication. In the following sections we introduce briefly each tool with a very simple example to illustrate their use and options[6].

2.11 Lambda and Micron Rule Set

There are five new open source standard cell libraries, the vsclib, wsclib, vxlib, vgalib and rgalib. They have been drawn with the GRAAL software from Alliance, part of an exten-

sive open source software suite for designing integrated circuits with a standard cell design methodology.

The libraries have been characterized in a generic 0.13 μm technology. The spice model comes from the University of California, Berkeley. The layout has been drawn using MOSIS layer numbers and names and the pharosc rule set, and then scaled to what are slightly oversized 0.13 μm rules which should be compatible with most foundries.

The vxlib is compatible with the sxlib created by the Alliance software authors. The vsclib is a completely new library design. The Alliance sxlib has also been characterized in 0.13 μm using the same methodology and converted to the same 0.13 μm layout rules[7].

2.12 Layout

Layout is usually drawn in the micron rules of the target technology. When a new technology becomes available, the layout of any circuits which can be migrated needs to be adapted to the new design rule set. This can be a problem if the original layout has aggressively used all the minimum widths and spacing which are then incompatible with the rules of the new technology.

A solution made famous by Mead and Conway is to draw the layout in a nominal 2 μm layout and then apply a lambda scaling factor to the desired technology. This actually involves two steps:

1) A lambda scaling factor based on the pitch of various elements like transistors, metal, poly etc.

2) Under or over-sizing individual layers to meet specific design rules.

Thus, for the generic 0.13 μm layout rules, a lambda scaling factor of 0.055 is applied which scales the poly from 2 μm to 0.11 μm. Then the poly is oversized by 0.005 μm per side to bring its width up to 0.12 μm. If the foundry requires drawn poly geometries of 0.13 μm, then the oversize is set to 0.01 μm per side.

The layout rules includes a generic 0.13 μm set. Generic means that layout drawn with these rules could be ported to a 0.13 μm foundry with no scaling, but some individual layers (especially contact, via, implant and poly) might need to be over or undersized[7].

Chapter 3

Processor Architecture

3.1 Processor Block

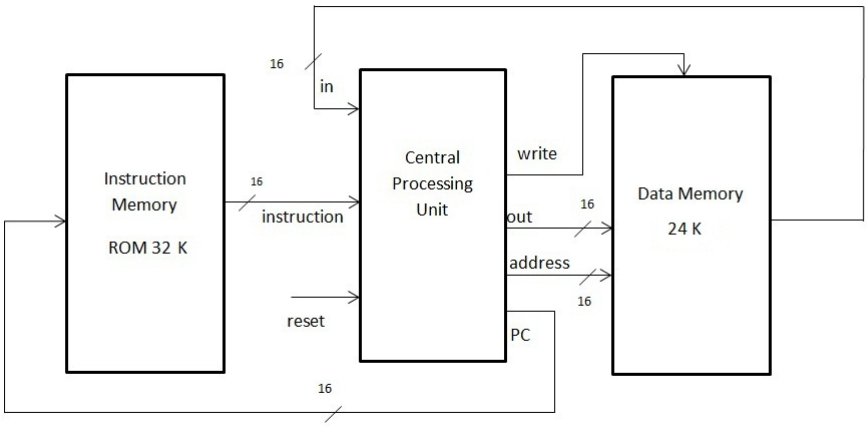

Figure 3.1: Block Diagram of Proposed Processor

3.1.1 Instruction Memory

Instruction Memory is implemented using direct access Read Only Memory (ROM). This ROM consists of 32 K addressable 16-bit Registers. The 32 K ROM can be accessed using the Program Counter (PC) which is one of the outputs of Central Processing Unit (CPU). The instruction is fetched from the Instruction Memory according to the PC specified by the CPU.

3.1.2 Data Memory

Data Memory chip has the interface of a typical Random Access Memory (RAM). To read the contents of this memory, 'address' (one of the output of CPU) which is the address of the register from which data has to be read is given as a input. The contents read from the memory is 'in' which is also one of the inputs to the CPU as shown in Figure 3.1. To 'write' the data into the Data Memory 'write' signal has to be activated and the address in which the data has to be stored should be given as the input. RAM consists of 24 K addressable 16-bit Registers.

3.1.3 Central Processing Unit

Central Processing Unit (CPU) consists of Arithmetic Logic Unit (ALU) which performs various Arithmetic and Logical Operations. The operation to be performed by the CPU is selected using the decoded instruction. The CPU output 'out' can be written into RAM or can be used as a input data for the next instruction. The instruction fetched from the Instruction Memory is decoded using a decoder. The instruction to be fetched is the output of Program Counter.

3.2 Central Processing Unit Block

In the Central Processing Unit the instruction (Address Instruction or Data Instruction) is decoded using a decoder. The contents of A-Register depends on the output of the Multiplexer which can be either the data from previous output of the ALU or an instruction depending on whether the instruction is Data Instruction or Address Instruction. The D-Register is used to provide previous output of ALU as one of the input for computation.

3.2.1 Arithmetic Logic Unit

ALU performs arithmetic and logical operations and the proposed ALU is designed to perform eighteen operations. The operations performed by the ALU depends on the control bits (C1-C6) obtained by decoding the Data Instruction.

3.2.2 Program Counter

Program Counter is used to fetch instruction from Instruction Memory. When ‘reset’ is set to ‘1’ the output of PC is ‘0000000000000000’. The control bits are used to increment the PC and are also used to load the PC with new value of address from which instruction has to be fetched.

3.3 Controller and Data Path

The controller block controls the entire working of the processor. The input ‘reset’ to the controller is to set the PC value to ‘0000000000000000’, ‘PC’ is used to fetch the instruction from ROM. The outputs of controller C1 to C6 is used as control bits for ALU operation as shown in Table 3.1. D1, D2, D3 are used to store the data in memory register which operation is shown in Table 3.2. J1, J2, J3 are used as control bits for Jump instruction as shown in Table 3.3. ‘write’ bit is used to write the data into Data Memory as shown in Figure 3.2. Data Path consists of ALU and Memory. The inputs for Data Path are C1-C6, J1-J3, D1-D3 and ‘write’. The output of the Data Path are ‘out’ and ‘PC’ as shown in Figure 3.3.

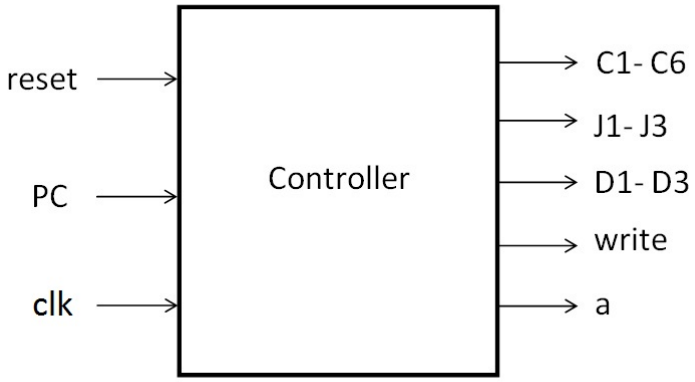

Figure 3.2: Block Diagram of Controller

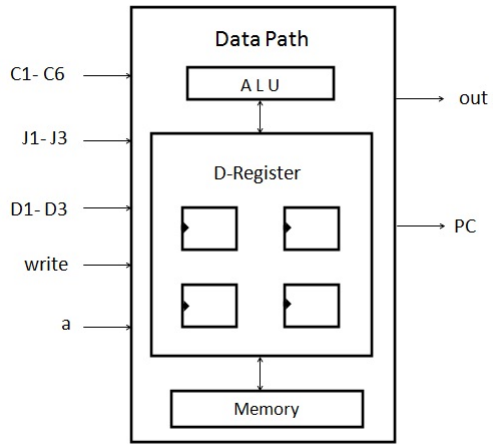

Figure 3.3: Block Diagram of Data Path

3.4 Instruction Set Architecture

Instruction Set Architecture of the proposed processor consists of two instructions, namely Address instruction and Data Instruction.

3.4.1 Address Instruction

The format of Address Instruction is

0vvv vvvv vvvv vvvv

where v's refer to the address where data has to be stored in the Data Memory. 'v' can take value either '0' or '1'.

3.4.2 Data Instruction

The format of the Data Instruction is

111a C1C2C3C4 C5C6D1D2 D3J1J2J3

where C1-C6 refer to Compare computation, D1-D3 refer to Destination where data has to be stored and J1-J3 refer to Jump operations. In Table 3.1 'A' stands for data stored in A-Register, 'D' stands for data stored in D-Register and 'M' stands for data from memory.

C1= zx, it is set to '1' when 'x' has to be equal to '0'.

C2= nx, it is set to '1' when 'x' has to be complemented.

C3= zy, it is set to '1' when 'y' is equal to '0'.

C4= ny, it is set to '1' when 'y' has to be complemented.

C5=f, if 'f'= '1' output is equal to x+y. else output is equal to x&y.

Table 3.1: Computation Specification

when a=0 comp mnemonic	C1	C2	C3	C4	C5	C6	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	

C6= no, if set= '1' output should be complemented.

Table 3.2: Destination Specification

D1	D2	D3	Mnemonic	Destination
0	0	0	Null	Value is not stored
0	0	1	M	Memory A
0	1	0	D	D Register
0	1	1	MD	Memory A and D Register
1	0	0	A	A Register
1	0	1	AM	A Register and memory A
1	1	0	AD	A Register and D Register
1	1	1	AMD	A Register, memory A and D Register

Table 3.3: Jump Specification

J1 (out<0)	J2 (out=0)	J3 (out>0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If out >0 jump
0	1	0	JEQ	If out =0 jump
0	1	1	JGE	If out >=0 jump
1	0	0	JLT	If out <0 jump
1	0	1	JNE	If out !=0 jump
1	1	0	JLE	If out <=0 jump
1	1	1	JMP	Jump

Chapter 4

Implementation

4.1 Blocks and Circuits

4.1.1 Introduction

All the hardware devices are constructed using integrated package of elementary logic gates. And these gates are in turn, built from NAND gate. All of these gates are typically implemented by transistors, and each transistor is made of silicon chips.

The design of a computing system can be described bottom-up, focusing on lower-level modules and can be used to construct more complex ones. We began with the most basic elements like logic gates and went upward, constructing general purpose processor.

Every digital device is based on a set of chips designed to store and process data. All these chips are made from the same building blocks of elementary logic gates. We start with one logic gate i.e, NAND gate and built all the other logic gates from it.

Boolean algebra deals with binary values that are labeled '1' or '0'. A Boolean function is a function that operates on binary inputs and returns binary outputs. Computer hardware is based on the representation and manipulation of binary values. Every Boolean function can be expressed using Boolean expression. For each row, we construct term by ANDing together that fix the values of all the row's inputs. Every Boolean function, can be expressed using only three Boolean operators AND, OR, and NOT.

The NAND function has a property i.e, each one of the operations AND, OR, and NOT can be constructed from NAND alone.

e.g., x OR y = (x NAND x) NAND (y NAND y)

Once we have physical device that implements NAND, we can use many copies of this device to implement the hardware of any Boolean function.

A gate is a physical device that implements a Boolean function. Most gates are implemented as transistors etched in silicon, packaged as chips. Our hardware design starts from such primitive gates and designs more complicated functionality by interconnecting them, leading to the construction of composite gates.

4.1.2 Gates

All of these basic logic gates are built from NAND gates alone.

The NAND Gate

The starting point of our Computer Architecture is the NAND gate, from which all other gates and chips are built. Behavioral code for NAND gate is written and simulated. Truth table and symbol of NAND gate is shown in Table 4.1 and Figure 4.1 respectively.

Table 4.1: Truth table of NAND gate

x	y	out
0	0	1
0	1	1
1	0	1
1	1	0

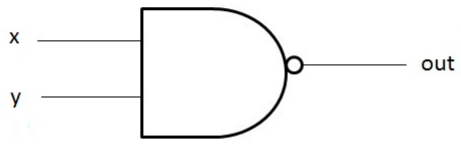

Figure 4.1: Symbol of NAND gate

NOT Gate

The single-input NOT gate, also known as inverter, inverts its input from '0' to '1' and vice versa. Truth table and implementation of NOT gate is shown in Table 4.2 and Figure 4.2 respectively.

