IMPLEMENTATION OF 8-BIT RISC PROCESSOR

A Project report submitted in partial fulfillment of the requirements for

the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

S.Naveen Kumar (316126512170)

G.Vasanthi (316126512196)

P.Tirumala(316126512212)

B.Leela Madhav (316126512214)

Under the guidance of

Mrs. P S M Veena

(Ph.D), Asst.Prof

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(*Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'A' Grade*)

Sangivalasa, bheemili mandal, visakhapatnam dist.(A.P)

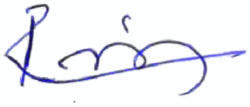
2019-2020

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES
(*Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'A' Grade*)
Sangivalasa, bheemili mandal, visakhapatnam dist.(A.P)

CERTIFICATE

This is to certify that the project report entitled **"Implementation of 8-Bit RISC Processor"** submitted by **S.Naveen Kumar (316126512170), G.Vasanthi (316126512196), P.Tirumala (316126512212), B.Leela Madhav (316126512214)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Engineering** in **Electronics & Communication Engineering** of Andhra University, Visakhapatnam is a record of bonafide work carried out under my guidance and supervision.



Project Guide

Mrs. P S M Veena
(Ph.D),Assistant Professor
Department of ECE
ANITS

Head of the Department

Dr. V.Rajyalakshmi
M.E,Ph.D,MHRM,MIEEE,MIE,MIETE
Department of ECE
ANITS

ACKNOWLEDGEMENT

We would like to express our deep gratitude to our project guide **Mrs. P S M Veena** Assistant Professor, Department of Electronics and Communication Engineering, ANITS, for her guidance with unsurpassed knowledge and immense encouragement. We are grateful to **Dr. V. Rajyalakshmi**, Head of the Department, Electronics and Communication Engineering, for providing us with the required facilities for the completion of the project work.

We are very much thankful to the **Principal and Management, ANITS, Sangivalasa,** for their encouragement and cooperation to carry out this work.

We express our thanks to all **teaching faculty** of Department of ECE, whose suggestions during reviews helped us in accomplishment of our project. We would like to thank **all non-teaching staff** of the Department of ECE, ANITS for providing great assistance in accomplishment of our project.

We would like to thank our parents, friends, and classmates for their encouragement throughout our project period. At last but not the least, we thank everyone for supporting us directly or indirectly in completing this project successfully.

PROJECT STUDENTS

S.Naveen Kumar(316126512170),
G.Vasanthi(316126512196),
P.Tirumala(316126512212),
B.Leela Madhav(316126512214).

ABSTRACT

A processor is a small chip that resides in computers and other electronic devices. Its basic job is to receive input and provide the appropriate output. While this may seem like a simple task, modern processors can handle trillions of calculations per second. Modern CPUs often include multiple processing cores, which work together to process instructions. While these "cores" are contained in one physical unit, they are actually individual processors. This project is going to design & simulate a 8-bit Microprocessor which is near to Reduced Instruction Set Computing (RISC) architecture based processor. The purpose of RISC microprocessor is to execute a minuscule batch of instructions. The designing process is using modules like ALU, Control Unit, Program Counter, MUX, Memory, Register File by using the Verilog Hardware Description Language (HDL).

Microprocessor is a programmable, multipurpose electronic device that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions and provides results as output. It executes the sequence of instructions one after the other.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it. Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device.

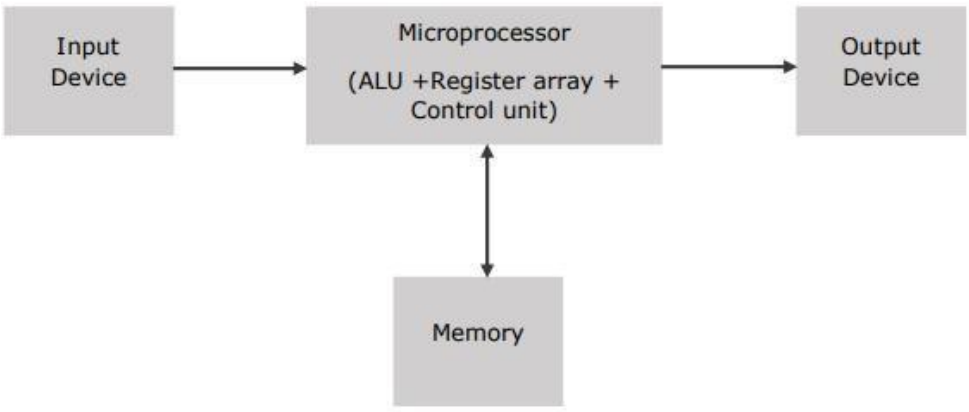


Figure 1.1 Basic Block diagram of Micro-computer

1.1 Microprocessor

The microprocessor follows a sequence: Fetch, Decode, and then Execute. Initially, the instructions are stored in the memory in a sequential order. The microprocessor fetches those instructions from the memory, then decodes it and executes those instructions till STOP instruction is reached. Later, it sends the result in binary to the output port. Between these processes, the register stores the temporarily data and ALU performs the computing functions.

Microprocessor is a programmable, multipurpose electronic device that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions and provides results as output. Microprocessors & Microcontrollers are generally designed in the vicinity of two

main computer architectures: Complex Instruction Set Computing i.e. CISC architecture and Reduced Instruction Set Computing i.e. RISC architecture. The concept of CISC is based on Instruction Set Architecture (ISA) design that redoubles performing further with several instructions utilizing changeable number of operands and an out spread variation of addressing modes in disparate locations in its Instruction Set. Thus causing them to have varying execution time and lengths thereby authoritatively mandating an intricate Control Unit, which inhabits an immensely existent region on the chip.

Compared with their CISC analogue, RISC processors typically support a minuscule set of instructions. A display that just a poses RISC processor with CISC processor, the number of instructions in a RISC Processor is low while the number of general purpose registers, addressing modes fixed instruction length and load-store architecture is more this in turn facilitates the execution of instructions to be carried out in a short time thus achieving higher overall performance.

Currently, the efficiency of the RISC processors is generally accepted to be greater than that of their CISC counterparts. Before their execution the instructions are translated into RISC instructions in even the most popular CISC processors. The attributes mentioned above accentuate the design strength of RISC in the market for embedded systems known as "system-on-a-chip (SoC)". The premier microprocessors exhibiting reduced instruction set are SPARC, ARM, MIPS and IBM's PowerPC. RISC processor typically has load store architecture. This denotes there are two instructions for accessing memory which are a load instruction set to load data from the memory and store instruction set to Write Back (WB) the data into memory.

1.2 Characteristics of RISC

The major characteristics of a RISC processor are as follows:

- It consists of simple instructions.
- It supports various data-type formats.
- It utilizes simple addressing modes and fixed length instructions for pipelining.
- It supports register to use in any context.

- One cycle execution time.
- “LOAD” and “STORE” instructions are used to access the memory location.
- It consists of larger number of registers.
- It consists of less number of transistors.

Nowadays, computers perform required calculations for almost all electronic devices in the market. Computer hardware is build based upon binary operations to satisfy a set of mathematical, data transfer, and control instructions. The basic instructions realized directly by hardware are known as the processor instruction set. It is used for the creation of structured software programs for data manipulation in problem solving. In this work, an 8-bit RISC processor design strategy that uses simplified instructions for higher performance with faster execution of instruction. It also reduces the delay in execution. This processor comprises of Control unit, general purpose registers, Arithmetic and logical unit, shift registers. In this processor control unit follows instruction cycle of 3 stages fetch, decode and execute cycle. According to the instruction to the fetch stage, control unit generate signal to decode the instruction.

This architecture supports instructions of arithmetic, logical, shifting and branch operations. This architecture principle Reduced Instruction Set Computer is commonly known as RISC. RISC processors allow special load and store operations to access memory. The other operations are performed on register-to-register basis. This feature makes instruction set more clear and simple as it allows execution of instructions at one-instruction-per-cycle rate. 8-bit computers are used extensively as controllers for simple computational tasks. According to the divide and conquer principle, a common personal computer is divided into smaller ones (commonly 8-bit) which share information with the main computer (32 or 64-bit). An 8-bit processor generally handles the drivers for almost every component card inside a computer

In this we also designed an assembler (which is a program) that converts assembly language into machine code. It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor.This is written in Python which is an interpreted, high-level, general-purpose programming language.

Assemblers are similar to compilers in that they produce executable code. However, assemblers are more simplistic since they only convert low-level code (assembly language) to machine code. Since each assembly language is designed for a specific processor, assembling a program is performed using a simple one-to-one mapping from assembly code to machine code. Compilers, on the other hand, must convert generic high-level source code into machine code for a specific processor.

In this Project we used Iverilog for simulating Verilog Code. *Icarus Verilog* is a Verilog simulation and synthesis tool. It operates as a compiler, compiling source code written in Verilog (IEEE-1364) into some target format. For batch simulation, the compiler can generate an intermediate form called *vvp assembly*. This intermediate form is executed by the ``vvp'' command.

For Graphical analysis we used Vivado Design Suite which is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis. It is also used to program FPGAs.

In order to write Verilog Code with Syntax highlighting we used Visual Studio Code which is a source-code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control and Git Hub, syntax highlighting, intelligent code completion, snippets, and code refactoring.

For tracking changes in source code we use Git which is distributed version-control system. It is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

CHAPTER 2

PROCESSOR

2.1 Architecture

We have tried to depict the architecture of processor with this following image**:-**

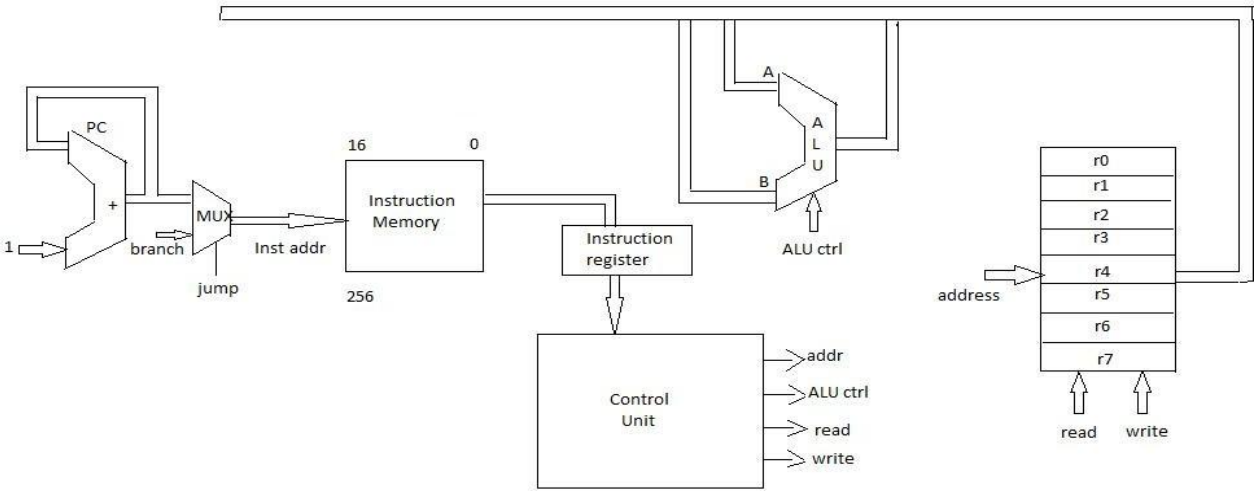


Figure 2.1 Architecture of 8-bit RISC Processor

2.2 Microprocessor – Functional Units

It consists of the following functional units –

Arithmetic and logic unit

As the name suggests, it performs arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8-bit data.

