ABSTRACT

This microcontroller has been designed using modular blocks and correspond Harvard-type architecture. It has a stack of eight levels and an Arithmetic Logic Unit (ALU) for operations with 8-bit data and bit-oriented instructions. Also, it has a program memory of 8Kx14-bit and a data memory of 512x8-bit.

The proposed device has an instruction OPCODE correspondent to PIC16FXX processor core, so the system can be programmed with the same machine language. It also presents a high support for compilers such as MPASM (Microchip Assembler) and CCS PIC-C (C language). The compatibility with the C compiler is quite interesting, making easy the development of large-scale digital designs, thanks to the existence of extensive libraries in this programming language.

Keywords: Embedded Microcontroller, FPGA, Harvard Architecture, Verilog

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1: INTRODUCTION TO FPGA

The acronym **FPGA** stands for **Field Programmable Gate Array**. It is an integrated circuit that can be programmed by a user for a specific use after it has been manufactured. Contemporary FPGAs contain adaptive logic modules (ALMs) and logic elements (LEs) connected via programmable interconnects. These blocks create a physical array of logic gates that can be customized to perform specific computing tasks. This makes them very different from other types of microcontrollers or Central Processing Units (CPUs), whose configuration is set and sealed by a manufacturer and cannot be modified.

FPGAs consist of logical modules connected by routing channels. Each module is made up of a programmable lookup table that is used to control the elements that each cell consists of and to perform logical functions of the elements that make up the cell. In addition to the lookup table, each cell contains cascaded adders enabling addition to be done. Subtraction can also be done by changing the logical states of the input. Beyond these, there are also registers (logical elements used to perform the simplest memory functions) and multiplexers (switching elements).

FPGAs can also include static and dynamic on-chip memories, depending on the specific manufacturer model. In addition, in FPGAs you can find ready components, such as CPU cores, memory controllers, USB controllers or network cards. These components are so popular that there is no need to implement them in the FPGA structure. Instead, you can use an already manufactured component.[5]

1.1.1: USES

FPGAs are mainly used to design application-specific integrated circuits (ASICs). First, you design the architecture of such a circuit. Then, **you use an FPGA to build and check its prototype**. Errors can be corrected. Once the prototype works as expected, an ASIC project is created and manufactured based on the FPGA design. This allows you to save time, as manufacturing an integrated circuit can be a very complex and time-consuming process. It also saves money, as one FPGA can be used to prepare many iterations of the same project.

FPGAs are used in projects where hardware configuration is subject to change and a circuit that can be adjusted to these changes is called for. In case you change your hardware suppliers and the new hardware does not have the required interface, FPGA becomes a natural choice. [4]

1.2: INTRODUCTION TO MICROCONTROLLER

A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip.

A microcontroller is embedded inside of a system to control a singular function in a device. It does this by interpreting data it receives from its I/O peripherals using its central processor. The temporary information that the microcontroller receives is stored in its data memory, where the processor accesses it and uses instructions stored in its program memory to decipher and apply the incoming data. It then uses its I/O peripherals to communicate and enact the appropriate action. Microcontrollers are used in a wide array of systems and devices. Devices often utilize multiple microcontrollers that work together within the device to handle their respective tasks.

Common MCUs include the Intel MCS-51, often referred to as an 8051 microcontroller, which was first developed in 1985; the AVR microcontroller developed by Atmel in 1996; the programmable interface controller (PIC) from Microchip Technology; and various licensed Advanced RISC Machines (ARM) microcontrollers. A number of companies manufacture and sell microcontrollers, including NXP Semiconductors, Renesas Electronics, Silicon Labs and Texas Instruments.[4]