NOT x = (x NAND x) NAND (x NAND x)

Table 4.2: Truth table for NOT gate

x	out
0	1
1	0

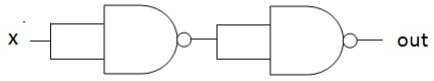

Figure 4.2: Symbol of NOT gate

AND Gate

The AND function returns '1' when both its inputs are '1', and '0' otherwise. Truth table and implementation of AND gate is shown in Table 4.3 and Figure 4.3 respectively.

x AND y = (x NAND y) NAND (x NAND y)

Table 4.3: Truth table of AND gate

x	y	out
0	0	0
0	1	0
1	0	0
1	1	1

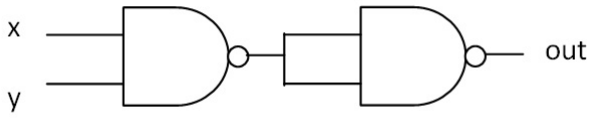

Figure 4.3: Implementation of AND gate using NAND gate

OR Gate

The OR function returns '1' when at least one of its inputs is '1', and '0' otherwise. Truth table and implementation of OR gate is shown in Table 4.4 and Figure 4.4 respectively.

x OR y = (x NAND x) NAND (y NAND y)

Table 4.4: Truth table for OR gate

x	y	out
0	0	0
0	1	1
1	0	1
1	1	1

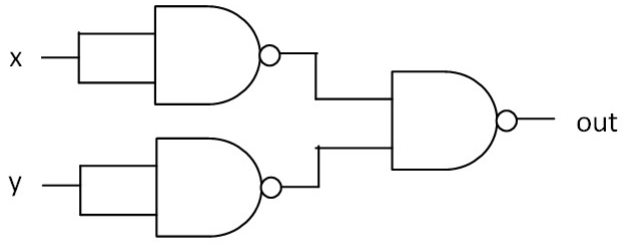

Figure 4.4: Implementation of OR gate using NAND gates

XOR Gate

The XOR function, also known as Exclusive OR, returns '1' when its two inputs have opposing values, and '0' otherwise. Truth table and implementation of XOR gate is shown in Table 4.5 and Figure 4.5 respectively.

Table 4.5: Truth table for XOR gate

x	y	out
0	0	0
0	1	1
1	0	1
1	1	0

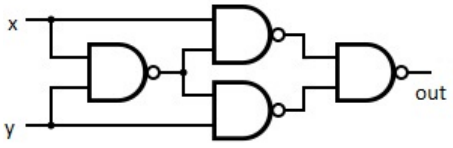

Figure 4.5: Implementation of XOR gate using NAND gates

Multiplexer

A Multiplexer (MUX) 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. Truth table of MUX is shown in Table 4.6.

Table 4.6: Truth table for MUX

x	y	sel	out
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	1

De-Multiplexer

A De-Multiplexer (De-MUX) performs the opposite function of a Multiplexer. It takes a single input and channels it to one of two possible outputs according to a selector bit that specifies which output to chose. Truth table of De-Mux gate is shown in Table 4.7.

Table 4.7: Truth table for De-MUX

sel	a	out1	out2
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

Half Adder

The first step on our way to adding binary numbers is to be able to add two bits. Least significant bit of the addition 'sum', and the most significant bit 'carry'. Truth table and implementation of Half Adder gate is shown in Table 4.8 and Figure 4.6.

Table 4.8: Truth table for Half Adder

x	y	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

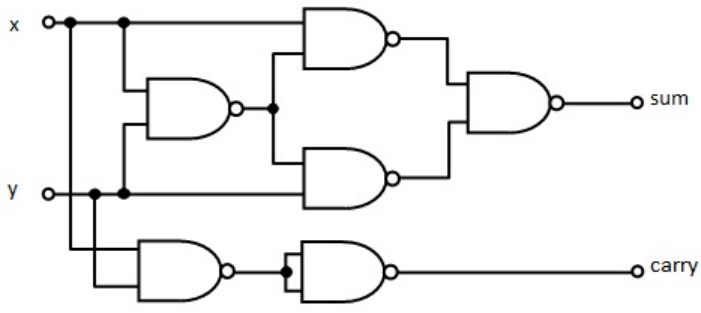

Figure 4.6: Implementation of Half Adder gate using NAND gates

Full Adder

Full Adder is designed to add three bits. It produces two outputs, 'sum' and 'carry'. Truth table and implementation of Full Adder gate is shown in Table 4.9 and Figure 4.7 respectively.

Table 4.9: Truth table for Full Adder

x	y	cin	sum	carry
0	0	0	0	0
0	0	1	1	0
o	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

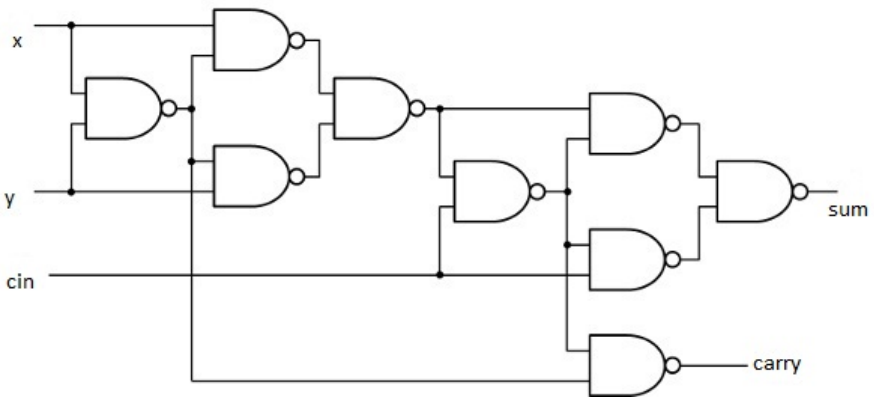

Figure 4.7: Implementation of Full Adder gate using NAND gates

4.1.3 Arithmetic and Logic Unit (ALU)

ALU computes a fixed set of functions out = f(x,y) where 'x' and 'y' are the two 16-bit inputs, 'out' is the 16-bit output, and 'f' is an arithmetic or logical function selected from eighteen possible operations. We instruct the ALU to compute the required computation by setting six input bits, called control bits, to the selected binary values.

The exact input-output specification is as shown in the chapter 3. Here each one of the six control bits instructs the ALU to carry out a certain operation. The combined effects of these operations cause the ALU to compute a variety of useful functions. Since the overall operation is driven by six control bits, programming our ALU to compute a certain function f(x,y) is done by setting the six control bits to the code of the desired function. For example, if C1-C6 is '001110' where the ALU is instructed to compute the function 'x-1'. The 'C1'

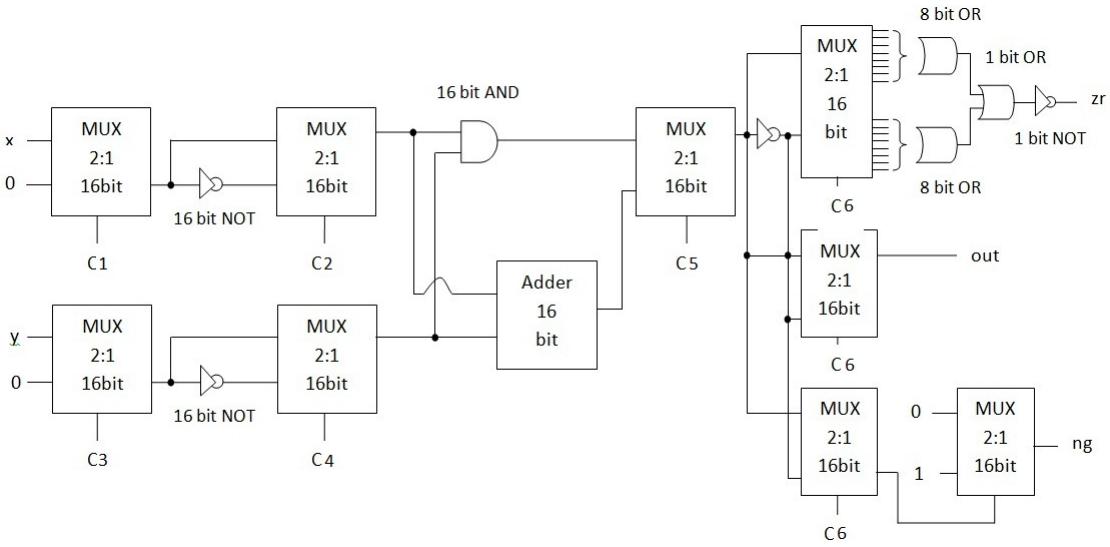

Figure 4.8: Implementation of ALU

and ‘C2’ bits are ‘0’, so the ‘x’ input is neither zeroed nor negated. The ‘C3’ and ‘C4’ bits are ‘1’, so the ‘y’ input is first zeroed, and then negated bitwise. Bitwise negation of zero, (0000000000000000), gives ‘1111111111111111’, the 2’s complement code of ‘-1’. Thus the ALU inputs end up being ‘x’ and ‘-1’. Since the bit is ‘1’, the selected operation is arithmetic addition, causing the ALU to calculate ‘x+(-1)’. Finally, since the ‘C6’ bit is ‘0’, the output is not negated but rather left as is. The ALU ends up computing ‘x-1’.

Implementation of ALU is shown in the Figure 4.8. ‘x’ (one of the input) or ‘0’ inputs to 16-bit 2:1 MUX is selected using ‘C1’, and output of which is given as first input to the second MUX and output of first MUX is inverted and given as another input to 16-bit 2:1 MUX, here ‘C2’ is used as the select line. Same process is repeated for another input ‘y’ where ‘C3’ and ‘C4’ are used as select lines for two MUXes respectively. To perform Addition or Subtraction both the outputs of 16-bit 2:1 MUXes are given to 16-bit Full Adder. The outputs of MUXes are ANDed and given as the input to 16-bit 2:1 MUX, output of the Full Adder i.e, sum is given as another input and carry is ignored where ‘C5’ is used to select one of the two inputs to the MUX. Output of the MUX is inverted and another MUX is used to select between the previous output of the MUX and inverted one, where ‘C6’ is used as select bit. To get flag bits ‘zr’ and ‘ng’ to identify if the output is zero or negated the circuit is constructed as shown in Figure 4.8.

4.1.4 Sequential Logic

Combinational chips compute functions that depend solely on combinations of their input values. These relatively simple chips provide many important processing functions, but they

cannot maintain state. Since computers must be able to not only compute values but also store and recall values, they must be equipped with memory elements that can preserve data over time. These memory elements are built from sequential circuits. The implementation of memory elements involve clocking, and feedback loops. Using Flip-Flops as elementary building blocks, we can build all the memory devices. From registers to memory banks and counters.

Memory Elements

D Flip-Flop (DFF), has single-bit data input and a single-bit data output. In addition, the DFF has a clock input that continuously changes according to the clock signal. Together, the data and the clock inputs enable the DFF to implement *out(t) = in(t-1)*, where 'in' and 'out' are the gate's input and output values. DFF simply outputs the input value from the previous time unit. Therefore this DFF can be used in all the hardware devices that computers use to maintain state, from registers to large Random Access Memory (RAM) units. Figure 4.9 shows positive edge DFF is built using NAND gates. A Register is a

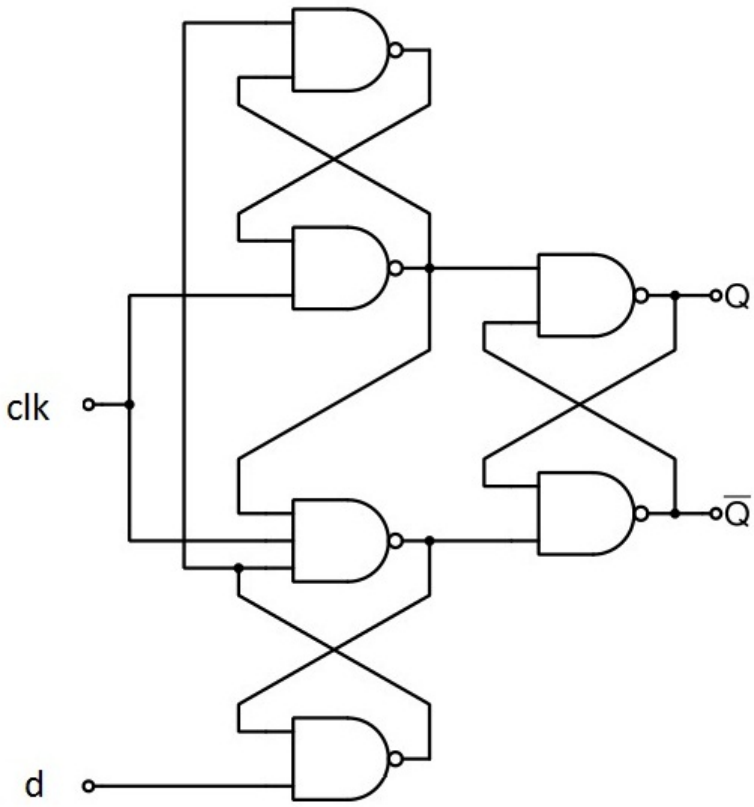

Figure 4.9: Implementation of D Flip-Flop using NAND gates

storage device that can store a value over time, implementing behavior out(t) = out(t-1). A DFF, on the other hand, can only output its previous input, namely, out(t) = in(t-1). Thus

a register can be implemented from a DFF by feeding the output of the DFF back into its input. The output of device at any time *t* will give its output at time *t-1*, thus forming a storage unit. Implementation of 1-bit Register using MUX and DFF is shown in Figure 5.8. Once we have these basic elements like Registers, we can build memory banks of any