General purpose register

There are 8 general purpose registers in this processor, i.e. R0, R1, R2, R3...R7. Each register can hold 8-bit data.

Program counter

It is an 8-bit register used to store the memory address location of the next instruction to be executed. Microprocessor increments the program whenever an instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed.

Temporary registers

These are 8-bit register A and B, which holds the temporary data of arithmetic and logical operations.

Instruction register and decoder

It is a 16-bit register. When an instruction is fetched from memory then it is stored in the Instruction register. Instruction decoder in Control Unit decodes the information present in the Instruction register.

Timing and control unit

It provides timing and control signal to the microprocessor to perform operations. Following are the timing and control signals, which control external and internal circuits − Control Signals: READ, WRITE, ALU Ctrl, Addr

Address bus and data bus

Data bus carries the data to be stored. It is bidirectional, whereas address bus carries the location to where it should be stored and it is unidirectional. It is used to transfer the data & Address I/O devices.

2.3 Addressing Modes

These are the instructions used to transfer the data from one register to another register, from the memory to the register, and from the register to the memory without any alteration in the content. Addressing modes in this processor is classified into 5 groups –

Immediate addressing mode

In this mode, the 8-bit data is specified in the instruction itself as one of its operands.

For example: LDI R0, 01: means 01 is copied into register R0.

Register addressing mode

In this mode, the data is copied from one register to another.

For example: MOV R0, R1: means data in register R1 is copied to register R0.

Direct addressing mode

In this mode, the data is directly copied from the given address to the register.

For example: LDM R0 05: means the data at address 05 is copied to register R0.

2.4 Instruction Sets

Instruction sets are instruction codes to perform some task. It is classified into five categories.

2.4.1 Arithmetic Instructions

ADD RegDestination RegSrc1 RegSrc2

The contents of the source register 1 and 2 gets added and the result is stored in the Destination Register.

Example − ADD R0 R1 R2.

Here the contents of the source register R1 and R2 gets added and the result is stored in the Destination Register R0 as Specified in the Instruction itself.

SUB RegDestination RegSrc1 RegSrc2

The contents of the source register 1 and 2 gets added and the result is stored in the Destination Register.

Example − ADD R0 R1 R2.

Here the contents of the source register R2 are subtracted from the contents of the source register R1 and the result is stored in the Destination Register R0 as Specified in the Instruction itself.

INC Reg

The contents of the designated register is incremented by 1 and their result is stored at the same place.

Example − INR R0

Here the contents of the designated register R0 is incremented by 1 and their result is stored at the same place R0 itself.

DEC Reg

The contents of the designated register is decremented by 1 and their result is stored at the same place.

Example − DEC R0

Here the contents of the designated register R0 is decremented by 1 and their result is stored at the same place R0 itself.

2.4.2 Logical Instructions

AND RegDestination RegSrc1 RegSrc2

The contents of the source register 1 and 2 gets logically ANDed and the result is stored in the Destination Register.

Example − AND R0 R1 R2.

Here the contents of the source register R1 and R2 gets ANDed and the result is stored in the Destination Register R0 as Specified in the Instruction.

OR RegDestination RegSrc1 RegSrc2

The contents of the source register 1 and 2 gets logically ORed and the result is stored in the Destination Register.

Example − OR R0 R1 R2.

Here the contents of the source register R1 and R2 gets ORed and the result is stored in the Destination Register R0 as Specified in the Instruction.

XOR RegDestination RegSrc1 RegSrc2

The contents of the source register 1 and 2 gets logically XORed and the result is stored in the Destination Register.

Example − XOR R0 R1 R2.

Here the contents of the source register R1 and R2 gets XORed and the result is stored in the Destination Register R0 as Specified in the Instruction.

INV Reg

The contents of the designated register is inverted and their result is stored at the same place.

Example − INV R0

Here the contents of the designated register R0 is inverted and their result is stored at the same place R0 itself.

SHL Reg RegNoOfTimes

The SHL instruction is an abbreviation for 'Shift Left'. This instruction simply shifts the mentioned bits in the register to the left side one by one by inserting the same number (bits that are being shifted) of zeroes from the right end. The leftmost bit that is being shifted is stored in the Carry Flag (CF).

Example − SHL R0 R1

This instruction simply shifts the mentioned bits in the register R0 to the left side by number of times specified in R1.

SHR Reg RegNoOfTimes

The SHR instruction stands for 'Shift Arithmetic Right'. This instruction shifts the mentioned bits in the register to the right side one by one inserting the same number

(bits that are being shifted) of zeroes from the left end. The rightmost bit that is being shifted is stored in the Carry Flag (CF).

Example − SHR R0 R1

This instruction simply shifts the mentioned bits in the register R0 to the right side by number of times specified in R1.

2.4.3 Data-transfer Instructions

LDI Reg Data

The 8-bit data is loaded into the destination register specified in the instruction.

Example – LDI R0 01

Here register R0 is loaded with the value 01 as specified in the instruction

MOV Rd Rs

This instruction copies the contents of the source register into the destination register without any alteration.

Example − MOV R0, R1

Here the contents of the source register R1is copied into the destination register R0 without any alteration

LDM Reg addr

The contents of a memory location, specified by an 8-bit address in the operand, are copied to the register specified.

Example − LDM R0 08

Here the contents of a memory location 08, specified by an 8-bit address in the operand, are copied to the register R0 specified in the instruction.

STM Reg addr

The contents of register are copied to the memory location, specified by an 8-bit address in the operand.

Example − LDM R0 08

Here the contents of register R0 are copied to the memory location 08, specified by an 8-bit address in the operand.

2.4.4 Branching Instructions

JNZ Reg addr

The program sequence is transferred to the memory address given in the operand if the content of register is not zero.

Example − JNZ R0 06

Here the program sequence is transferred to the memory address 06 given in the operand if the content of register is not zero

JMP addr

The program sequence is transferred to the memory address given in the operand if this instruction is encountered.

Example − JMP 06

Here the program sequence is transferred to the memory address 06 given in the operand if this instruction is encountered in the program.

2.4.5 Control Instructions

HLT

The CPU finishes executing the current instruction and stops further execution. Reset is necessary to exit from the halt state.

2.5 Instruction Decoding

The decoding process allows the CPU to determine what operation is to be performed so that the processor can tell how many operands it needs to fetch in order to perform the operation. The instruction fetched from the memory is decoded for the next steps and moved to the appropriate registers.

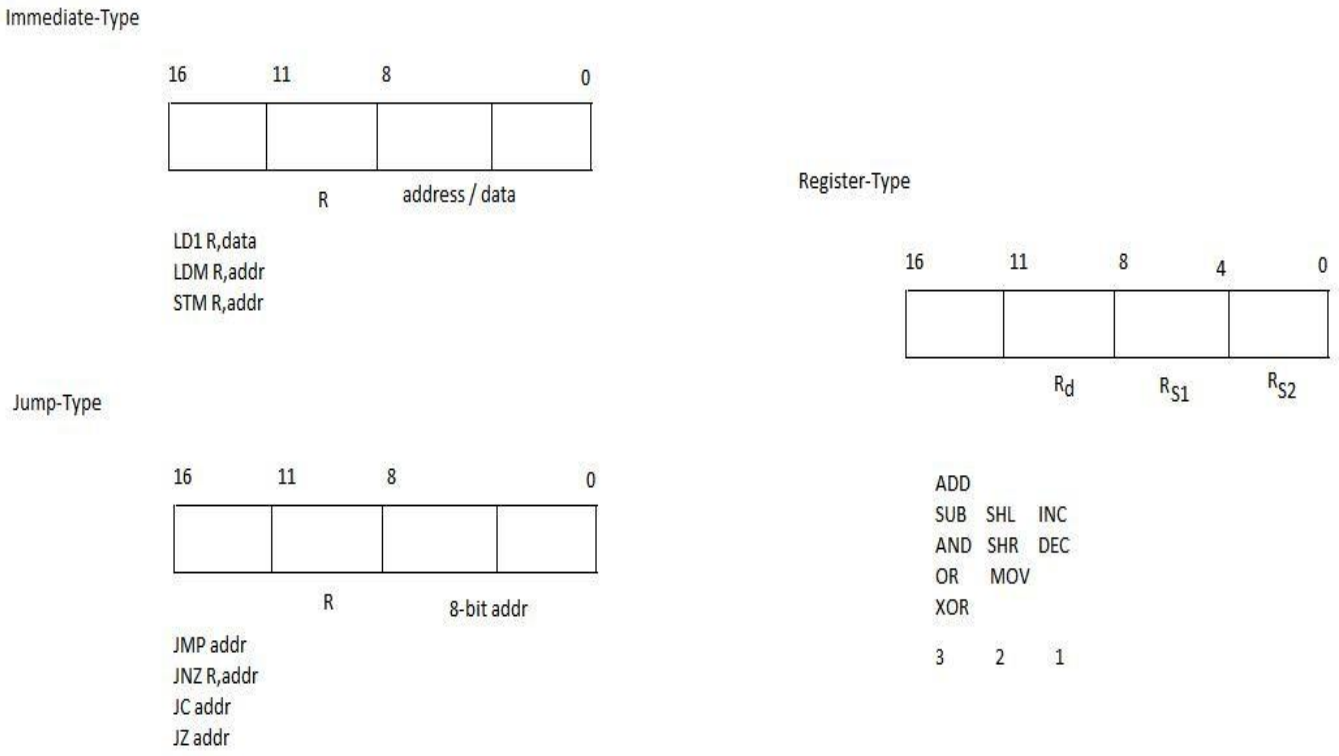


Figure 2.2 Instruction Decode

2.5.1 Immediate Type Decoding

In Immediate Type Decoding of instruction lower 8 bit specifies address or data and the bits 11 to 15 specifies opcode, if there is any register to specify it is done using the remaining three bits.

Example-

LDI R1 08

01000_001_00001000

From the example shown 01000 specifies opcode LDI 001 specifies R1 and 00001000 specifies 8-bit data.

2.5.2 Register Type Decoding

In Register Type Decoding of instruction lower 8 bit specifies two destination registers and the bits 11 to 15 specifies opcode and destination register is specified

using the remaining three bits. Here two bits are unused in lower 8-bits and can be used to increase destination registers up to 16 in future.

Example-

ADD R0 R1 R2

00000_000_**0**001_**0**010

From the example shown 00000 specifies opcode ADD 000 specifies R0 and 001 specifies R1 and 010 specifies R2 and remaining two bits are unused(Bold).

SHR R0 R1

10010_000_**0**001_**0000**

From the example shown 10010 specifies opcode SHR 000 specifies R0 and 001 specifies R1 and remaining lower 4 bits are unused (Bold).

INC R0

01010_000_**0000_0000**

From the example shown 01010 specifies opcode INC 000 specifies R0 and and remaining lower 8 bits are unused(Bold).