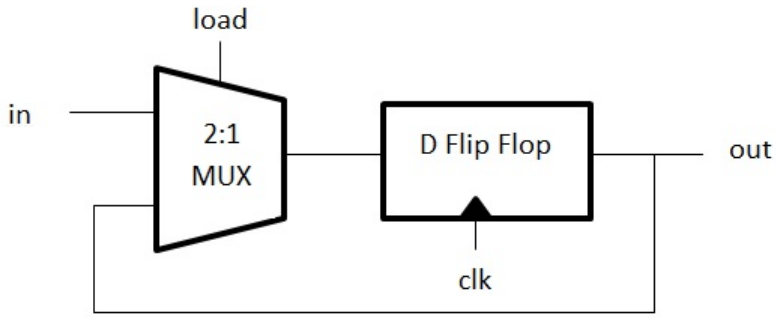

Figure 4.10: Implementation of Register

length. This can be done by stacking together many Registers to form a Random Access Memory (RAM) unit. RAM should be able to access randomly chosen words irrespective of its location. Each word in the 24 K register RAM has a unique address from 0 to (24 K)-1, accordingly each register will be accessed. Depending on the address, RAM should able to write/read the information, therefore De-MUX is used to select the corresponding register whose address has been specified. RAM device accepts three inputs, a data input, an address input, and a load bit. Sixteen 1-bit Registers are placed in serial to form 16-bit Register. Eight such Registers are stacked to form RAM8. This process is continued until we get 24 K RAM. This process is shown in Figure 4.11. When 'load' is low Read operation is performed i.e, the value which is stored in the RAM is read from the location specified by the 'address' bits and when 'load' is high write operation is performed i.e, the value is written into the RAM to the location specified by the 'address' bits.

Program Counter

A Program Counter (PC) is a register in a computer processor that contains the address of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by '1'. PC is incremented after fetching an instruction, and holds the memory address of the next instruction that would be executed. Instructions are usually fetched sequentially from memory, but control transfer instructions change the sequence by placing a new value in the PC. These include branches, subroutine

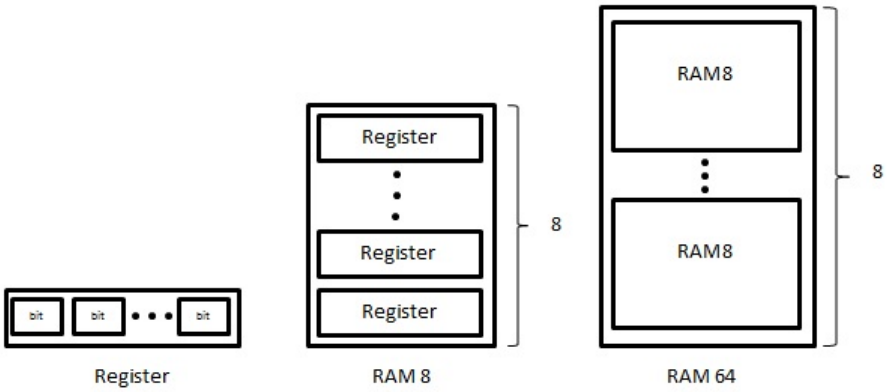

Figure 4.11: Implementation of RAM blocks

calls, and returns. A transfer that is conditional on the truth of some assertion lets the computer follow a different sequence under different conditions. A branch provides that the next instruction is fetched from somewhere else in memory. A subroutine call not only branches but saves the preceding contents of the PC somewhere. A return retrieves the saved contents of the PC and places it back in the PC, resuming sequential execution with the instruction following the subroutine call. Block of Program Counter is shown in Figure 4.12. When the 'reset' value is set to '1' the output value of the program counter becomes

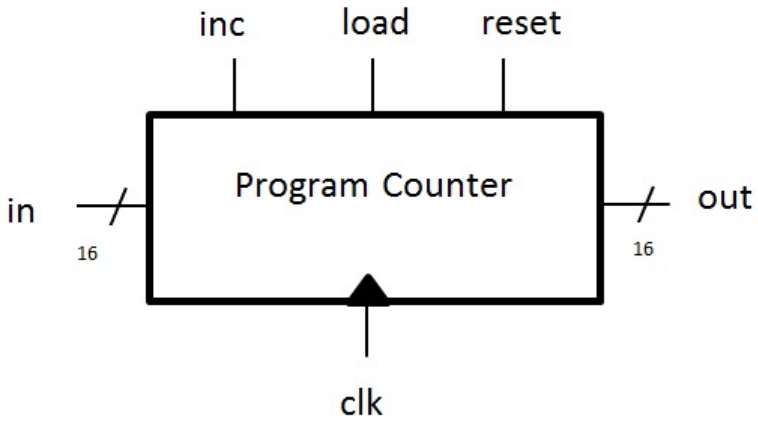

Figure 4.12: Block Diagram of Program Counter

'0'. When the 'load' value is set to '1' the input value is loaded to the output. When the 'inc' value is '1' and the 'load' value is set to '0' the output of the program counter will be the incremented value of the previous output value. 'reset' is used as the select line of the MUX. When the 'reset' value is set to '1' the output value will be set to '0'. 'load' is also used as the select line of one more MUX. When the load value is set, the MUX selects the input and loads it to the MUX and also to the Full Adder since the 'load' value is '1' the output of the AND gate becomes '0' because of the NOT gate present and the output of the

AND gate acts like a select line to the MUX. If the 'reset' is set to '1' then the MUX output is directly given to the output. Implementation of Program Counter is shown in Figure 4.13.

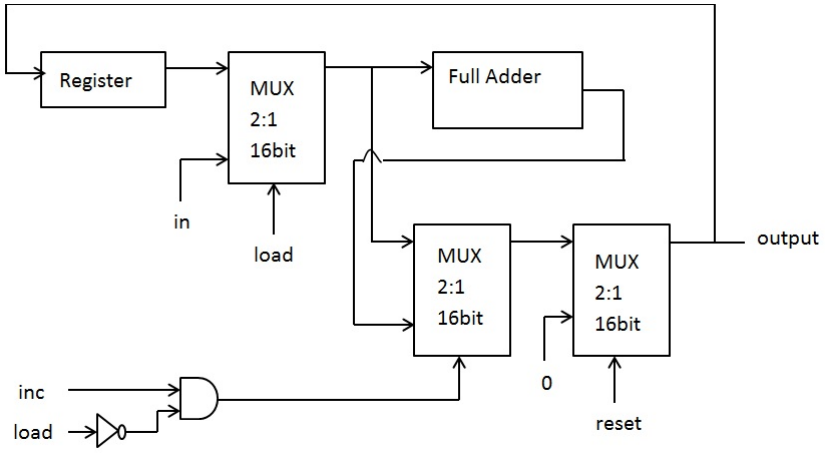

Figure 4.13: Implementation of Program Counter

4.1.5 Processor Block

The instruction that is fetched from ROM is decoded to perform necessary operations in CPU. If istr[15] is '1' then the previous output of ALU is used for operation else, new data is taken from ROM. If istr[15] is '1' then it depends on istr[5] if it is '1' then the value is stored in the A-Register. istr[12] is used as the select line to select between A-Register output and the 'in' from RAM. The output of the MUX is given as the second input to the ALU. The first input to the ALU is the output of D-Register which is used to store the previous output of the ALU for the next operation, if istr[4] is '1' then the value is stored. 'zr' and 'ng' are the flag bit outputs of ALU used to indicate whether the output is zero or negative is used along with istr[3:0] and istr[15] to load or increment operation on Program Counter. The circuit is as shown in Figure 4.14. The output of the Program Counter is used to fetch the next instruction. If both istr[15] and istr[3] is '1' then the 'write' bit is '1' indicates the output should written in to RAM.

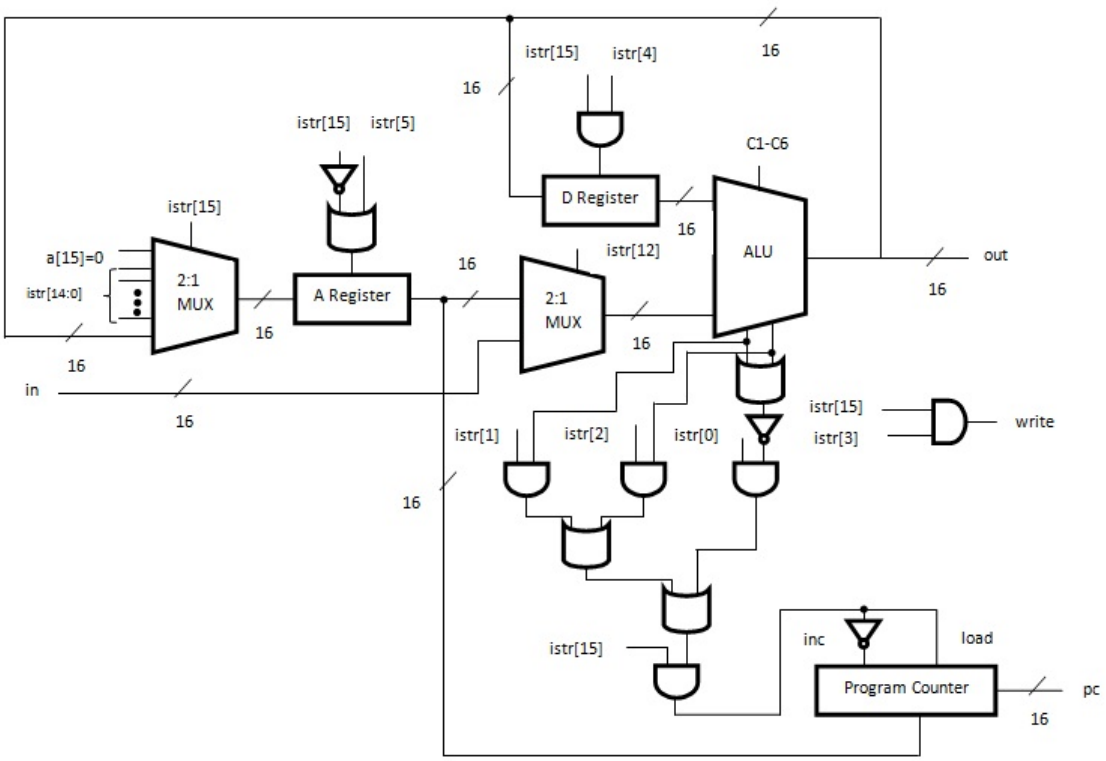

Figure 4.14: Implementation of CPU

4.2 Layout Generation

4.2.1 ASIMUT

ASIMUT (A SIMUlation Tool) is a command line tool. It can be used to check the correctness of the description of a circuit in its VHDL behavioral or structural form. It can also be used for simulating and generating the outputs of the circuit for a given set of inputs. We can run asimut by typing

% asimut [target_file] [input_pattern_file] [output_pattern_file]

To check the syntax correctness of the nand_gate which is saved as nand_gate.vbe, we use the following command

% asimut -b -c nand_gate

Note that we only indicate the target file name since its type (.vbe extension) is explicitly stated by the option -b. The output is as shown in Figure 4.15. When running ASIMUT, as shown above, we used two options. The first one -b tells ASIMUT that the input file is of behavioral type. The second one -c tells ASIMUT to compile the target file. This option makes ASIMUT check the target file for syntax errors before compiling it.

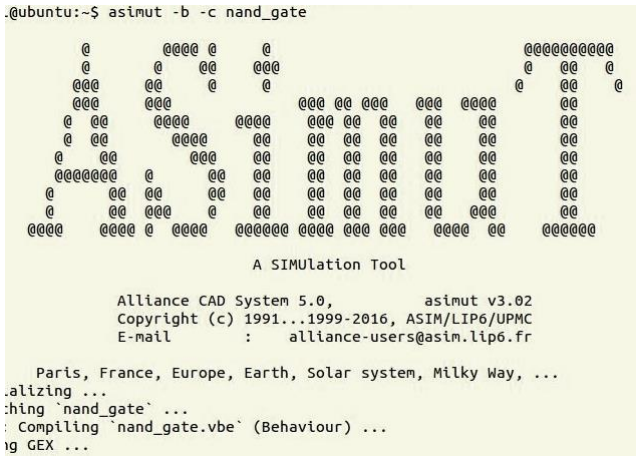

Figure 4.15: Output of the command ASIMUT

4.2.2 BOOM

The BOOM (BOOlean Minimization) tool is used in the first step in the synthesis process of Alliance. It optimizes a behavioral description. We use this tool by giving the command in following format.

% boom [-VTOAP] [-l num] [-d num] [-i num] [-a num] [-sjbgpwtmorn] filename [outname]

Following command is used to run Boolean Minimization on nand_gate.

% boom -l 3 -d 50 nand_gate

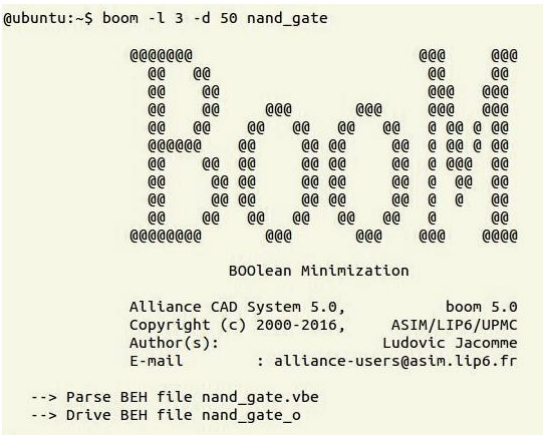

Figure 4.16: Output of the command BOOM

This tells boom to minimize the design described in nand_gate.vbe. The option -l gives the level of optimization (a value between 0 and 3). The option -d indicates what we want to optimize in percentage. A value of '0' indicates that we want to optimize delay. A value of '100' will optimize area. In our example we indicate that both, delay and area must be equally optimized. The corresponding output will be as shown in Figure 4.16.

4.2.3 BOOG

The BOOG (Binding and Optimizing On Gates) tool maps a behavioral description onto a predefined standard cell library. As we have shown in one example of the use of ASIMUT this tool is usually used in a second step of the synthesis process in Alliance. It is used after BOOM. The description optimized by BOOM is the input of this tool. BOOG builds a boolean network equivalent to the description obtained with BOOM. Then for each boolean function of each node of the network it tries to find a cell or a set of cells that implements that function. The result will be a structural description based on the cells of the sxlib library of Alliance. We use this tool by giving the command in following format.

% boog [-hmxold] input_file output_file [lax_file]

To use it on nand_gate we give the following command.

% boog nand_gate_o nand_gate -x 1 -m 2

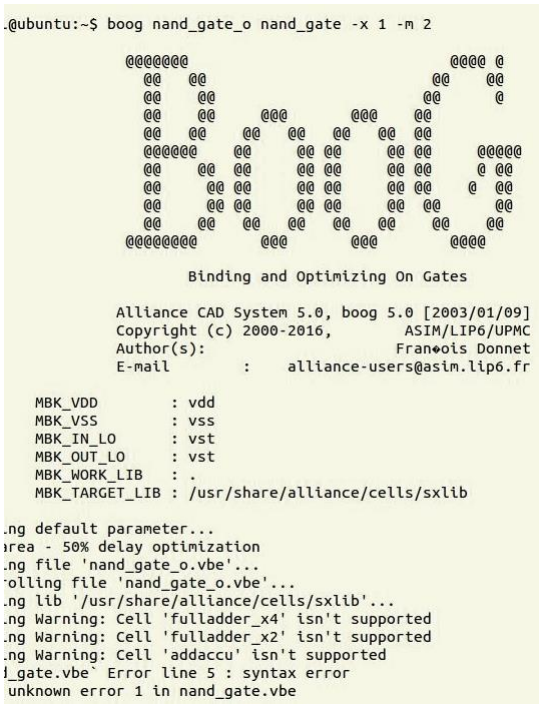

Figure 4.17: Output of the command BOOG

This tells boog to use nand_gate_o.vbe and generate an optimized nand_gate.vst structural file. The option -x indicates that we want to generate a nand_gate.xsc output file. This file will display the cells used and the delay paths. If we indicate a value of '0' for this option only the critical path will be colored. If we indicate a value of '1' all the paths will be colored. The option -m indicates what we want to optimize (we can use values between 0 and 4). A value of '0' indicates that we want to optimize area. A value of '4' will optimize

delay. Here both delay and area must be optimized. The corresponding output will be as shown in Figure 4.17 and Figure 4.18.

```
BEH: unknown error 1 in nand_gate.vbe
Mapping Warning: Cell 'nand_gate' isn't supported
Mapping Warning: Cell 'addaccu_o' isn't supported
Mapping Warning: Cell 'adder4_o' isn't supported
Mapping Warning: Cell 'halfadder_x2' isn't supported
Mapping Warning: Cell 'sff3_x4' isn't supported
Mapping Warning: Cell 'buf_x8' isn't supported
Mapping Warning: Cell 'noa3ao322_x4' isn't supported
Mapping Warning: Cell 'nao22_x4' isn't supported
Mapping Warning: Cell 'inv_x4' isn't supported
Mapping Warning: Cell 'nts_x2' isn't supported
Mapping Warning: Cell 'addaccu_o' isn't supported
Mapping Warning: Cell 'inv_x8' isn't supported
Mapping Warning: Cell 'sff2_x4' isn't supported
Mapping Warning: Cell 'noa2a2a23_x4' isn't supported
Mapping Warning: Cell 'o2_x4' isn't supported
Mapping Warning: Cell 'no4_x4' isn't supported
Mapping Warning: Cell 'no3_x4' isn't supported
Mapping Warning: Cell 'adder4' isn't supported
Mapping Warning: Cell 'oa22_x4' isn't supported
Mapping Warning: Cell 'na2_x4' isn't supported
Mapping Warning: Cell 'halfadder_x4' isn't supported
Mapping Warning: Cell 'noa2a22_x4' isn't supported
Mapping Warning: Cell 'nao2o22_x4' isn't supported
Mapping Warning: Cell 'a3_x4' isn't supported
Mapping Warning: Cell 'ts_x8' isn't supported
Controlling lib '/usr/share/alliance/cells/sxlib'...
Preparing file 'nand_gate_o.vbe'...
Capacitances on file 'nand_gate_o.vbe'...
Unflattening file 'nand_gate_o.vbe'...
Mapping file 'nand_gate_o.vbe'...
Saving file 'nand_gate.vst'...
Quick estimated critical path (no warranty)...173 ps from 'b' to 'y'
Quick estimated area (with over-cell routing)...1000 lambda²
Details...
        na2_x1: 1
        Total: 1
Saving delay gradient in xsch color file 'nand_gate.xsc'...
End of boog...
```

Figure 4.18: Output of the command BOOG

4.2.4 XSCH

This is a X-window graphical tool to visualize schematics. All its functionalities are accessed through its menus.

% xsch [-l file_name] [-xor] [-install] [-force] [-I input_format] [-slide file_name ...]

To see the schematics of the design we use the graphical tool XSCH. We call it with the following command.

% xsch

Then from its File menu we can choose any .vst file in our working directory. The corresponding output will be as shown in Figure 4.19.

4.2.5 OCP

The OCP tool is used for placement in Alliance. It uses as input a netlist of standard cells. The netlist format can be structural VHDL, EDIF or Alliance internal format. Optionally, we can also give a file indicating the placement of connectors (ioc type file). If nonstandard

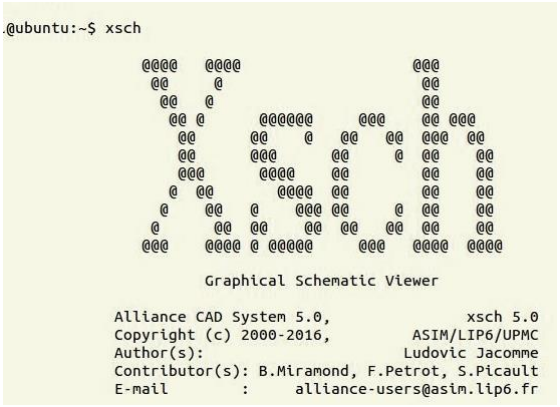

Figure 4.19: Output of the command XSCH

blocks are used in the design, one has to provide the placement for those blocks. The output of the placer will be a physical layout file with placed cells and connectors. We use OCP with the following command.

% ocp [options] netlist outputname

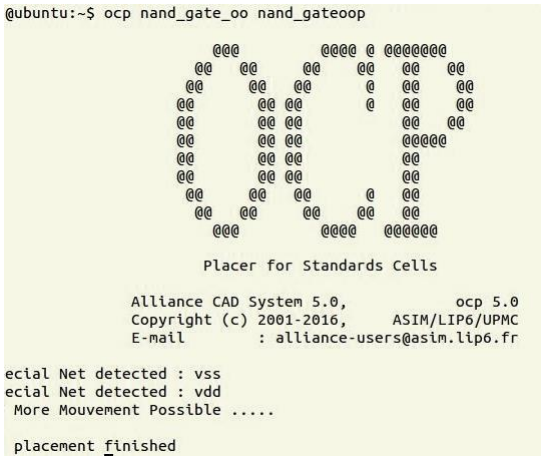

Figure 4.20: Output of the command OCP

Usage of command is shown in Figure 4.20. We can also force a vertical placement of the design indicating the number of rows we want to use, but we must carefully choose the number of rows to avoid the use of an excessive number of tie cells (blank cells).

4.2.6 GRAAL

GRAAL is the hierarchical symbolic layout editor of Alliance. It is a graphical tool that requires an X11 and Motif installation. GRAAL functionalities can be accessed through its seven menus. GRAAL could be called with a command as the one that follows.

% graal [-l file_name] [-scale n] [-debug] [-xor] [-install] [-force]

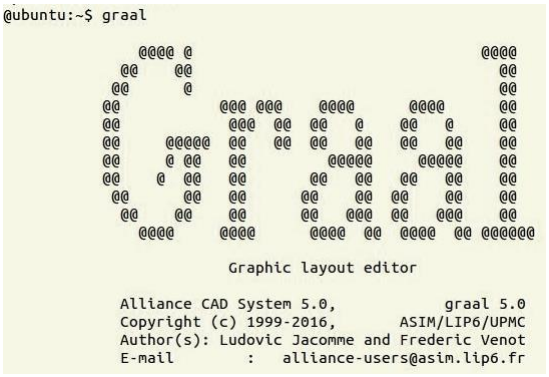

Figure 4.21: Output of the command GRAAL

Placement of standard cells are shown in Figure 4.22 and usage of command is shown in Figure 4.21.

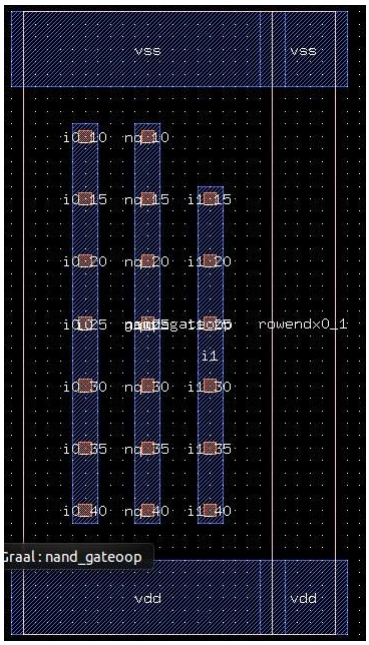

Figure 4.22: Output of the command GRAAL

4.2.7 NERO

NERO (NEgotiating ROuter) is a simple over the cell router provided in Alliance that can process designs of up to 4 K gates in size. It takes as input a netlist and a placement of it. By default it assumes that the netlist and placement file have the same name. The user can provide with an optional different name for the placement file. If a design has a half perimeter greater than 800 lambdas, NERO will use global routing to route it. When NERO uses global routing, the longest nets will be routed first using layers 3 and 4. Then, the remaining nets will be routed using all other available layers. When global routing is used at least four layers are used for routing. The nets in this case are routed from the shortest

to the longest one with the same routing algorithm. We use NERO with the following command.