2.5.3 Jump Type Decoding

In Jump Type Decoding of instruction lower 8 bit specifies address and the bits 11 to 15 specifies opcode, if there is any register to specify it is done using the remaining three bits

Example-

JMP addr

01111_**000**_0000_0010

From the example shown 01111 specifies opcode JMP 0000_0010 specifies address and remaining 3 bits are unused (Bold).

JNZ R1,addr

01110_001_0000_0011

From the example shown 01110 specifies opcode JMP 0000_0011 specifies address and remaining 3 bits 001 specifies R1.

2.6 Assembler

This processor has an assembler written in python in order to reduce the burden of writing machine codes. An assembler is a program that converts assembly language into machine code. It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor.

Assemblers are similar to compilers in that they produce executable code. However, assemblers are more simplistic since they only convert low-level code (assembly language) to machine code. Since each assembly language is designed for a specific processor, assembling a program is performed using a simple one-to-one mapping from assembly code to machine code. Compilers, on the other hand, must convert generic high-level source code into machine code for a specific processor.

Most programs are written in high-level programming languages and are compiled directly to machine code using a compiler. However, in some cases, assembly code may be used to customize functions and ensure they perform in a specific way. Therefore, IDEs often include assemblers so they can build programs from both high and low-level languages.

For example:

ADD R0 R0 R1 -----→ Python---- → 00000_000_0000_0001

LDI R1 05 -----→ Python ----→00100_001_0000_0101

2.7 Instructions

00000_000_0000_0001 // ADD R0<=R0+R1

00001_001_0000_0001 // SUB R1<=R0-R1

00010_010_0000_0001 // AND R2<=R0&R1

00011_010_0000_0001 // OR R2<=R0|R1

00100_010_0000_0001 // XOR R2<=R0^R1

00101_000_0000_0001 // INV R0

00110_000_0001_0001 // SHL R0<<R1

00111_001_0000_0000 // MOV R1<=R

00100_001_0000_0101 // LDI R1<=05

01001_000_0000_0000 // JZ label ***

01010_000_0000_0001 // INC R0

01011_000_0000_0001 // DEC R0

01100_001_0000_0001 // HLT

01101_000_0000_0000 // JC label***

01110_001_0000_0011 // JNZ R1,addr

01111_000_0000_0010 // JMP addr

10000_000_0000_1000 // LDM R0

10001_000_0000_1000 // STM R0

10010_000_0001_0000 // SHR R0>>R1

In all the above instructions first five bits specify Opcode and remaining bits are according to instruction decoding as specified above in instruction decoding section.

2.8 Basic programs

2.8.1 Multiplication

1000_0000_0000_1000 // LDI R0<=8 00

1000_0001_0000_0111 // LDI R1<=7 01

1000_0010_0000_0000 // LDI R2<=0 10

1011_0001_0000_0001 // DEC R1 11

0000_0010_0000_0010 // ADD R2<=R0+R2

1110_0001_0000_0011 // JNZ R1,abvline

1100_0001_0000_0001 // HLT

2.8.2 DIVISON

1000_0000_0100_0010 // LDI R0<=10 00

1000_0001_0000_0011 // LDI R1<=2 01

0111_0011_0000_0000 // MOV R3<=R0 10

1000_0010_0000_0000 // LDI R2<=0 11

1010_0010_0000_0001 // INC R2 100

0001_0011_0011_0001 // SUB R3<=R3-R1 101

1101_0000_0000_1000 // JC 1000 110

1111_0000_0000_0100 // JMP 100 111

1011_0010_0000_0001 // DEC R2 1000

0000_0011_0011_0001 // ADD R3<=R3+R1 1001

1100_0001_0000_0001 // HLT 1010

2.8.3 Memory Operations

01000_000_0000_1111 // LDI R0<=05

10000_000_0000_0110 // LDM R0

10000_001_0000_0111 // LDM R1

00000_010_0000_0001 // ADD R2<=R0+R1

10001_010_0000_1000 // STM R0

01100_001_0000_0001 // HLT

01100_001_0000_0011 //data1 110

01100_001_0000_0111 //data2 111

//store 1000

CHAPTER 3

VERILOG HARDWARE DESCRIPTION LANGUAGE

3.1 Hardware Description Language

Hardware description language (HDL) is a specialized computer language used to program electronic and digital logic circuits. The structure, operation and design of the circuits are programmable using HDL. HDL includes a textual description consisting of operators, expressions, statements, inputs and outputs. Instead of generating a computer executable file, the HDL compilers provide a gate map. The gate map obtained is then downloaded to the programming device to check the operations of the desired circuit. The language helps to describe any digital circuit in the form of structural, behavioural and gate level and it is found to be an excellent programming language for FPGAs and CPLDs. The three common HDLs are Verilog, VHDL, and System C.

3.2 Importance of HDLs

HDLs have many advantages compared to traditional schematic-based design.

• Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.

• By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly

because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.

• Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.

3.3 Introduction to Verilog HDL

Verilog HDL is one of the two most common Hardware Description Languages (HDL) used by integrated circuit (IC) designers. The other one is VHDL. HDL's allows the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are technology-independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

Verilog can be used to describe designs at four levels of abstraction:
(i) Algorithmic level (much like c code with if, case and loop statements).
(ii) Register transfer level (RTL uses registers connected by Boolean equations).
(iii) Gate level (interconnected AND, NOR etc.).
(iv) Switch level (the switches are MOS transistors inside gates).
The language also defines constructs that can be used to control the input and output of simulation.

Verilog has a variety of constructs as part of it. All are aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC. The language has a dual function – one fulfilling the need for a design description and the other fulfilling the need for verifying the design for functionality and timing constraints like propagation delay, critical path delay, slack, setup, and hold times.

Verilog as an HDL has been introduced here and its overall structure explained. A widely used development tool for simulation and synthesis has been introduced; the

brief procedural explanation provided suffices to try out the Examples and Exercises in the text.

3.4 Module

Any Verilog program begins with a keyword– called a "module." A module is the name given to any system considering it as a black box with input and output terminals as shown in Figure 4.1. The terminals of the module are referred to as 'ports'. The ports attached to a module can be of three types:

- **input ports** through which one gets entry into the module; they signify the input signal terminals of the module. x
- **output ports** through which one exits the module; these signify the output signal terminals of the module. X

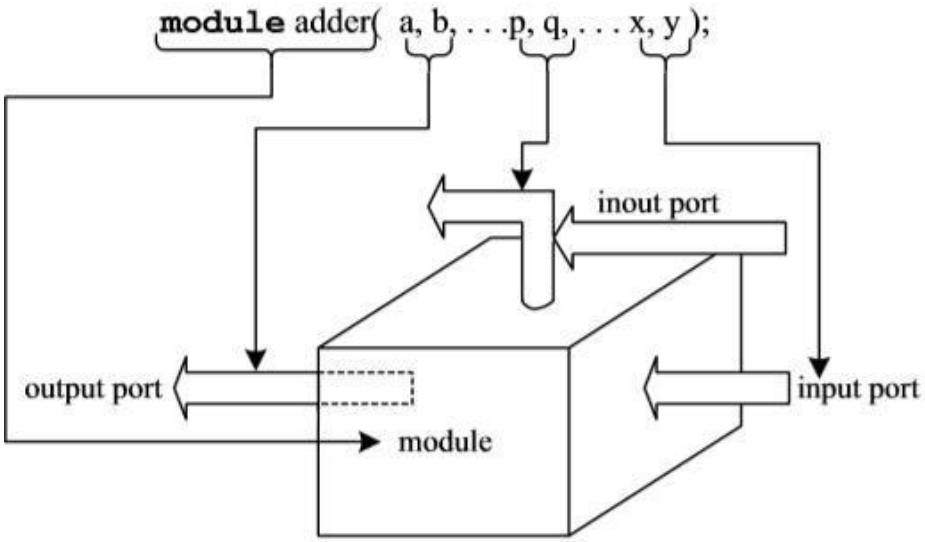


Figure 3.1 Representation of a module as black box with its ports

- **inout ports:** These represent ports through which one gets entry into the module or exits the module; These are terminals through which signals are input to the module sometimes; at some other times signals are output from the module through these.

3.5 Tokens of Verilog

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords.

3.5.1 Case Sensitivity

Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE, sENse,… etc., are all treated as different entities / quantities in Verilog.

3.5.2 Keywords

The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated. As such, a programmer cannot use a keyword for any purpose other than that it is intended for. All keywords in Verilog are in small letters and require to be used as such (since Verilog is a case-sensitive language). All keywords appear in the text in New Courier Bold-type letters.

Examples

Module:- signifies the beginning of a module definition.

End module:- signifies the end of a module definition.

Begin:- signifies the beginning of a block of statements. end<- signifies the end of a block of statements.

If:- signifies a conditional activity to be checked

While:- signifies a conditional activity to be carried out.

3.5.3 Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

Examples

a = ~ b; // ~ is a unary operator. b is the operand

a = b && c; // && is a binary operator. b and c are operands

a = b ? c : d; // ?: is a ternary operator. b, c and d are operands

3.5.4 Data Types

There are two groups of types, "**net data types**" and "**variable data types**."
An identifier of "**net data type**" means that it must be driven. Thevalue changes when the driver changes value. These identifiersbasically represent wires and are used to connect components.
"net data types" are: **wire**, **supply0**, **supply1**, **tri**, **triand**, **trior**,**tri0**, **tri1**, **wand**, **wor**"net data types" can have strength modifiers: **supply0**, **supply1**,**strong0**, **strong1**, **pull0**, **pull1**, **weak0**, **weak1**,**highz0**, **highz1**, **small**, **medium**, **large**.
Some "net data types" can take modifiers: **signed**, **vectored**, scalared.
An identifier of "**variable data type**" means that it changes value uponassignment and holds its value until another assignment. This is atraditional programming language variable and is used in sequential statements.
"variable data types" are: **integer**, **real**, **realtime**, **reg**, **time**.

- **integer** is typically a 32 bit twos complement integer.
- **real** is typically a 64 bit IEEE floating point number.
- **realtime** is of type **real** used for storing time as a floating point value.
- **reg** is by default a one bit unsigned value.
- The **reg** variable data type may have a modifier **signed**, and may have may bits by using the vector modifier [msb:lsb].
- **time** is typically 64 bit unsigned value that is the simulation time. The system function $time provides the simulation time.

Scalars and Vectors

Entities representing single bits — whether the bit is stored, changed, or transferred — are called “scalars.” Often multiple lines carry signals in a cluster –

like data bus, address bus, and so on. Similarly, a group of regs stores a value, which may be assigned, changed, and handled together. The collection here is treated as a "vector." were wr and rd are two scalar nets connecting two circuit blocks circuit1 and circuit2. b is a 4-bit-wide vector net connecting the same two blocks. b[0], b[1], b[2], and b[3] are the individual bits of vector b. They are "part vectors."

A vector reg or net is declared at the outset in a Verilog program and hence treated as such. The range of a vector is specified by a set of 2 digits (or expressions evaluating to a digit) with a colon in between the two. The combination is enclosed within square brackets.