% nero [options] netlist layout

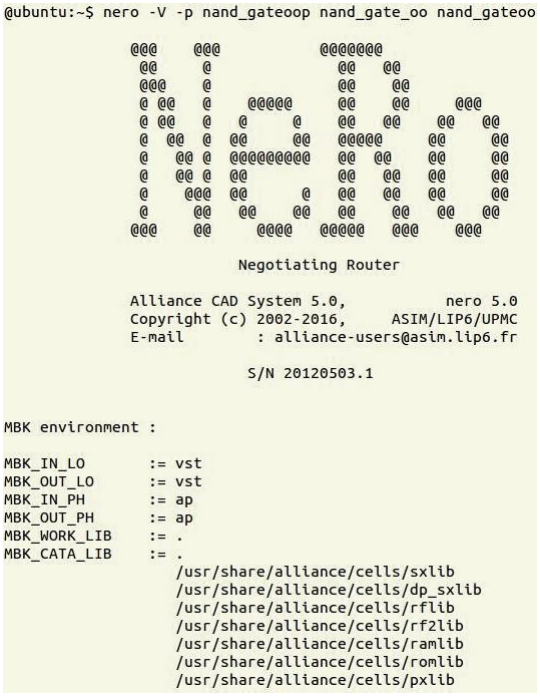

Figure 4.23: Output of the command NERO

Here -v means verbose mode and -p specifies the name for the placement file different from the netlist name. The netlist and layout file names must be different or the first one will be overwritten. Command used and the output at the terminal are shown in Figure 4.23 and Figure 4.24.

4.2.8 COUGAR

This tool is the netlist extractor used in Alliance. COUGAR is a hierarchical extractor that is applied to a symbolic layout, but it can be applied also to a real one if the corresponding technological file (RDS file) is provided. COUGAR can only extract CMOS transistors (no other devices are supported). It also extracts parasitic capacitance and resistance. COUGAR is used as follows.

% cougar [-v] [-c] [-f] [-t] [-ar] [-ac] input_name [output_name]

Here -t command notifies the transistor level extraction, the symbolic layout cell is flattened to transistor layout before extraction. When no options are specified, COUGAR will extract the current hierarchical level. The resulting netlist will be the list of interconnections

```
MBK_CATAL_NAME  := CATAL
MBK_VDD         := vdd
MBK_VSS         := vss
MBK_SEPAR       := .

Loading netlist "nand_gate_oo"...
Loading layout "nand_gateoop"...
Flattening layout...
Flattening netlist...
Building netlist dual representation (lofigchain)...
Binding logical & physical views...

Loading design into grid...
o  Using seed cell "y_ins" (model "na2_x1").
o  Grid offset : (0,0) [adjust (0,0)]
o  Small design, global routing disabled.
o  Allocating grid size [6,11,3].
o  Loading external terminals.
o  Finding obstacles.
o  Loading nets into grid.
o  Allocating the net scheduler.
o  Reading power grid.

Local routing stage.
- [   4] (hp :=     0) "vdd"
- [   3] (hp :=     0) "vss"
- [   2] (hp :=     5) "b"
- [   1] (hp :=     6) "a"
- [   0] (hp :=     6) "y"

Routing stats :
- routing iterations    := 0
- re-routing iterations := 0
- ratio                 := -nan%.

Dumping routing grid.
Saving MBK figure "nand_gateoor".
Saving layout as "nand_gateoor"...
```

Figure 4.24: Output of the command NERO

```
@ubuntu:~$ cougar -t nand_gateoor nand_gate_oo

          @@@@ @
        @@    @@
       @@      @
      @@       @    @@@    @@@  @@@@    @@@@@@   @@@@    @@@ @@@
      @@         @@   @@   @@    @@   @@  @@   @@   @    @@@  @@
      @@          @@     @@  @@    @@   @   @@   @@   @@   @@   @@
      @@          @@     @@  @@    @@   @   @      @@@@@   @@
      @@          @@     @@  @@    @@    @@@     @@   @@   @@
       @@       @ @@     @@  @@    @@   @@      @@   @@   @@
        @@     @@  @@    @@   @@   @@@   @@@@@@  @@   @@@   @@
          @@@@      @@@      @@@@  @@ @@   @@@  @@@@  @@ @@@@
                                    @     @
                                     @@@@@

                Netlist extractor ... formerly Lynx

            Alliance CAD System 5.0,         cougar 1.21
            Copyright (c) 1998-2016,      ASIM/LIP6/UPMC
            Author(s): Ludovic Jacomme and Gregoire Avot
            Contributor(s):             Picault Stephane
            E-mail        :  alliance-users@asim.lip6.fr


   ---> Extract symbolic figure nand_gateoor
   <--- done !

   ---> Total extracted capacitance
   <--- 0.0pF
```

Figure 4.25: Output of the command COUGAR

at the current layout hierarchy level. COUGAR is used to verify designs. The command used is as shown in Figure 4.25.

4.2.9 S2R

S2R (Symbolic to Layout) is the tool in Alliance that let us translate our symbolic design to a real one. For this, S2R uses the technology file defined by the RDS_TECHNO_NAME environment variable. If we provide an appropriate technology file the output of S2R will be a layout file that can be given to a foundry for the fabrication of the design. We use S2R with the following command.

% s2r [-tc1rv] source [result]

```
@ubuntu:~$ s2r -v nand_gateoor nand_gatecore

                                @@@@
                               @   @@
                              @@    @@
                      @@@@@@  @@@   @@ @@@ @@@
                     @@    @   @   @@   @@@  @@
                     @@@           @    @@   @@
                      @@@@        @     @@
                        @@@@     @      @@
                     @    @@@   @    @  @@
                     @@    @@  @@@@@@   @@
                     @ @@@@@  @@@@@@@  @@@@

                    Symbolic to Real layout converter

               Alliance CAD System 5.0,            s2r 5.0
               Copyright (c) 2002-2016,     ASIM/LIP6/UPMC
               E-mail        : alliance-users@asim.lip6.fr


  o loading technology file : /etc/cmos.rds
  o loading all level of symbolic layout : nand_gateoor
  o removing symbolic data structure
  o layout post-treating
          with top connectors,
          with sub connectors,
          with signal names,
          without scotch.
  --> post-treating model na2_x1
      rectangle merging :
  --> post-treating model rowend_x0
      rectangle merging :
  --> post-treating model nand_gateoor
      ring flattenning  :
      rectangle merging :
  o saving nand_gatecore.cif
  o memory allocation informations
  --> required rectangles = 0  really allocated = 0
  --> Number of allocated bytes: 50706
```

Figure 4.26: Output of the command S2R

-v specifies verbose mode on. The command used is as shown in the Figure 4.26.

4.2.10 DREAL

DREAL tool is a hierarchical real layout editor similar to GRAAL (the tool we used to visualize the placed and routed designs), but it has no design rule checker neither an extractor as GRAAL does. We use DREAL providing the following command.

% dreal [-l file_name] [-xor] [-debug] [-install] [-force]

This tool is usually used on the output of the Alliance tool S2R. S2R converts the symbolic layout obtained with NERO into a real technology layout in 'cif' or 'gds' format. We will run S2R on the routed symbolic layout of our NAND gate. We then use DREAL to see the generated nand_gatecore real layout file. DREAL requires a X11 environment and the usage of this command is shown in Figure 4.27.

% dreal -l nand_gatecore

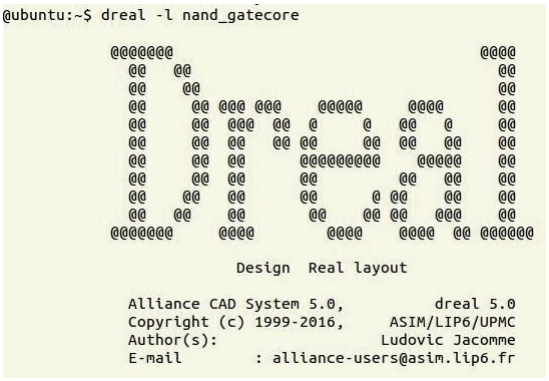

Figure 4.27: Output of the command DREAL

-l loads the file name (with or without extension). The corresponding real layout after flattening is shown in Figure 4.28.

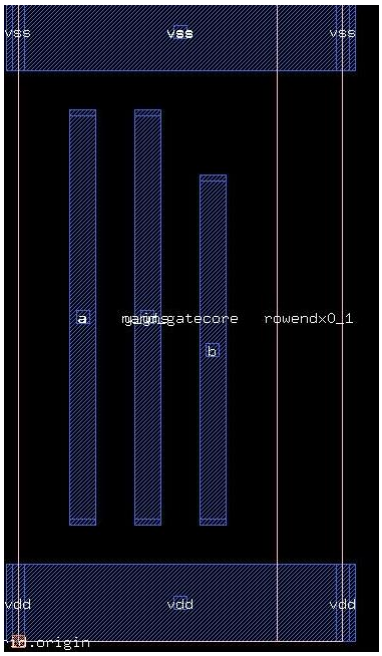

Figure 4.28: Layout of NAND gate using command DREAL

4.2.11 DRUC

DRUC (Design RUle Checker) is the Alliance parametrized VLSI design rule checker. The rules it uses are defined by the RDS_TECHNO_ NAME environment variable. DRUC flattens all the hierarchy of the design to check if there is any violation to the specified design rules, so when applying it is necessary to assure that root and instantiated cells are in the current directory. Lets apply DRUC to our nand_gateoor.ap design file to check it and the

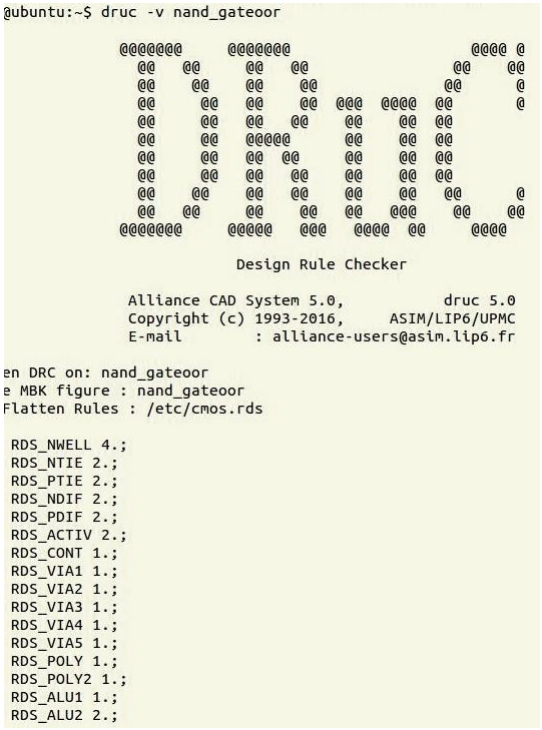

Figure 4.29: Output of the command DRUC

output is shown in Figure 4.29. From the output shown in Figure 4.30, we see that no error has been detected. And the layout generated by OCP and NERO does not contain violations to the design rules specified in the cmos.rds file.

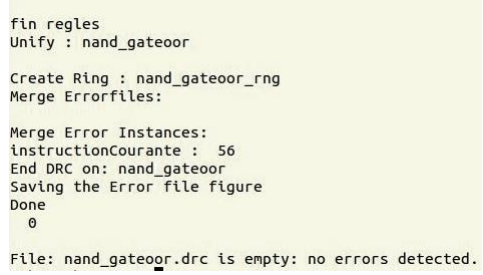

Figure 4.30: Output of the command DRUC

Chapter 5

Results

5.1 Arithmetic and Logic Unit (ALU)

Arithmetic and Logic Unit performs eighteen operations. The output shown in Figure 5.1 is seen Xilinx ISE Simulator (ISim) . The two inputs to the ALU ‘x’, ‘y’ are ‘1111000011110000’ and ‘0000111100001111’ and control bits C1-C6 are given as ‘111111’. From the Computation Specification Table shown in Chapter 3, we see that corresponding output should be equal to ‘1’. Similarly if the control bits are ‘000000’ the output is ANDing of two inputs ‘x’ and ‘y’. Hence the output is ‘0000000000000000’ as seen in Figure 5.1. This shows our circuit output matches with the theoretically calculated output of ALU.
RTL schematic obtained from Xilinx software for ALU code is as shown in Figure 5.2. The maximum path delay of ALU is 25.671 ns with 21 levels of logic.

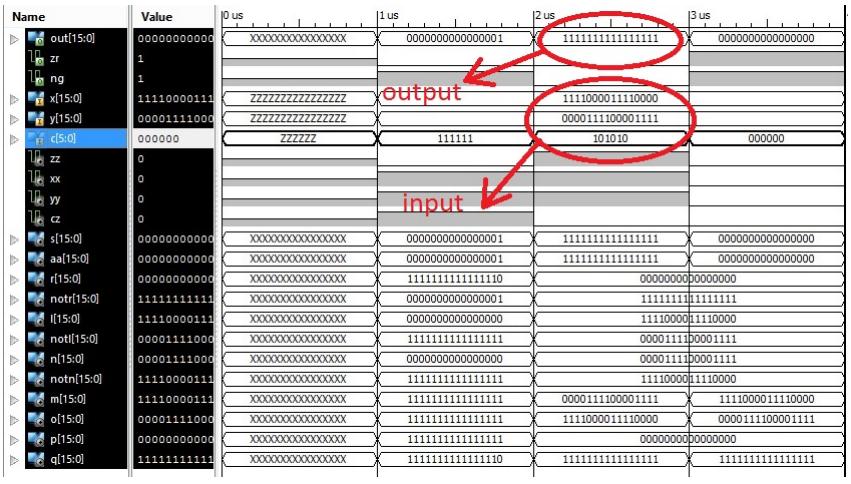

Figure 5.1: Output of ALU

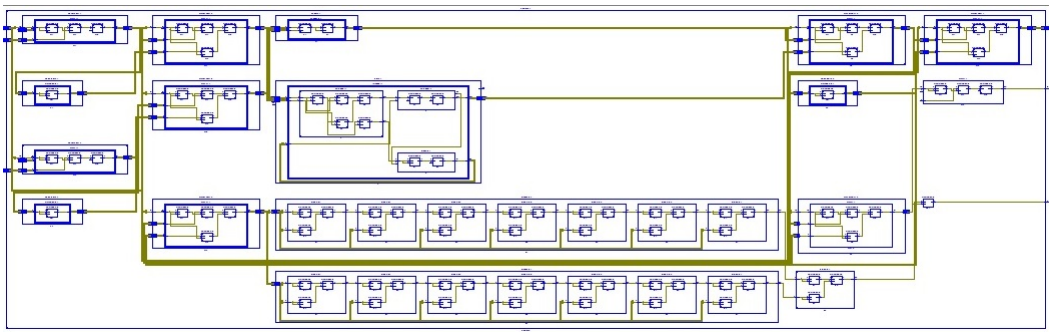

Figure 5.2: Schematic of ALU

5.2 Program Counter (PC)

Program Counter is set to '0' when 'reset' is high and PC is incremented when 'inc' is made high. PC can be made to point to a particular address specified by 'in' when load is made high. The simulation output is shown in the Figure 5.3. RTL schematic obtained from Xilinx software is shown in Figure 5.4. The maximum combinational path delay is 83.523 ns with 78 levels of logic.

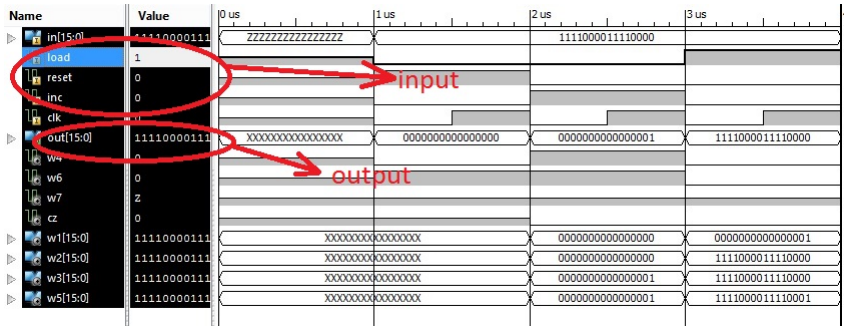

Figure 5.3: Output of Program Counter

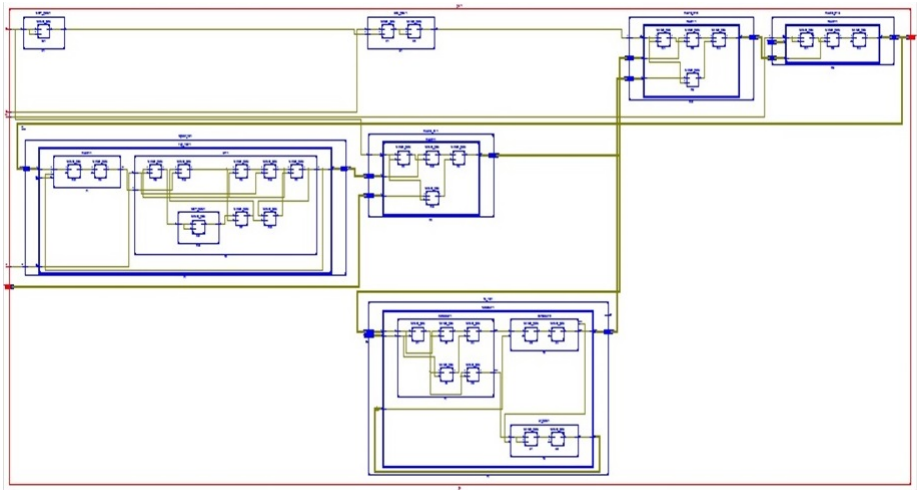

Figure 5.4: Schematic of Program Counter

5.3 Central Processing Unit (CPU)

As we could see from Figure 5.5, when ‘reset’ is high, PC is reset to ‘0000000000000000’, depending on Instruction bits ‘istr’, CPU decodes Computation Specification, Jump Specification, Destination Specification, etc. The Computation Specification for the instruction ‘1110010101001000’ is ‘101010’, from the Computation Specification Table the output should be equal to ‘0’, this can be observed from Figure 5.5. For the next clock cycle, since the ‘reset’ was low, PC is incremented. RTL schematic obtained from Xilinx software is shown in Figure 5.6. The maximum combinational path delay of CPU is 219.326 ns.

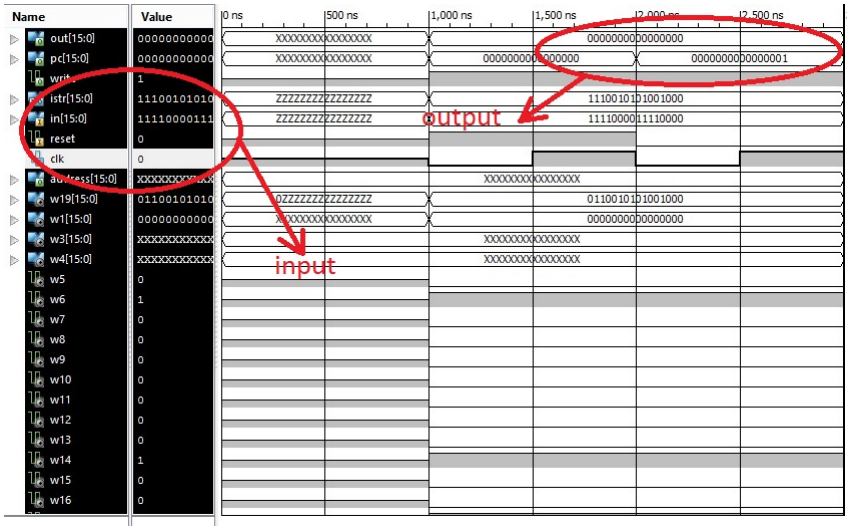

Figure 5.5: Output of CPU

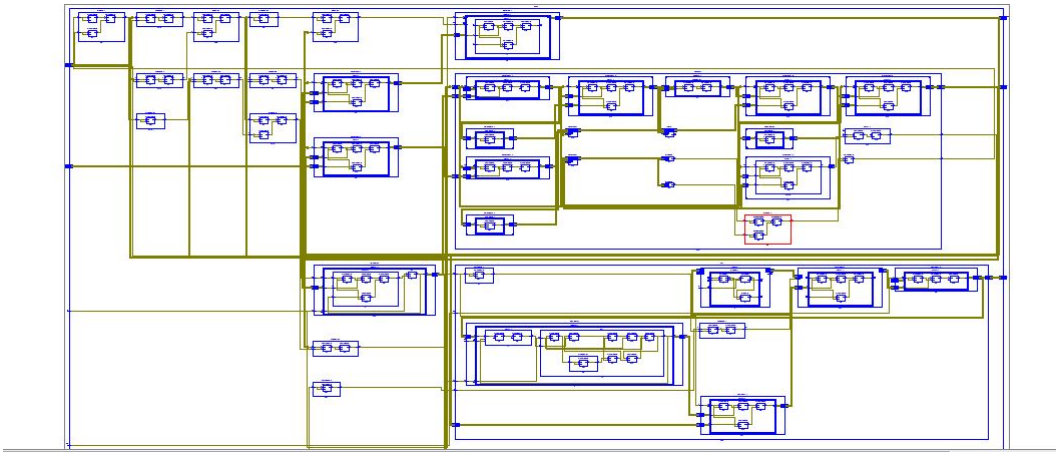

Figure 5.6: Schematic of CPU

5.4 Random Access Memory (RAM)

When the ‘address’ is ‘00000’ the input to the RAM ‘1111000011110000’ is stored at the specified location when ‘load’ is high. When ‘load’ is low, the new input is not loaded to

RAM. But, the value stored in the RAM is read at the output from the address specified as shown in Figure 5.7. Each RAM block is made up of Registers, RTL schematic of the 1-bit Register is shown in Figure 5.8, RTL schematic of 8x16 RAM obtained from Xilinx software is shown in Figure 5.9. The maximum delay of RAM 8x16 is 12.069 ns with 9 levels of logic. 9

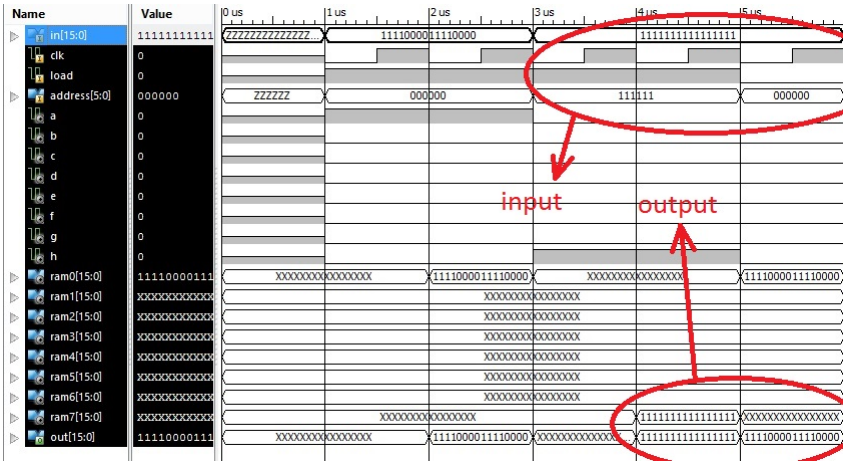

Figure 5.7: Output of RAM

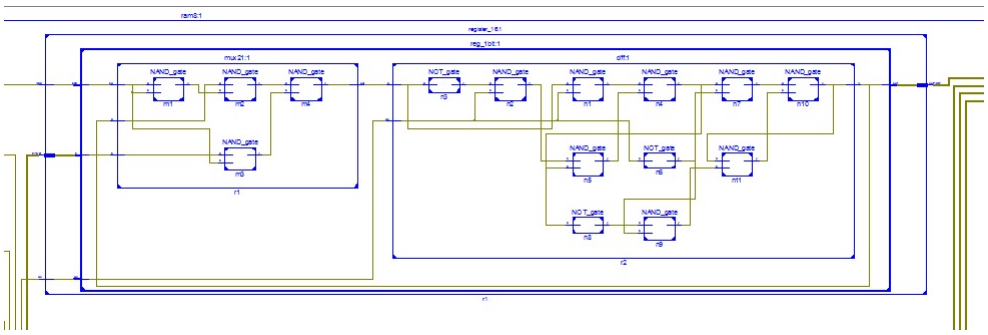

Figure 5.8: Schematic of 1-bit Register

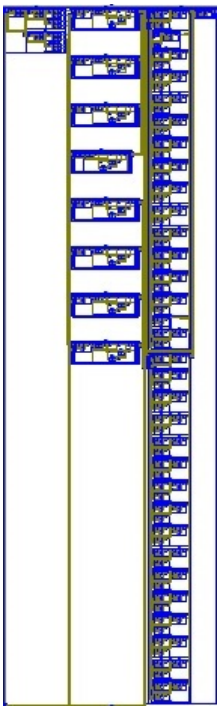

Figure 5.9: Schematic of RAM

5.5 Processor

Processor starts its execution by setting ‘reset’ high and applying ‘clk’ to the processor. When ‘reset’ equals to ‘1’ PC will be set to ‘0000000000000000’ to fetch the first instruction from ROM. We have written a code for incrementing decimal number 10 and store it in RAM address ‘0000000000010001’. In the program shown in Figure 5.10 first instruction

```
1  `timescale 1ns / 1ps
2
3  module rom_add(istr, pc, clk);
4
5  input [4:0] pc;
6  input clk;
7  output [15:0] istr;
8  reg  [15:0] istr;
9
10 always @ (posedge clk)
11 begin
12 case(pc)
13 5'b00000: istr = 16'b0000000000001010;
14 5'b00001: istr = 16'b1110000011010000;
15 5'b00010: istr = 16'b1110111110010000;
16 5'b00011: istr = 16'b0000000000010001;
17 5'b00100: istr = 16'b1110001100001000;
18 endcase
19 end
20 endmodule
```

Figure 5.10: Code to increment and store in RAM

sets A-Register of CPU to decimal value 10. Second instruction is used to store A-Register value to D-Register. Third instruction increments the value of D-Register. Fourth and fifth instruction directs the value of D-Register to be stored in RAM memory of address 17. The output of this program is shown in Figure 5.11.