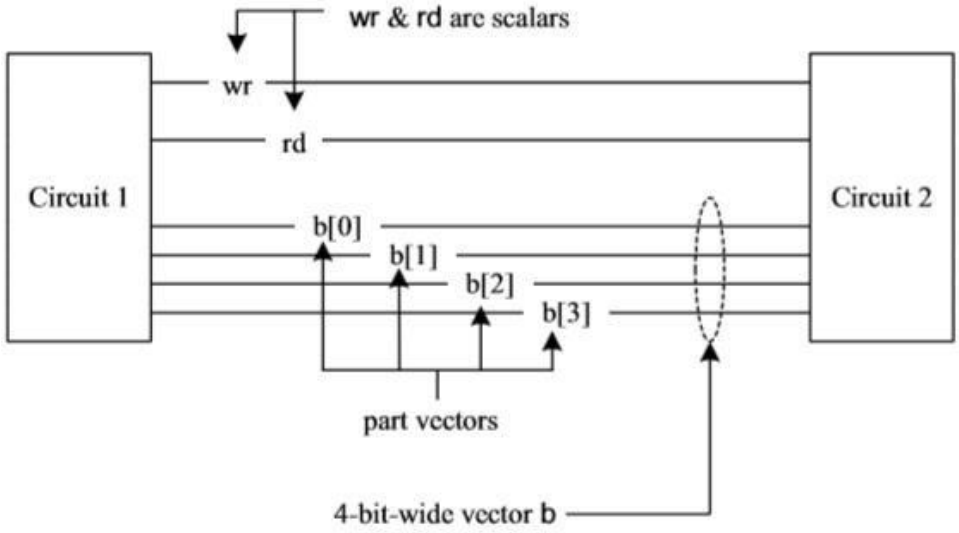


Figure 3.2 Illustration of Scalars and Vectors

Examples:

wire[3:0] a; */* a is a four bit vector of net type; the bits are designated as a[3], a[2], a[1] and a[0]. */* **reg[2:0] b;** */* b is a three bit vector of reg type; the bits are designated as b[2], b[1] and b[0]. */*

reg[4:2] c; */* c is a three bit vector of reg type; the bits are designated as c[4], c[3] and c[2]. */*

wire[-2:2] d; */* d is a 5 bit vector with individual bits designated as d[-2], d[-1], d[0], d[1] and d[2]. */*

Whenever a range is not specified for a net or a reg, the same is treated as a scalar – a single bit quantity. In the range specification of a vector the most significant bit and the least significant bit can be assigned specific integer values. These can also be expressions evaluating to integer constants – positive or negative.Normally vectors – nets or regs – are treated as unsigned quantities. They have to be specifically declared as “signed” if so desired.

Examples

wire signed[4:0] num; *// num is a vector in the range -16 to +15.*

reg signed [3:0] num_1; *// num_1 is a vector in the range -8 to +7.*

3.5.5 Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

a = b && c; // This is a one-line comment

/* This is a multiple line comment */

/* This is /* an illegal */ comment */

/* This is //a legal comment */

3.5.6 Number Specification

There are two types of number specification in Verilog they are sized and unsized.

Sized numbers

Sized numbers are represented as <size> '<base format><number>.

<size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and

octal ('o or 'O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

4'b1111 // This is a 4-bit binary number

12'habc // This is a 12-bit hexadecimal number

16'd255 // This is a 16-bit decimal number.

Un-sized numbers

Numbers that are specified without a <base format> specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

23456 // This is a 32-bit decimal number by default

'hc3 // This is a 32-bit hexadecimal number

'o21 // This is a 32-bit octal number

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

12'h13x // This is a 12-bit hex number; 4 least significant bits unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

An X or Z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is 0, X, or Z, the number is automatically extended to fill the most significant bits, respectively, with 0, X, or Z. This makes it easy to assign X or Z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

-6'd3 // 8-bit negative number stored as 2's complement of 3

-6'sd3 // Used for performing signed integer math

4'd-2 // Illegal specification

3.6 Module Declaration

Modules are the building blocks of Verilog designs. A module can be an element or a collection of lower-level design blocks. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. Module interface refers, how module communicates with external world. This communication is possible through different ports such as input, output and bi-directional (in out) ports. Design functionality is implemented inside module, after port declaration. In Verilog, a module is declared by the keyword module. A corresponding keyword end module must appear at the end of the module definition. Each module must have a module_name, which is the identifier for the module, and a port list, which describes the input and output terminals of the module. Design functionality is implemented inside module, after port declaration. The design functionality implementation part is represented as "body" here.

Syntax

module module_name(port_list);

input[msb:lsb] input_port_list;

output[msb:lsb] output_port_list;

inout[msb:lsb] inout_port_list;

……statements……

End module

NOTE All module declarations must begin with the **module** (or **macro-module**) keyword and end with the **end module** keyword. After the module declaration, an identifier is required. A ports list is an option. After that, ports declaration is given with declarations of the direction of ports and the optionally type. The body of module can be any of the following:

- Any declaration including parameter, function, task, event or any variable declaration.
- Continuous assignment.
- Gate, UDP or module instantiation.
- Specify block.
- Initial block
- Always block.

If there is no instantiation inside the module, it will be treated as a top-level module.

Example

modulemodule_1(a, b, c);

parameter size = 3 ;

input [size : 0] a, b ;

output [size : 0] c;

assign c = a & b;

end module

3.7 VLSI Design Flow

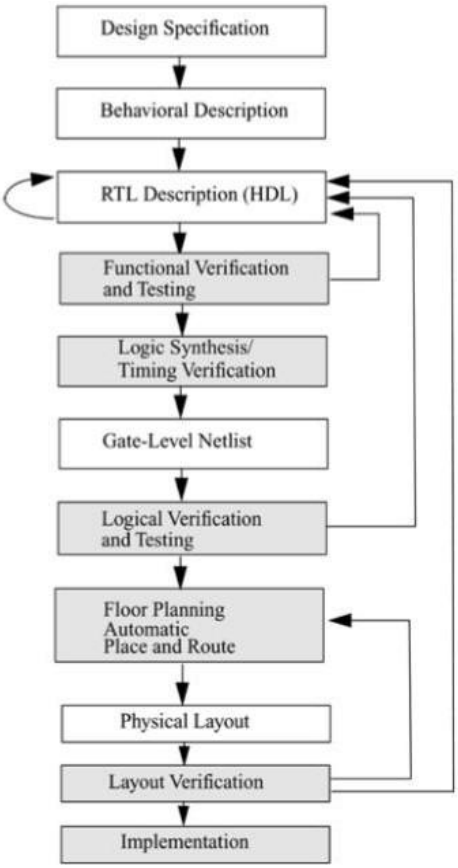


Figure 3.3 VLSI Design Flow

Verilog has four levels of Modeling:

1. The switch level Modeling.
2. Gate-level Modeling.
3. The Data-Flow level.
4. The behavioural Modeling

1) Switch level modeling:

A circuit is defined by explicitly showing how to construct it using transistors like pmos and nmos, predefined modules.

Example:

```
module inverter (out, in);
  output out;
  input in;
  supply0gnd;
  supply1 vdd;
  nmosx1 (out, in, gnd);
  pmosx2(out, in, vdd);
  end module
```

2) Gate level modeling:

A circuit is defined by explicitly showing how to correct it using logic gates, predefined modules, and the connections between them. In this first we think of our circuit as a box or module which is encapsulated from its outer environment, in such a way that its only communication with the outer environment is through input and output ports. We then set out to describe the structure within the module by explicitly describing its gates and sub modules, and how they connect with one another as well as to the module ports. In other words, structural modeling is used to draw a schematic diagram for the circuit. As an example, consider the full-adder below.

Example:

```
module fulladder (a, b, sum, C out);
 Input a, b;
 output sum, C out;
 xor x1(a, b, y);
```

xor x_2(a, b, y);

end module

3) Data-flow modeling:

Dataflow modelling uses Boolean expressions and operators. In this we use assign statement.

Example :

module fulladder (a, b, sum, C_{out});

input a, b;

output sum, C_{out};

assign sum=a^b;

assign C_{out} =a^b;

end module

4) Behavioural modeling:

It is higher level of modeling where behaviour of logic is modelled. Verilog behavioural code is inside procedure blocks, but there is an exception: some behavioural code also exist outside procedure blocks.

There are two types of procedural blocks in Verilog :

Initial: initial blocks execute only once at time zero(start execution at time zero)

Always: always blocks loop to execute over and over again; in other words, as the name suggests, it executes always.

An always statement executes repeatedly, it starts and its execution at 0ns

Syntax:

always@

sensitivitylist)

```
begin
Procedural statements
end
```

Example:

```
module fulladder (a, b, clk, sum);
input a, b, clk;
output sum;
always@ (posedgeclk)
begin
sum =a+b;
end module
```

CHAPTER 4

PYTHON

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

4.1 Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

4.2 Python Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

In Python, variables are created when you assign a value to it:

```
x= 5
y = "Hello, World!"
```

Python allows you to assign a single value to several variables simultaneously. For example – a = b = c = 1

4.3 Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

4.4 Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

var1 = 1
var2 = 10
You can delete a single object or multiple objects by using the del statement. For example –
del var
del var_a, var_b

Python supports four different numerical types −

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)

- complex (complex numbers)

4.5 Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example −

```
str ='Hello World!'

print str          # Prints complete string
print str[0]# Prints first character of the string
print str[2:5]# Prints characters starting from 3rd to 5th
print str[2:]# Prints string starting from 3rd character
print str *2# Prints string two times
print str +"TEST"# Prints concatenated string
```

This will produce the following result –

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

4.6 Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example −

```
list=['abcd',786,2.23,'john',70.2]
tinylist=[123,'john']

print list          # Prints complete list
print list[0]# Prints first element of the list
print list[1:3]# Prints elements starting from 2nd till 3rd
print list[2:]# Prints elements starting from 3rd element
printtinylist*2# Prints list two times
print list +tinylist# Prints concatenated lists
```

This produce the following result −

['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']

4.7 Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example-

```
tuple=('abcd',786,2.23,'john',70.2)
tinytuple=(123,'john')
print tuple          # Prints complete list
```

```
print tuple[0]# Prints first element of the list
print tuple[1:3]# Prints elements starting from 2nd till 3rd
print tuple[2:]# Prints elements starting from 3rd element
```

```
printtinytuple*2# Prints list two times
print tuple +tinytuple# Prints concatenated lists
```

This produce the following result −

('abcd', 786, 2.23, 'john', 70.2)

abcd

(786, 2.23)

(2.23, 'john', 70.2)

(123, 'john', 123, 'john')

('abcd', 786, 2.23, 'john', 70.2, 123, 'john')

4.8 Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

```
dict={}
dict['one']="This is one"
dict[2]="This is two"

tinydict={'name':'john','code':6734,'dept':'sales'}

printdict['one']# Prints value for 'one' key
printdict[2]# Prints value for 2 key
printtinydict# Prints complete dictionary
printtinydict.keys()# Prints all the keys
```

```
printtinydict.values()# Prints all the values
```

This produce the following result −

This is one

This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'}

['dept', 'code', 'name']

['sales', 6734, 'john']

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

4.9 Operators

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

4.9.1 Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators

4.10 Python - Decision Making

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages −

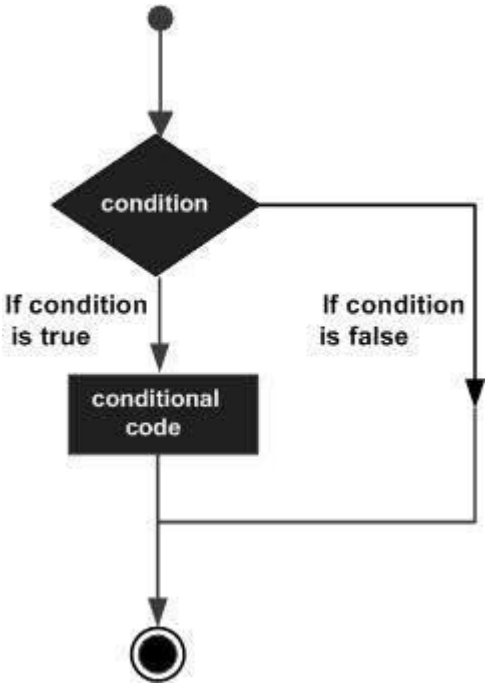


Figure 4.1 Decision Making Structure

Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

4.10.1 IF Statement

Syntax

The syntax of the *if...else* statement is −

if expression:

statement(s)

else: statement(s)

Example

```
var=100
```

```
ifvar==200:
print"1 - Got a true expression value"
printvar
elifvar==150:
print"2 - Got a true expression value"
printvar
elifvar==100:
print"3 - Got a true expression value"
printvar
else:
print"4 - Got a false expression value"
printvar

print"Good bye!"
```

When the above code is executed, it produces the following result −

3 - Got a true expression value

100

Good bye!

4.10.2 Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −

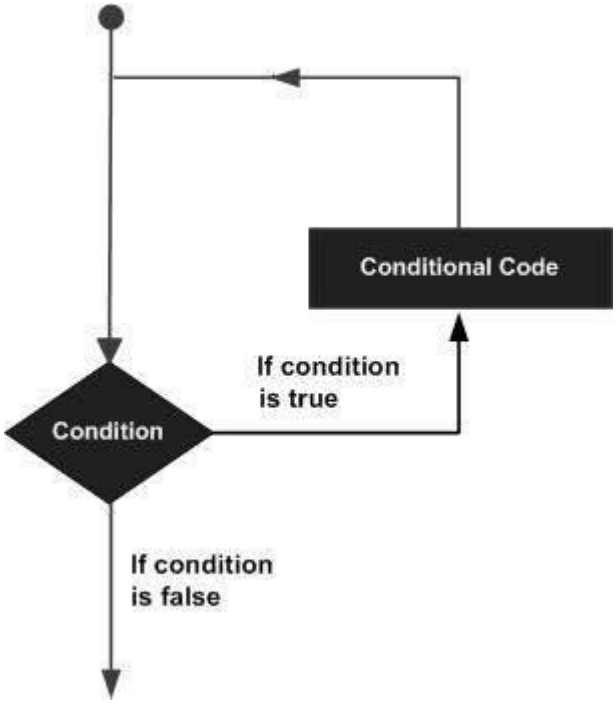


Figure 4.2 General Loop Structure

Python programming language provides following types of loops to handle looping requirements.

4.10.3 While Loop

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

```
while expression:
   statement(s)
```

Example

```
count =0
while(count <9):
print'The count is:', count
  count = count +1


print"Good bye!"
```

When the above code is executed, it produces the following result –

The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

Syntax

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Example

```
for letter in'Python':# First Example
print'Current Letter :', letter

fruits=['banana','apple','mango']
for fruit in fruits:# Second Example
print'Current fruit :', fruit

print"Good bye!"
```

When the above code is executed, it produces the following result −

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

4.11 Files

4.11.1 The *open* Function

Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are parameter details −

- **file_name** − The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

4.11.2 The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

fileObject.close()

4.11.3 Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

4.11.4 The *write()* Method

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string −

Syntax

fileObject.write(string)

Here, passed parameter is the content to be written into the opened file.

4.11.5 The *read()* Method

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

fileObject.read([count])

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

CHAPTER 5
VIVADO and FPGA

5.1 Introduction

This tutorial guides you through the design flow using Xilinx Vivado software to create a simple digital circuit using Verilog HDL. A typical design flow consists of creating model(s), creating user constraint file(s), creating a Vivado project, importing the created models, assigning created constraint file(s), optionally running behavioral simulation, synthesizing the design, implementing the design, generating the bitstream, and finally verifying the functionality in the hardware by downloading the generated bitstream file. You will go through the typical design flow targeting the Artix-100 based Nexys4 board. The typical design flow is shown below. The circled number indicates the corresponding step in this tutorial.

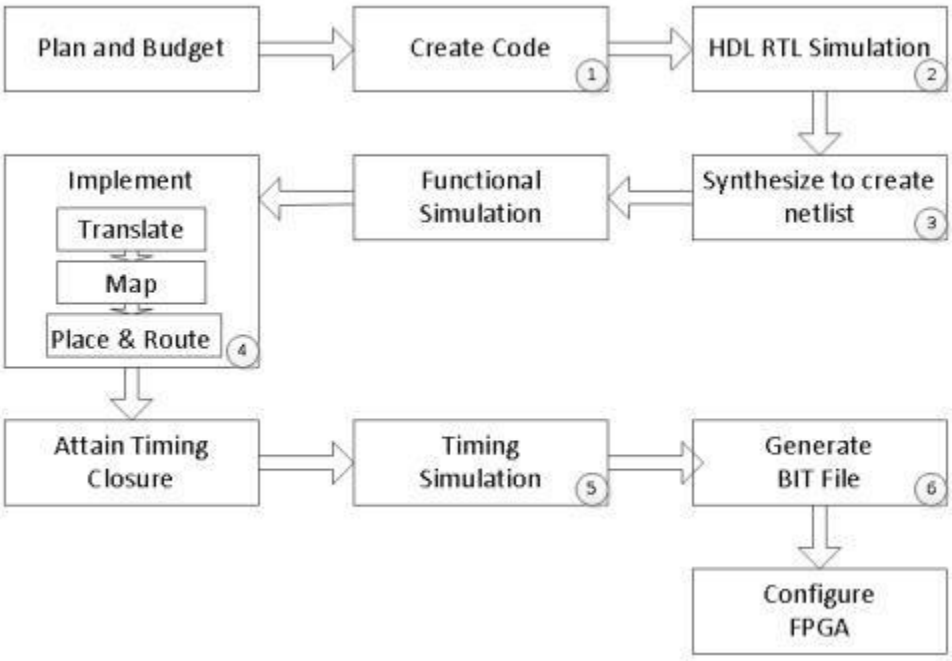


Figure 5.1 A typical design flow

5.2 Design Description

The design consists of some inputs directly connected to the corresponding output LEDs. Other inputs are logically operated on before the results are output on the remaining LEDs as shown in Figure 1.

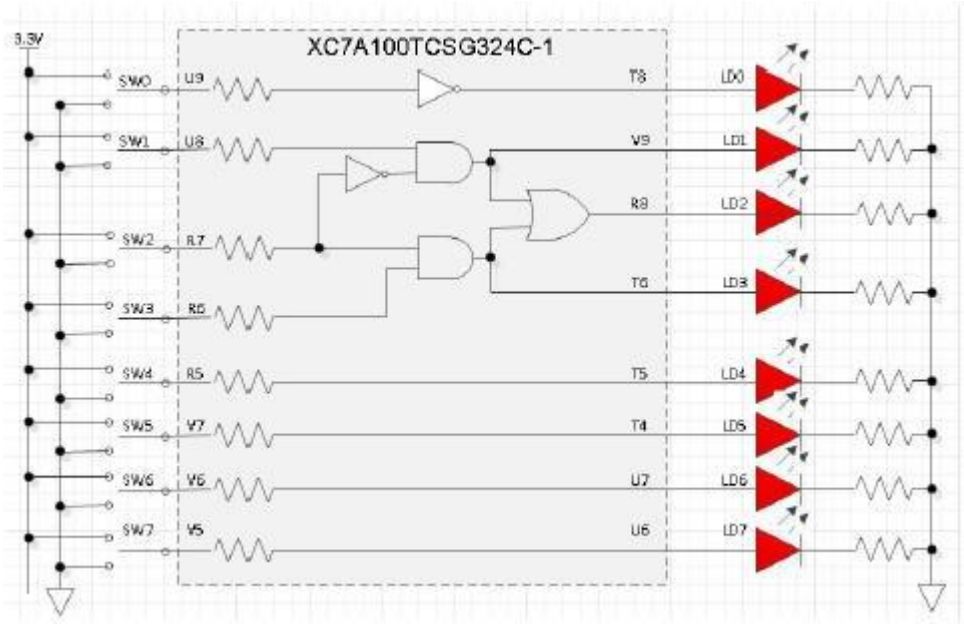


Figure 5.2 Completed Design

5.3 General Flow

- Create a Vivado project and analyze source files
- Simulate the design using XSim simulator
- Synthesize the design
- Implement the design
- Perform the timing simulation
- Verify the functionality in hardware using the Nexys4 board
- Run the tools in batch mode using the provided Tcl script

5.3.1 Create a Vivado Project using IDE

- Launch Vivado and create a project targeting the XC7A100TCSG324C-1 device and using the Verilog HDL. Use the provided tutorial.v and tutorial.xdc files from the sources directory.
- Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado 2013.3**
- Click **Create New Project** to start the wizard. You will see Create A New Vivado Project dialog box. Click **Next**.
- Click the Browse button of the Project location field of the **New Project** form, browse to **c:\xup\digital**, and click **Select**.
- Enter **tutorial** in the Project name field. Make sure that the Create Project Subdirectory box is checked. Click **Next**.

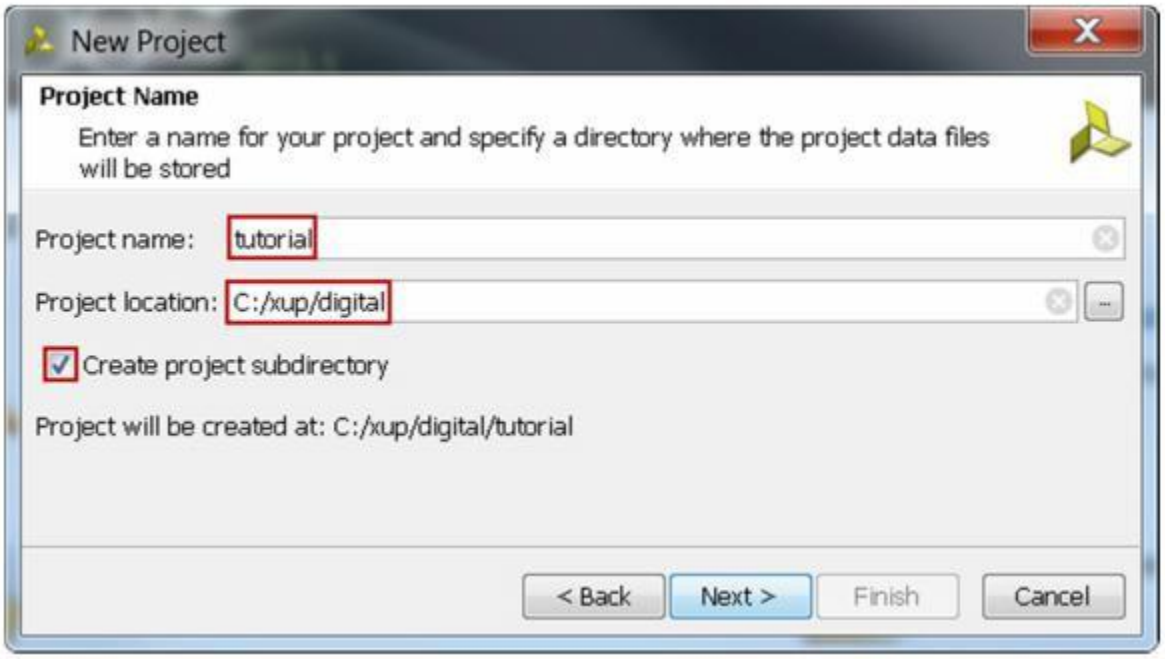


Figure 5.3 Project Name and Location entry

- Select **RTL Project** option in the Project Type form, and click **Next**.
- Select **Verilog** as the Target language and Simulator language in the Add Sources form.