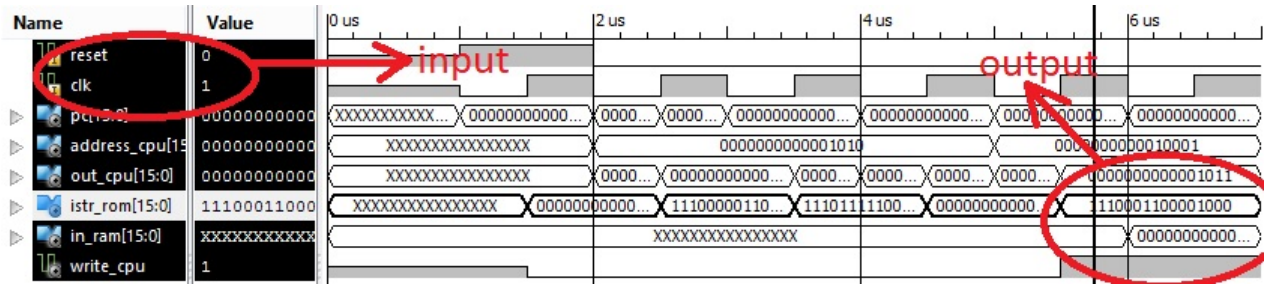

Figure 5.11: Output of Program to Increment and store in RAM

5.6 Layout

The behavioral code for Full Adder has been simulated using ASIMUT command. The synthesis process is carried out and BOOM command performs boolean minimization. The placement of standard cell has been done using OCP command after this process the output can be seen as in Figure 5.12 using GRAAL command. Netlist is extract from the symbolic

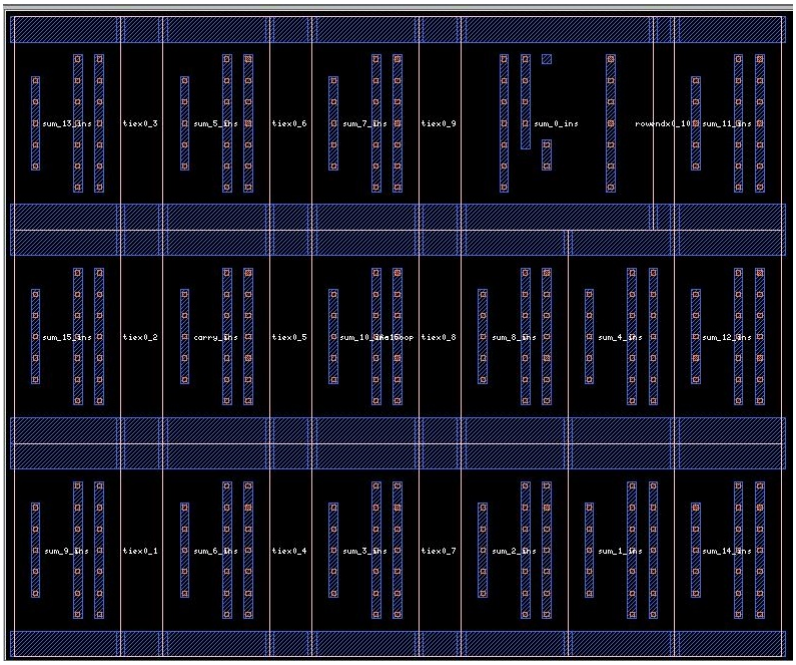

Figure 5.12: Placement of Standard Cells

layout obtained from previous steps. S2R command is used to translate symbolic design to the real layout and the output is as shown in Figure 5.13 and Figure 5.14. The design rule checked for Full Adder using DRUC command and it is found that there is no design rule violation. This is shown in Figure 5.15.

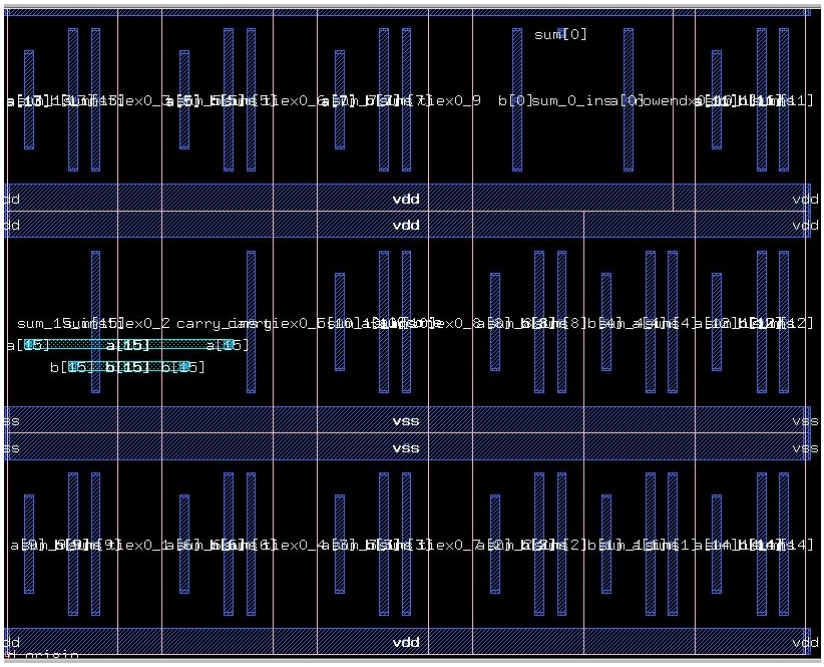

Figure 5.13: Layout of Full Adder

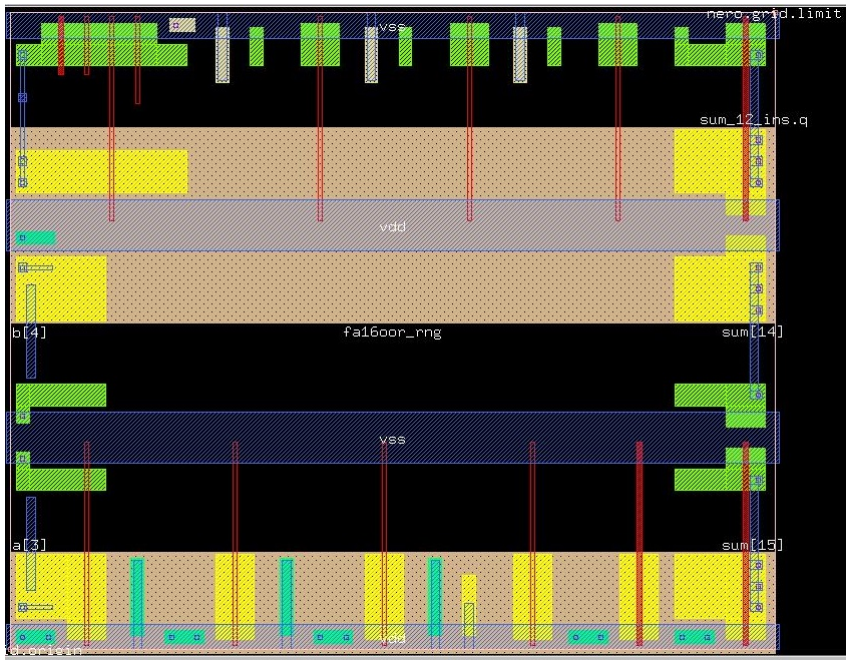

Figure 5.14: Layout of Full Adder

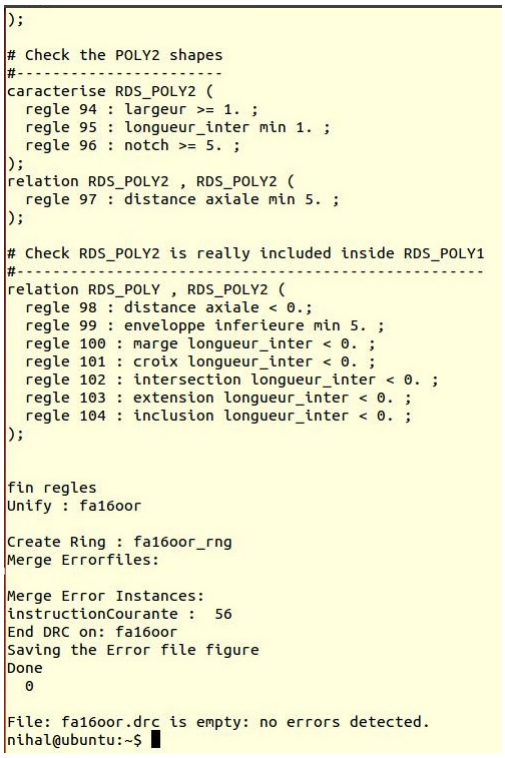

Figure 5.15: Design Rule Checker output for Full Adder

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

A 16-bit processor is designed and simulated in this work. Verilog code for Central Processing Unit (CPU) and Memory Blocks has been written and integrated. The binary code which has to be executed by the CPU has been written into Read Only Memory (ROM) and output has been verified using Xilinx ISE Simulator (ISim). Binary codes for addition and subtraction of two numbers and storing the result into Random Access Memory (RAM) was written and verified with manually calculated result. Using X-HDL software the Verilog HDL codes were converted to VHDL codes which is the supported format for Alliance VLSI CAD tool which has terminal interface and runs on Linux Operating System.

Layouts for some of the blocks are built and we have optimized the delay path and also the area. Both delay and area optimization is carried out by giving an intermediate value of '2' varying in the range '0 to 4' in boolean minimization command. The standard cells are then placed and routed in the Alliance VLSI CAD tool. Netlist extractor tool in Alliance is used to extract netlist from the symbolic layout. It also extracts parasitic capacitance and resistance. The resulting netlist is the list of interconnections at current layout hierarchy level. The symbolic layout obtained is then converted into a real technology layout in 'cif' or 'gds' format. DRUC (Design RUle Checker) is the Alliance parameterized VLSI design rule checker. DRUC flattens all the hierarchy of the design to check if there is any violation to the design rules. Finally, we use the Design Rule Checker to verify the output generated and also to detect any design rule violation.

The processor built can be used to run games and simple applications on mini Operating System.

6.2 Future Scope

The processor that is built is a 16-bit processor. The number of bits can be increased. The number of address line also can be increased so that we can add more complex operations which would reduce the work in implementing the software for this processor. The Program Status Word like flags can be added which would increase the functionality of the processor by giving more facility of branching and conditional operations.

Our Processor can also be made to pipeline the instructions which would make it faster and perform more complex and tedious operations in efficient and faster methods. Multicore processor can handle multiple operations at the same time. Our processor can also be optimized in area and delay during implementation and drawing the layout. ROM can be made to store mini Operating Systems along with provision to use keyboard and screen which will give user better features and this will make our processor user friendly.