- Click on the **Add Files…** button, browse to the **c:\xup\digital\sources\tutorial** directory, select tutorial.v, click **Open**, and then click **Next**.
- Click **Next** to get to the Add Constraints form.
- Click **Next** if the entry is already auto-populated, otherwise click on the **Add Files…** button, browse to the **c:\xup\digital\sources\turorial** directory and select tutorial.xdc, and click **Open**.

  This Xilinx Design Constraints file assigns the physical IO locations on FPGA to the switches and LEDs located on the board. This information can be obtained either through a board's schematic or board's user guide.
- In the Default Part form, using the **Parts** option and various drop-down fields of the **Filter** section, select the **XC7A100TCSG324-1** part. Click **Next**.

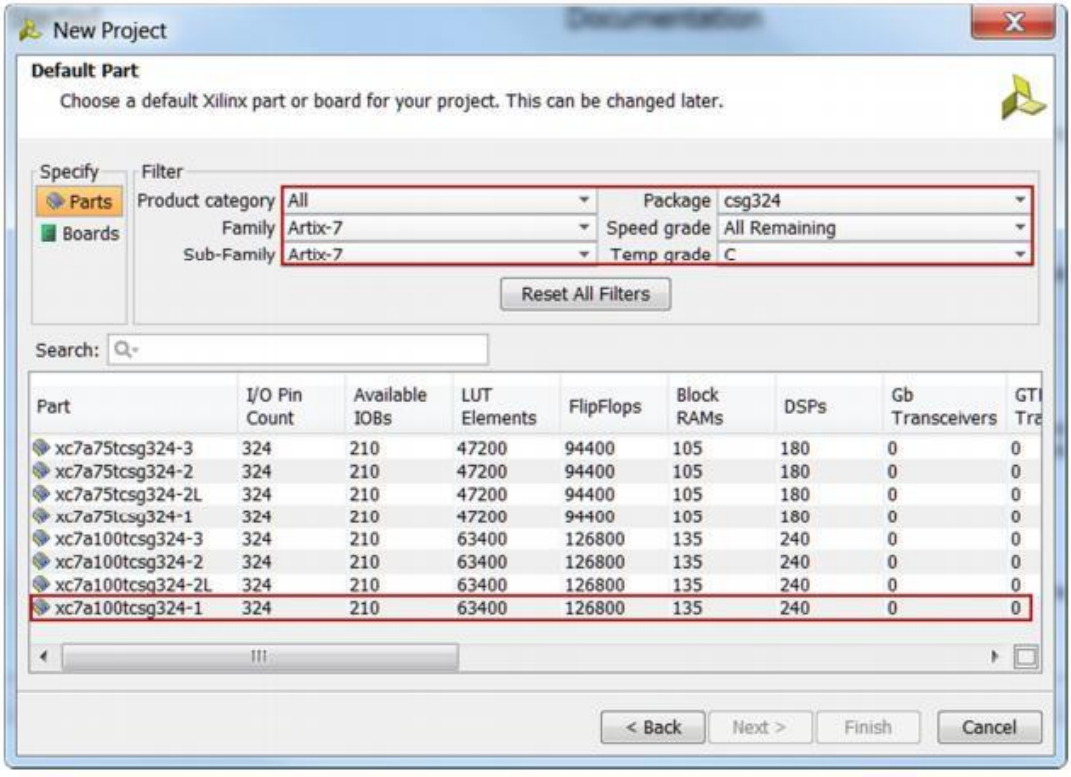


Figure 5.4 Part Selection

- Click **Finish** to create the Vivado project.

Use the Windows Explorer and look at the c:\xup\digital\tutorial directory. You will find that the tutorial.data and tutorial.srcs directories and the tutorial.xpr (Vivado) project file have been created. The tutorial.data directory is a place holder for the Vivado program database. Two more directories, constrs_1 and sources_1, are created under the tutorial.srcs directory; deep down under them, the copied tutorial.xdc (constraint) and tutorial.v (source) files are respectively placed.

5.4 FPGA (Field Programmable Gate Array)

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC). Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools.

Figure 5.5 Spartan FPGA from Xilinx

FPGAs contain an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Many

FPGAs can be reprogrammed to implement different logic functions, allowing flexible reconfigurable computing as performed in computer software.

The full form of **FPGA** is "**Field Programmable Gate Array**". It contains ten thousand to more than a million logic gates with programmable interconnection. Programmable interconnections are available for users or designers to perform given functions easily. A typical model FPGA chip is shown in the given figure. There are I/O blocks, which are designed and numbered according to function. For each module of logic level composition, there are **CLB's (Configurable Logic Blocks)**.

CLB performs the logic operation given to the module. The inter connection between CLB and I/O blocks are made with the help of horizontal routing channels, vertical routing channels and PSM (Programmable Multiplexers).

The number of CLB it contains only decides the complexity of FPGA. The functionality of CLB's and PSM are designed by VHDL or any other hardware descriptive language. After programming, CLB and PSM are placed on chip and connected with each other with routing channels.

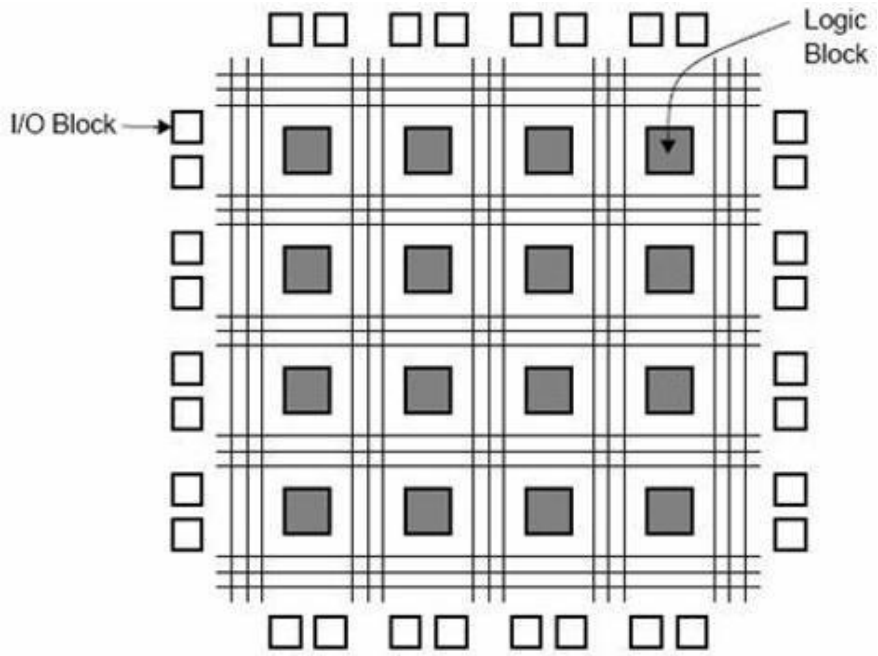


Figure 5.6 Field Programmable Gate Array Chip

5.5 Advantages of FPGA

- It requires very small time; starting from design process to functional chip.
- No physical manufacturing steps are involved in it.
- The only disadvantage is, it is costly than other styles.

5.6 Features of FPGA artix7

- Leading system performance-per-watt for cost-sensitive applications
    - Lowest total power per logic cell
    - Best-in-class performance for DDR3, DSP, parallel, and serial I/O
    - Low-cost, small-footprint packaging
    - Part of a broad cost-effective all-programmable low-end portfolio
- Programmable system integration
    - Up to 215 K LCs; AXI IP and analog mixed signal integration
- Increased system performance
    - Up to 16 x 6.6G GTs, 930 GMAC/s, 13 Mb BRAM, 1.2 Gb/s LVDS, DDR3-1066
- BOM cost-reduction
    - Small wire-bond packaging and analog component savings
- Total power reduction
    - 65% lower static and 50% lower power than 45 nm generation devices
- Accelerated design productivity
    - Scalable optimized architecture, comprehensive tools, and IP

5.7 Applications of FPGA artix7

- Low-cost ultrasound
- Wireless backhaul Artix-7 solution
- Multi-protocol machine-vision cameras
- Programmable logic controllers

- Small form-factor and battery-powered software-defined radios

5.8 NEXYS A7 BOARD

The Nexys A7 board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx®. With its large, high-capacity FPGA, generous external memories, and collection of USB, Ethernet, and other ports, the Nexys A7 can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys A7 to be used for a wide range of designs without needing any other components.

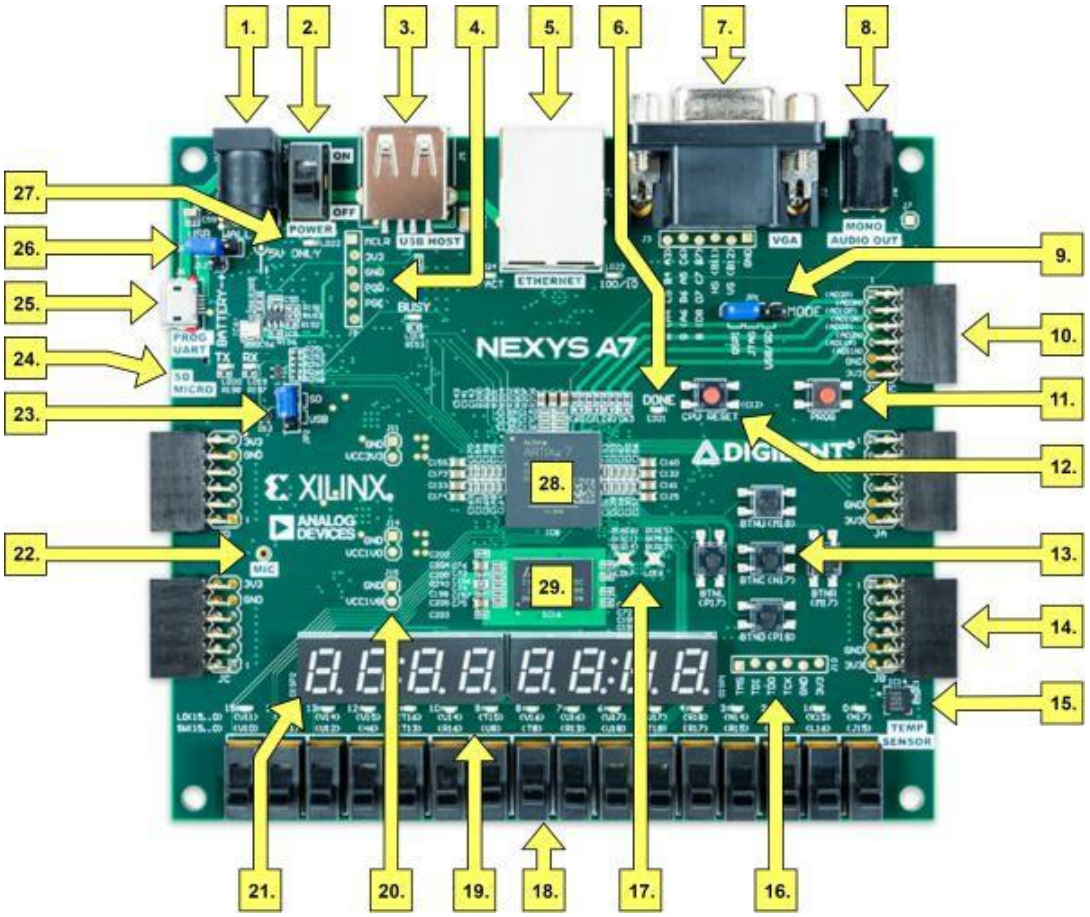


Figure 5.7 Nexys A7 Feature Callout Board

Functional Board Description

Callout	Component Description	Call out	Component Description
1	Power jack	**16**	JTAG port for (optional) external cable
2	Power switch	**17**	Tri-color (RGB) LEDs
3	USB host connector	**18**	Slide switches (16)
4	PIC24 programming port	**19**	LEDs (16)
5	Ethernet connector	**20**	Power supply test point(s)
6	FPGA programming done LED	**21**	Eight digit 7-seg display
7	VGA connector	**22**	Microphone
8	Audio connector	**23**	External configuration jumper (USB)
9	Programming mode jumper	**24**	MicroSD card slot
10	Analog signal Pmod port (XADC)	**25**	Shared UART/ JTAG USB port
11	FPGA configuration reset button	**26**	Power select jumper and battery header
12	CPU reset button (for soft cores)	**27**	Power-good LED
13	Five pushbuttons	**28**	Xilinx Artix-7 FPGA
14	Pmod port(s)	**29**	DDR2 memory
15	Temperature sensor		

Table 5.1 Functional board description of Nexys A7 board

CHAPTER 6

RESULTS

In this chapter Schematic of the Processor and various programs simulated in this processor using Iverilog and Vivado softwares are shown in figures below:

6.1 Schematic

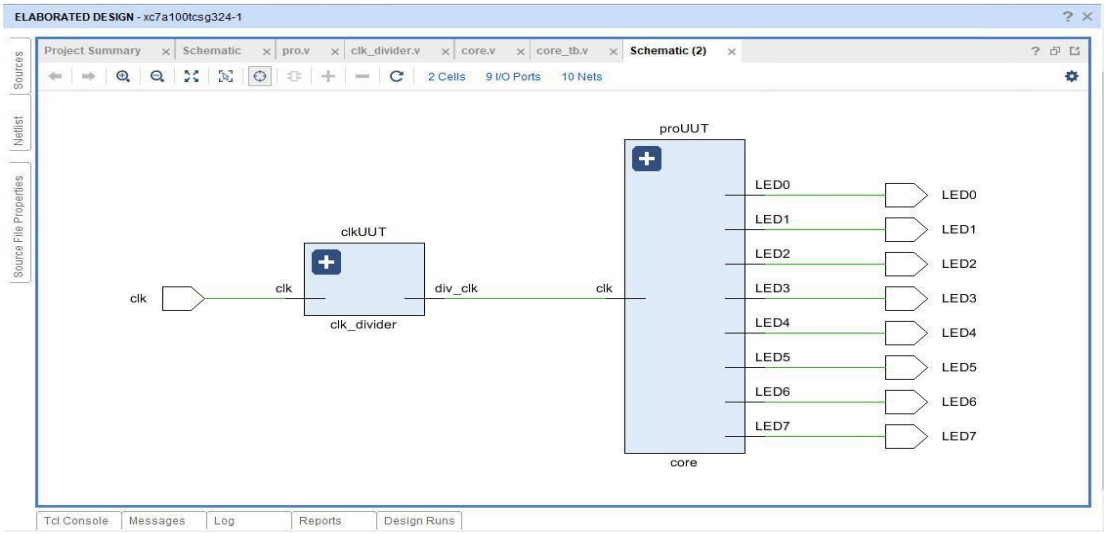


Figure 6.1 Schematic Block

6.2 Multiplication Output

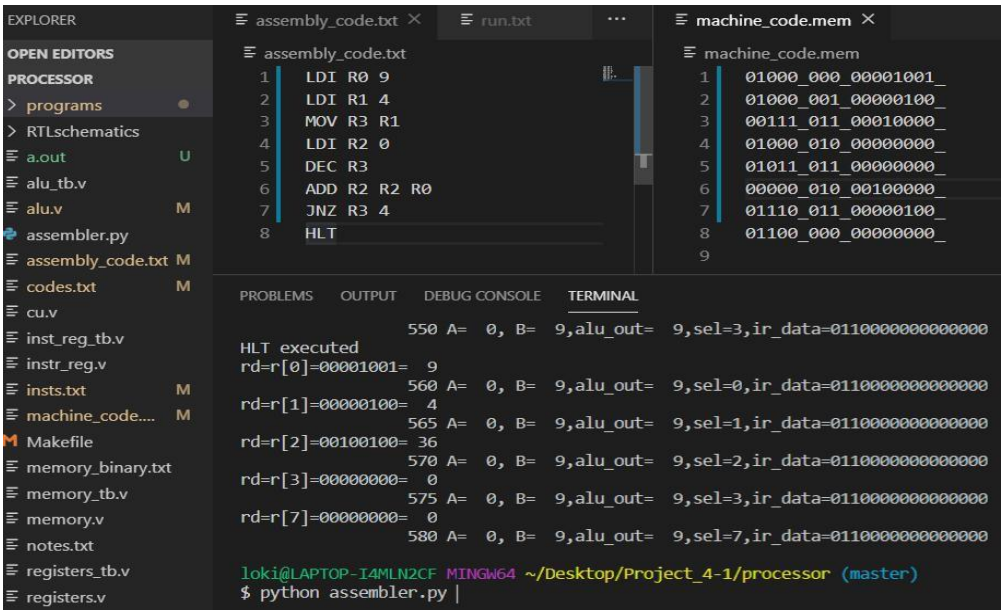


Figure 6.2 Multiplication Output

6.3 Memory operations Output

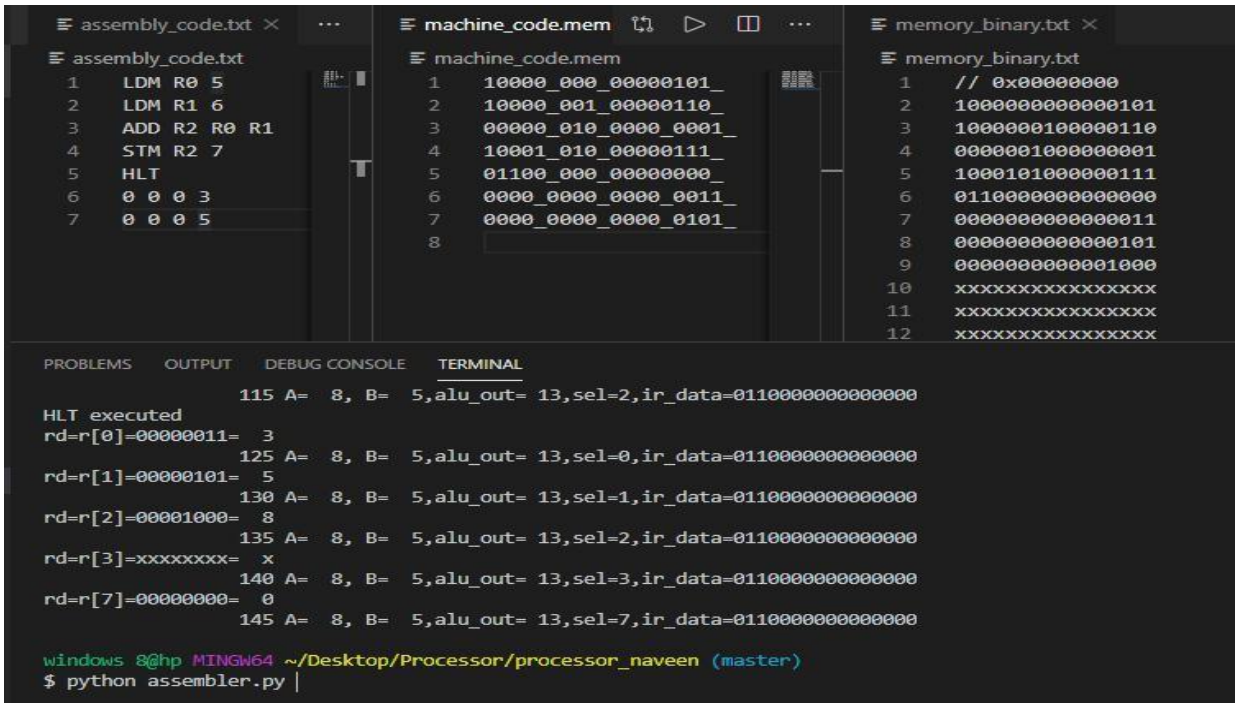


Figure 6.3 Memory operations output

6.4 Zeros counter output

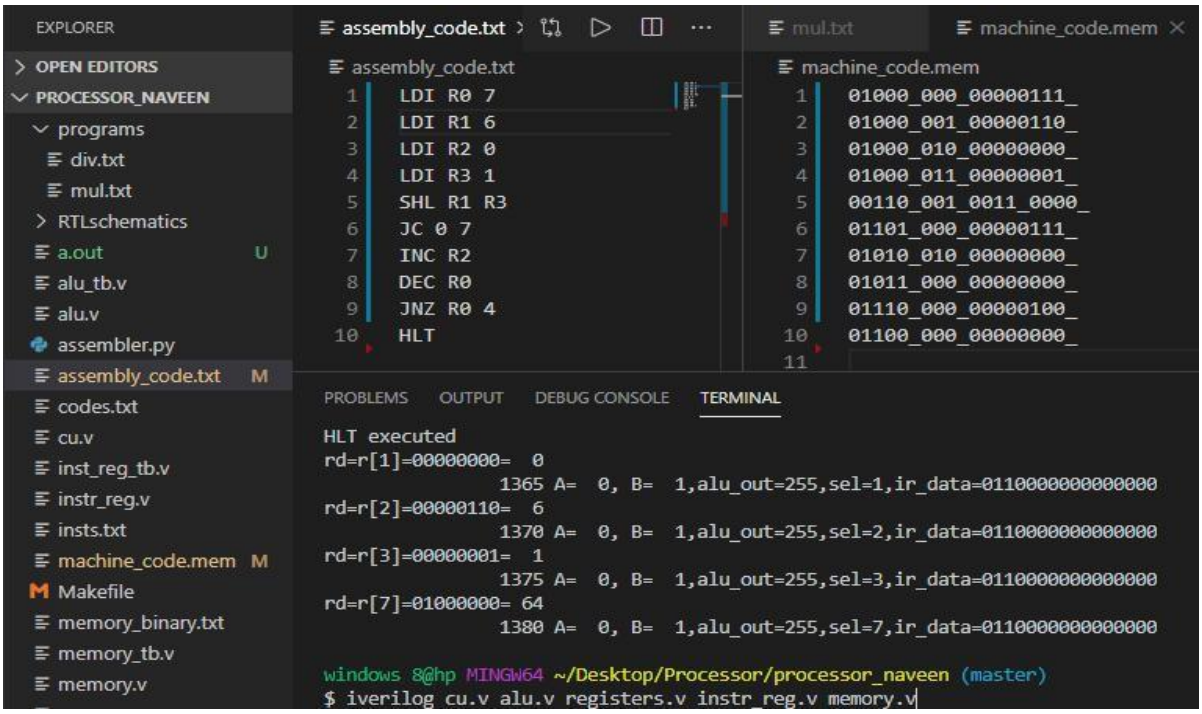


Figure 6.4 Zeros Counter output

6.5 Division output

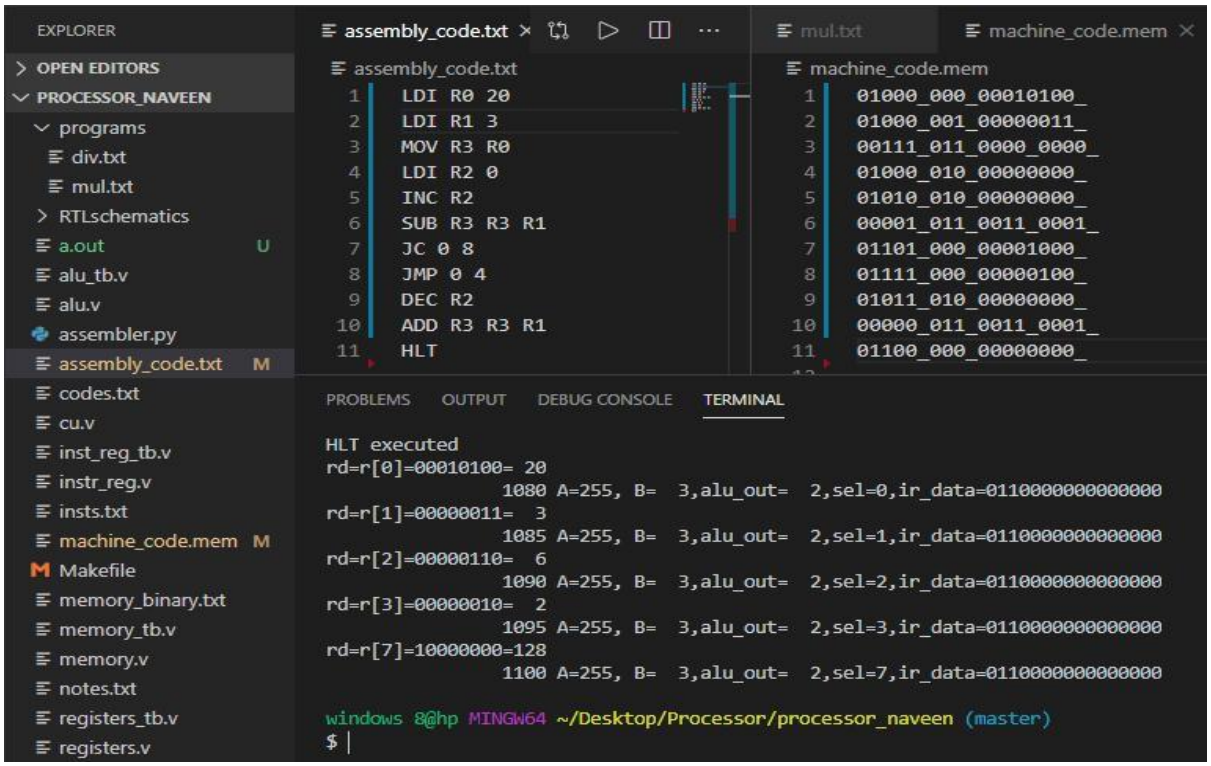


Figure 6.5 Division output

6.6 Up Counter

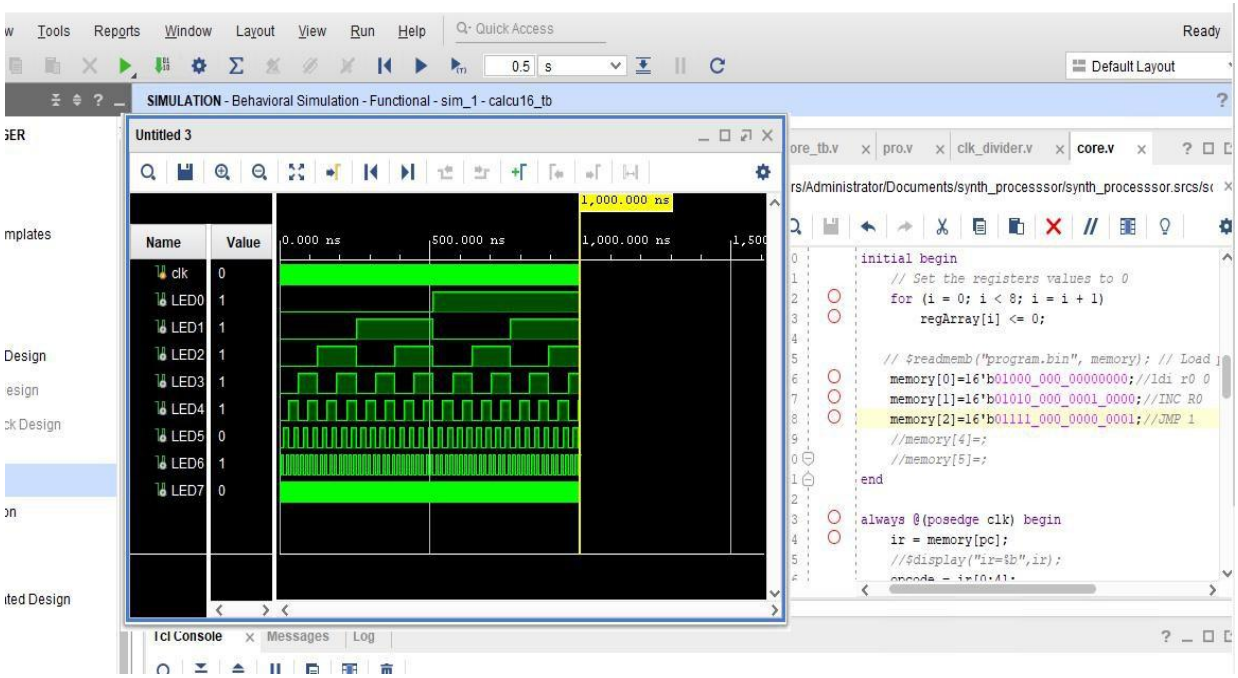


Figure 6.6 UP Counter

Result

CONCLUSION

In Digital systems for processing the data processors plays a major role.It reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions and provides results as output.In this we implemented a basic RISC processor which can execute all Arithmetic, Logical, Memory Operations and some Branch instructions using Verilog Hardware Description language.

As the machine codes which are taken by the processor for execution are not human friendly to write a program. Along with this processor we also implemented an Assembler which converts assembly level codes to machine codes using Python programming language which makes easier to write programs without any difficulty for this processor.

In order to write Verilog Code with Syntax highlighting we used Visual Studio Code which is a source-code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control, syntaxhighlighting and intelligent code completion.For simulating Verilog Code we used Icarus Verilog which is a Verilog simulation and synthesis tool.

REFERENCES

[1] Ramesh Gaonkar, Author, "Microprocessor Architecture, Programming and Applications with the 8085 6/e", Penram International Publishing. **(2013)**.

[2] M. Morris Mano, Author, "Computer System Architecture, 3e", Pearson Education India. **(2007)**.

[3] Priyavrat Bhardwaj and Siddharth Murugesan, "Design & Simulation of a 32-Bit RISC Based MIPS Processor Using Verilog", International Journal of Research in Engineering and Technology. vol. 5 **(2016)**.

[4] Ramandeep Kaur and Anuj, "8 Bit RISC Processor Using Verilog HDL", International. Journal of Engineering Research and Applications. vol. 4, no. 3, **(2014)**, pp. 417-422.

[5] R Uma "Design and performance analysis of 8 bit RISC processor using Xilinx tools", International Journal of Engineering Research and Applications **(2012)**, pp 053-058.

[6] B. Rajesh Kumar, Ravisaketh and Santha Kumar, "Implementation of A 16-bit RISC Processor for Convolution Application", Research India publications, **(2014)**, pp 441-446.

[7] M. Morris Mano, Author, "Digital Design", 5 edition, Prentice Hall. **(2012)**.

[8] MD.Shabeena Begum and M.Kishore Kumar, "FPGA based implementation of 32 bit risc processor", International Journal of Engineering Research and Applications (IJERA) . vol. 1, pp 1148-1151.

[9] Oscar Camacho Nieto and Jorge A. Huerta Ruelas "Design of a General Purpose 8- bit RISC Processor for Computer Architecture Learning", Research gate publication. vol. 19, no. 2, **(2015)**, pp 371–385.

[10] Oztekin, H., Temurtas, F., & Gulbag, A. "Microprocessor architecture design on reconfigurable hardware as an educational tool.", IEEE Symposium on Computers & Informatics, **(2011)**, pp. 385–389.

[11] Imyong lee, Dongwook Lee, Kiyoung choi "ODALRISC: A Small, Low power and Configurable 32-bit RISC processor" International SOC design conference **(2008)**.

[12] Mr.Sagar, P.Ritpurka, Prof Mangesh N.Thakare, Prof.Girish, D.Korde, "Review on 32Bit MIPS RISC Processor using VHDL", International Conference on Advances in Engineering & Technology **(2014)**, PP 46-50.

[13] M r. Sagar P. Ritpurkar, Prof. Mangesh N. Thakare, Prof. Girish D.Korde, "Review on 32Bit MIPS RISC Processor using VHDL",International Conference on Advances in Engineering & Technology **(2014)**.

[14] Mohit N. Topiwala, N. Sarawathi, "Implementation of a 32-bit MIPS Based RISC Processor using Cadence",International Conference on Advanced Communication Computer Modelling and Simulation **(2009)**.

PAPER PUBLICATION

[1] P S M Veena, S Naveen Kumar, G Vasanthi, P Tirumala and B Leela Madhav, "Implementation of 8-Bit RISC Processor", Journal of Information and Computational Science, Volume 10, Issue 4, ISSN: 1548-7741, April 2020.