Appendix A

Certificates and Paper Presented

The Institution Of Engineers (India) Student's Chapter

NMAM INSTITUTE OF TECHNOLOGY

(An Autonomous Institution affiliated to VTU, Belagavi)
(A unit of Nitte Education Trust)
NITTE - 574110, KARKALA TALUK, KARNATAKA

STATE LEVEL TECHNICAL PAPER PRESENTATION
AND PROJECT MODEL EXHIBITION
"IENITTE-2016"

ON

EMERGING AND SUSTAINABLE TECHNOLOGY FOR SUCCESSFUL ACCOMPLISHMENT OF NATIONAL PROGRAMME "MAKE IN INDIA"

This is to certify that Mr/Ms. KISHAN MUCHINNAY K

of NMAMIT, NITTE

has presented/~~exhibited~~ a paper/~~model~~ entitled Hadware Implementation of a Prouser

at **IENITTE-2016** *held on 5th March, 2016 at NMAM Institute of Technology, Nitte.*

Dr. Narasimha Marakala
Staff Coordinator

Dr. Niranjan N Chiplunkar
Principal

The Institution Of Engineers (India) Student's Chapter

NMAM INSTITUTE OF TECHNOLOGY

(An Autonomous Institution affiliated to VTU, Belagavi)
(A unit of Nitte Education Trust)
NITTE - 574110, KARKALA TALUK, KARNATAKA

STATE LEVEL TECHNICAL PAPER PRESENTATION
AND PROJECT MODEL EXHIBITION
"IEINITTE-2016"

ON

EMERGING AND SUSTAINABLE TECHNOLOGY FOR SUCCESSFUL ACCOMPLISHMENT OF NATIONAL PROGRAMME "MAKE IN INDIA"

This is to certify that Mr/Ms NIKITH MK

of NMAMIT, Nitte

has presented/~~exhibited~~ a paper/~~model~~ entitled Hadware Implementation of a Prcessor

at **IEINITTE-2016** *held on 5th March, 2016 at NMAM Institute of Technology, Nitte.*

Dr. Narasimha Marakala
Staff Coordinator

Dr. Niranjan N. Chiplunkar
Principal

The Institution Of Engineers (India) Student's Chapter

NMAM INSTITUTE OF TECHNOLOGY

(An Autonomous Institution affiliated to VTU, Belagavi)
(A unit of Nitte Education Trust)
NITTE - 574110, KARKALA TALUK, KARNATAKA

STATE LEVEL TECHNICAL PAPER PRESENTATION
AND PROJECT MODEL EXHIBITION
"IEINITTE-2016"

ON

EMERGING AND SUSTAINABLE TECHNOLOGY FOR SUCCESSFUL ACCOMPLISHMENT OF NATIONAL PROGRAMME "MAKE IN INDIA"

This is to certify that Mr/Ms. NIHAL P SHETTY

of NMAMIT, Nitte

has presented/~~exhibited~~ a paper/~~model~~ entitled Hardware Implementation of a Processor

at **IEINITTE-2016** *held on 5th March, 2016 at NMAM Institute of Technology, Nitte.*

Dr. Narasimha Marakala
Staff Coordinator

Dr. Niranjan N. Chiplunkar
Principal

The Institution Of Engineers (India) Student's Chapter

NMAM INSTITUTE OF TECHNOLOGY

(An Autonomous Institution affiliated to VTU, Belagavi)
(A unit of Nitte Education Trust)
NITTE - 574110, KARKALA TALUK, KARNATAKA

STATE LEVEL TECHNICAL PAPER PRESENTATION
AND PROJECT MODEL EXHIBITION
"IEINITTE-2016"

ON

EMERGING AND SUSTAINABLE TECHNOLOGY FOR SUCCESSFUL ACCOMPLISHMENT OF NATIONAL PROGRAMME "MAKE IN INDIA"

This is to certify that Mr/Ms. PRAJWAL SHARMA K

of NMAMIT, Nitte

has presented/exhibited a paper/~~model~~ entitled Hardware Implementation of a Processor

at **IEINITTE-2016** *held on 5th March, 2016 at NMAM Institute of Technology, Nitte.*

Dr. Narasimha Marakala
Staff Coordinator

Dr. Niranjan N. Chiplunkar
Principal

HARDWARE IMPLEMENTATION OF A PROCESSOR

Prajwal Sharma K[1], Kishan Muchinnaya[2], Nihal P Shetty[3], Nikhith M K[4], Mahaveera K[5]
[1,2,3,4] *Student,* [5] *Asst. Prof., Department of Electronics and Communication, NMAMIT, Nitte.*
[1]*prajwalsharmakote@gmail.com,* [2]*kishan.muchinnaya@gmail.com,* [3]*nihalshetty1994@gmail.com,*
[4]*nikhithmk@gmail.com,* [5]*mahaveera_nmamit@nitte.edu.in*

Abstract: Processor design is one of the important steps in any hardware design. Any processor is made up of millions of transistors and design is complex. The proposed processor is a 16 bit processor and designed based on Harvard Architecture i.e. it has separate Data memory and Instruction memory. The prominent part of the processor is Arithmetic Logic Unit (ALU) and it can perform 18 operations. The design is modeled using Verilog HDL. The processor designed is a non-pipelining processor. It uses sequential and combinational circuits as its building blocks. All the building blocks are built using NAND gates which is a universal gate.
The Instruction Set Architecture (ISA) is divided into two instructions. Firstly, data instruction which when decoded controls the ALU operation and data movements. Second instruction is address instruction which specifies the address where data has to be stored.
Key words: Processor, ALU, Harvard Architecture, NAND, Verilog HDL.

Introduction:

All the computation and storage in a computer is done using a Processor of required specification. The Processor consists of Central Processing Unit (CPU), Random Access Memory (RAM) and Read Only Memory (ROM). The CPU performs arithmetic and logical operations and hence called the brain of the computer. The CPU also gives the address as output to fetch the data from the memory.
The project involves design of a 16-bit processor which is capable of doing eighteen operations. The processor has an addressable 24K, 16-bit Data Memory which is also called as RAM. It also has addressable 32K, 16-bit Instruction Memory which is also called ROM.
To design the processor the Instruction Set is divided into two sets: one is Address Instruction and other is Data Instruction. Using the Data Instruction the operations to be performed by the Arithmetic Logic Unit (ALU) can be specified. The proposed processor is built using NAND gates.

Processor Block:

Instruction Memory:

Instruction Memory is implemented using direct access Read Only Memory (ROM). This ROM consists of 32K addressable 16-bit registers. The 32K ROM can be accessed using the Program Counter (PC) which is one of the outputs of Central Processing Unit (CPU). The instruction is fetched from the instruction memory according to the PC specified by the CPU.

Data Memory:

Data Memory chip has the interface of a typical Random Access Memory (RAM). To read the contents of this memory, 'address' (one of the output of CPU) which is the address of the register from which data has to be read is given as an input. The contents read from the memory is 'in' which is also one of the inputs to the CPU as shown in Fig.1 .To write the data into the Data Memory 'write' signal has to be activated and the address in which the data has to be stored should be given as the input. RAM consists of 24K addressable 16-bit registers.

Central Processing Unit:

Central Processing Unit (CPU) consists of Arithmetic Logic Unit (ALU) which performs various Arithmetic and Logical Operations. The operation to be performed by the CPU is selected using the decoded instruction. The CPU output (out) can write into RAM or can be used as an input data for the

next instruction. The instruction fetched from the Instruction Memory is decoded using a decoder. The instruction to be fetched is the output of Program Counter (PC).

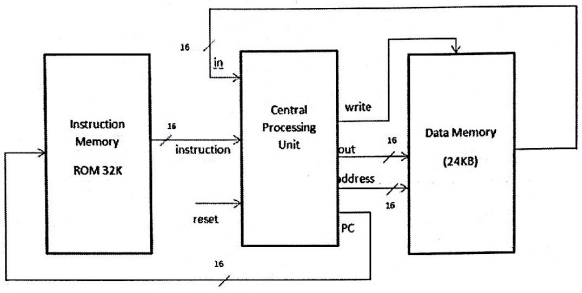

Fig.1. Block diagram of the proposed processor.

Central Processing Unit Block:
In the Central Processing Unit (CPU) the instruction (Address Instruction or Data Instruction) is decoded using a decoder. The contents of A-Register depend on the output of the Multiplexer which can be either the data from previous output of the ALU or an instruction depending on whether the instruction is Data Instruction or Address Instruction. The D-Register is used to provide previous output of ALU as one of the input for computation. The two output control bits (zr, ng) from the ALU act as input control bits to the Program Counter (PC). Multiplexer with 'a' as control bit is used to select between input from RAM and the input from A-Register. The block diagram of the CPU is shown in Fig.2.

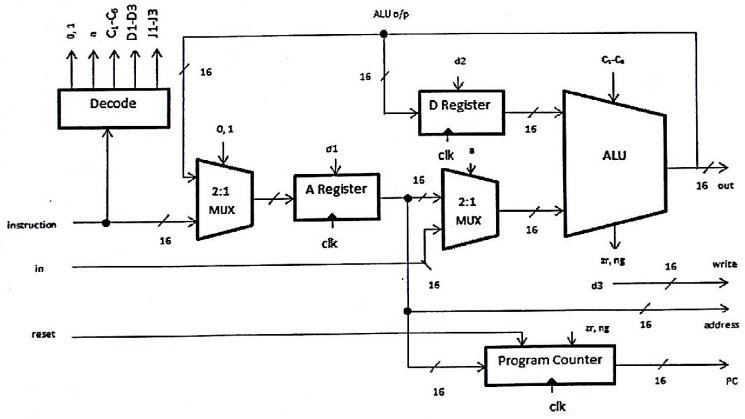

Fig.2. Block diagram of the Central Processing unit (CPU).

Arithmetic Logic Unit:
Arithmetic Logic Unit (ALU) performs Arithmetic and Logical operations and the proposed ALU is designed to perform eighteen operations. The operations performed by the ALU depends on the control bits (C1-C6) obtained by decoding the Data Instruction. Different operations performed by the ALU are shown in the Table 1.

Table 1 Computation Specifications.

when a=0 comp mnemonic	C1	C2	C3	C4	C5	C6	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

Program Counter:
Program counter is used to fetch instruction from Instruction Memory. When reset is set to '1' the output of PC is 0000H. The input control bits are used to increment the PC and are also used to load the PC with new value of address from which instruction has to be fetched.

Instruction Set Architecture:
Instruction Set Architecture of the proposed processor consists of two instructions, namely Address instruction and Data Instruction.

Address Instruction:
The format of Address Instruction is
0vvv vvvv vvvv vvvv
Where v's refer to the address where data has to be stored in the Data Memory. v can take value either 0 or 1.

Data Instruction:
The format of the Data Instruction is
111a C1C2C3C4 C5C6D1D2 D3J1J2J3
Where C1-C6 refers to compare computation, D1-D3 refers to destination where data has to be stored and J1-J3 refers to Jump operations. Destination and Jump specifications are shown in the Table 2 and Table 3 respectively.

Table 2 Destination specifications.

D1	D2	D3	Mnemonic	Destination
0	0	0	Null	Value is not stored
0	0	1	M	Memory A
0	1	0	D	D register
0	1	1	MD	Memory A and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory A
1	1	0	AD	A register and D register
1	1	1	AMD	A register, memory A and D register

Table 3 Jump specifications

J1 Out < 0	J2 Out =0	J3 Out >0	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If out >0 jump
0	1	0	JEQ	If out =0 jump
0	1	1	JGE	If out >=0 jump
1	0	0	JLT	If out <0 jump
1	0	1	JNE	If out != 0 jump
1	1	0	JLE	If out <= 0 jump
1	1	1	JMP	Jump

Conclusion:

Processor design is one of the important step in any hardware design. Any processor is made up of millions of transistors and design is complex. NAND gate is a universal gate which can be used in all digital designs. Combinational and Sequential blocks can be built using NAND gates. The Combinational Logic Circuits and Sequential Logic Circuits form the basic building of Arithmetic Logic Unit and Memory Unit respectively. It can be inferred that the Processor can be built entirely using NAND gates. This is an attempt to show that the complex Processor design can be simplified and designed using the simple NAND gates.

Reference:

***Journals / Books*:**

[1] J. L. Hennessy, D. A. Patterson, Computer Architecture A Quantitative Approach, Second Edition, Morgan Kaufmann and Harcourt India, 2000.

[2] J. F. Wakerly, Digital Design: Principles and Practices, Third Edition, Prentice- Hall, 2000.

[3] A. S. Tanenbaum, Structured Computer Organization, Fourth Edition, Prentice- Hall, 2000.

[4] Morris Mano, Michael D. Ciletti, Digital Design, Fifth Edition, 2013.

[5] Cheong-Ghil Kim, Jung-Hoon Lee, Instruction and Data NAND Flash Memory System for Embedded Systems, Information Science and Applications (ICISA), IEEE, pp. 1-6, May 2012.

References

[1] J. L. Hennessy, D. A. Patterson, “Computer Architecture A Quantitative Approach”, Second Edition, Morgan Kaufmann and Harcourt India, 2000.

[2] J. F. Wakerly, “Digital Design: Principles and Practices”, Third Edition, Prentice-Hall, 2000.

[3] A. S. Tanenbaum, “Structured Computer Organization”, Fourth Edition, Prentice-Hall, 2000.

[4] Morris Mano, Michael D. Ciletti, “Digital Design”, Fifth Edition, 2013.

[5] Cheong-Ghil Kim, Jung-Hoon Lee, “Instruction and Data NAND Flash Memory System for Embedded Systems”, Information Science and Applications (ICISA), IEEE, pp. 1-6, May 2012.

[6] Carlos Silva Cardenas, Takeo Yashida, “Introduction to VLSI CMOS Circuits Design”, Toin University of Yokohama, 2006.

[7] VLSI and ASIC Technology Standard Cell Library Design, 2008. www.vlsitechnology.org [Accessed on 04 April 2016].