CHAPTER 2: HDL LANGUAGES USED

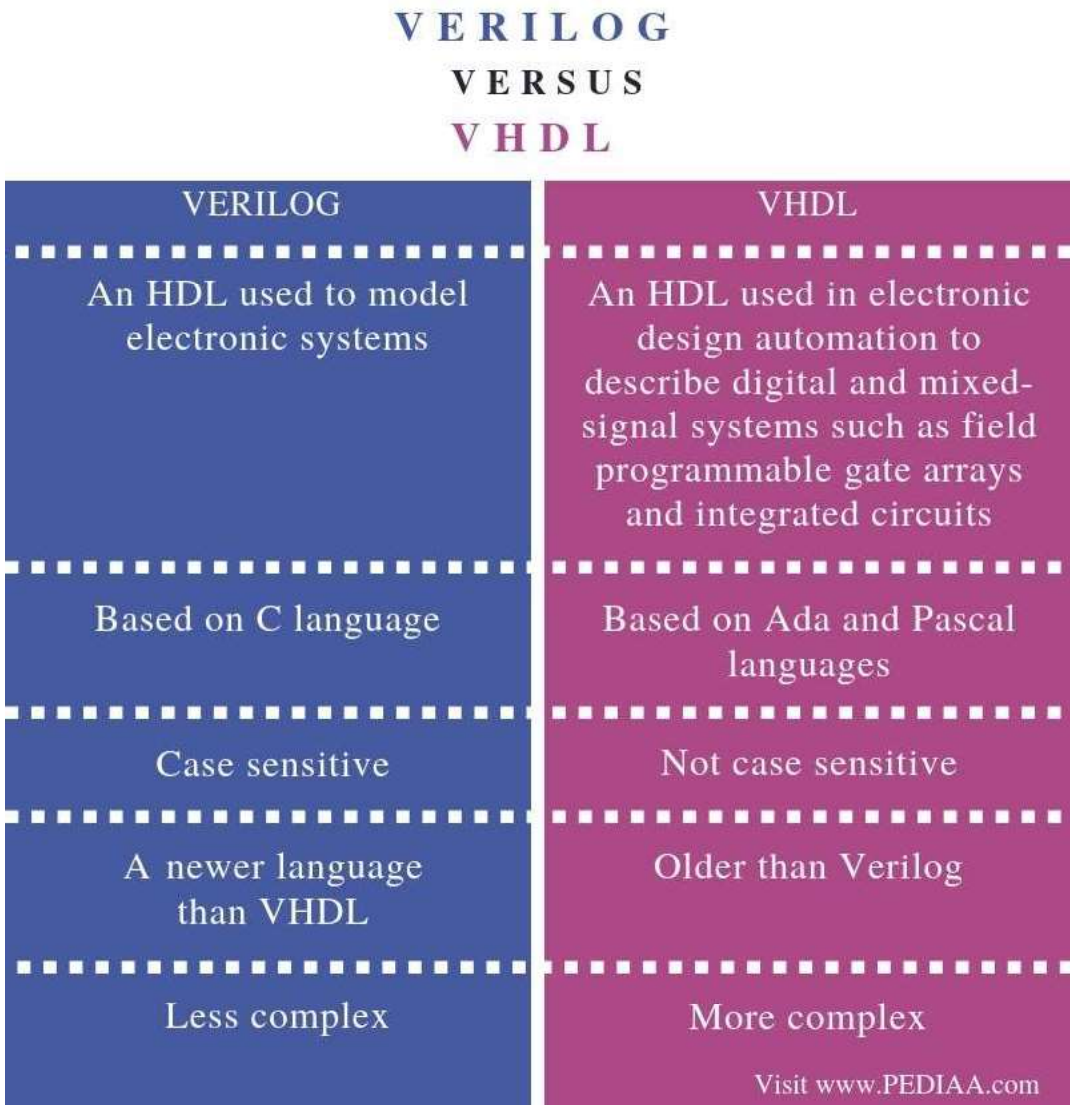
2.1: VERILOG

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.[4]

2.2: VHDL

The **VHSIC Hardware Description Language** (**VHDL**) is a hardware description language (HDL) that can model the behavior and structure of digital systems at multiple levels of abstraction, ranging from the system level down to that of logic gates, for design entry, documentation, and verification purposes.[4]

2.3: DIFFERENCE BETWEEN VERILOG AND VHDL



VERILOG

VERSUS

VHDL

VERILOG	VHDL
An HDL used to model electronic systems	An HDL used in electronic design automation to describe digital and mixed-signal systems such as field programmable gate arrays and integrated circuits
Based on C language	Based on Ada and Pascal languages
Case sensitive	Not case sensitive
A newer language than VHDL	Older than Verilog
Less complex	More complex

Visit www.PEDIAA.com

Fig 2.1: Comparison between Verilog and VHDL

CHAPTER 3: COMPONENTS OF MICROCONTROLLER:

3.1: Block Diagram

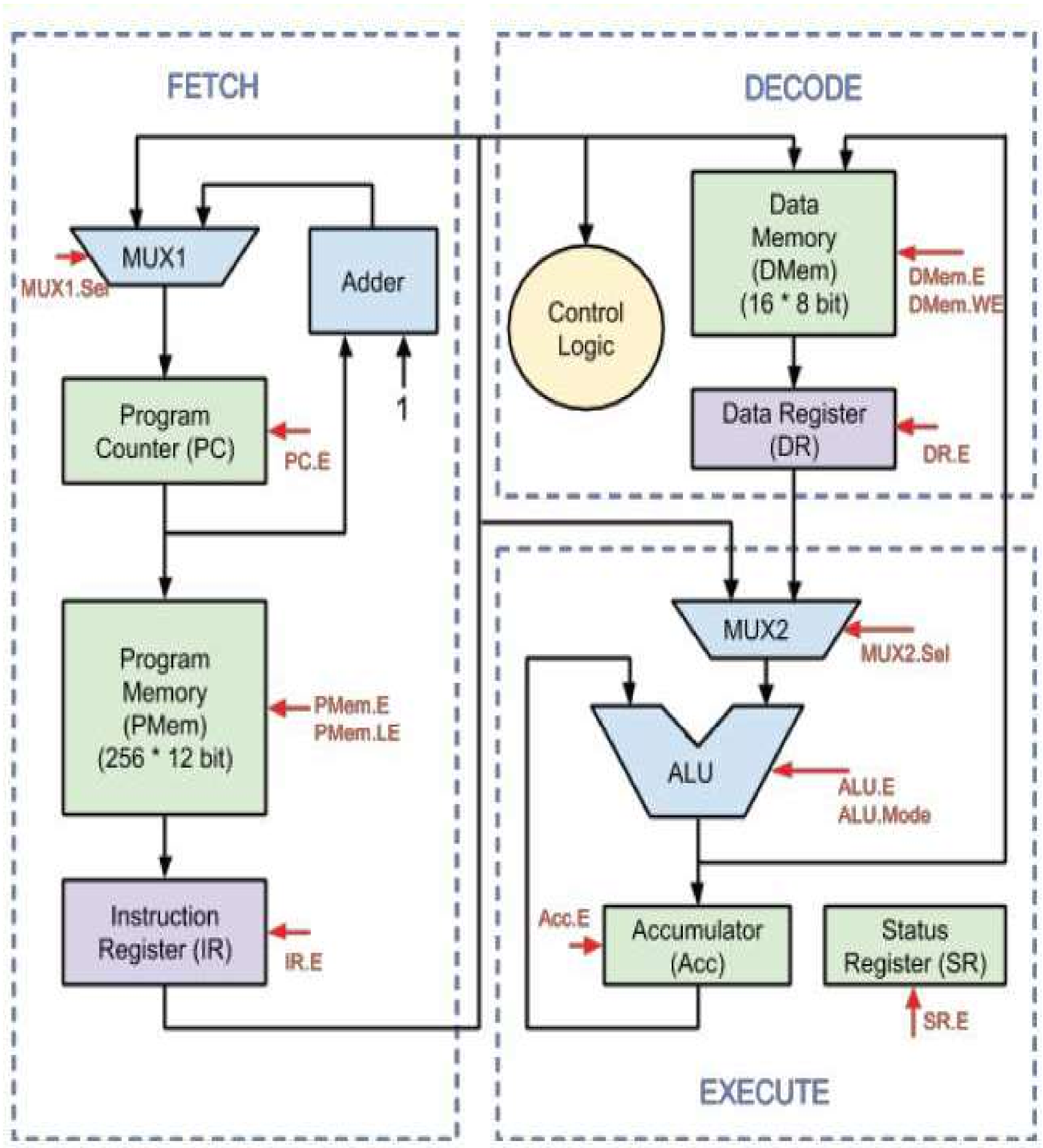


Fig 3.1: Block Diagram of the microcontroller

3.2: Registers

The microcontroller has 3 programmer visible register:
Program Counter (8 bit, denoted as PC): contains the index of current executing instruction.
Accumulator (8 bit, denoted as Acc): The accumulator register (Working Register) is a block which loads the8-bitdata from the ALU output and hold them for in the next clock cycle of the same ALU operates that data with a new one from the RAM or the IR. It holds result and 1 operand of the arithmetic or logic calculation.
Status Register (4 bit, denoted as SR): holds 4 status bit, i.e. Z, C, S, O.

Z (zero flag, SR[3]): 1 if result is zero, 0 otherwise.
C (carry flag, SR[2]): 1 if carry is generated, 0 otherwise.
S (sign flag, SR[1]): 1 if result is negative (as 2's complement), 0 otherwise.
O (overflow flag, SR[0]): 1 if result generates overflow, 0 otherwise.

Each of these registers has an enable port, as a flag for whether the value of the register should be updated in state transition. They are denoted as PC.E, Acc.E, and SR.E.
The microcontroller has 2 programmer invisible registers (i.e. they can not be manipulated by programmer):

Instruction Register (12 bit, denoted as IR): It is a register like the Accumulator, but it is a 14-bit register. It receives the instructions from the ROM and hold them to be used by different blocks as the data multiplexer MUX, the address multiplexer ADDR MUX, the ALU and the Control Unit. It contains the current executing instruction.

Data Register (8 bit, denoted as DR): contains the operand read from data memory.

Similarly, each of these registers has an enable port as a flag for whether the value of the register should be updated in state transition. They are denoted as IR.E and DR.E.[1]

3.3: PROGRAM MEMORY

The microcontroller has a 256 entry program memory that stores program instructions, denoted as PMem. Each entry is 12 bits, the ith entry is denoted as PMem[i]. The program memory has the following input/output ports.

Enable port (1 bit, input, denoted as PMem.E): enable the device, i.e. if it is 1, then the entry specified by the address port will be read out, otherwise, nothing is read out.
Address port (8 bit, input, denoted as PMem.Addr): specify which instruction entry is read out, connected to PC.
Instruction port (12 bit, output, denoted as PMem.I): the instruction entry that is read out, connected to IR. 3 special ports are used to load program to the memory, not used for executing instructions.
Load enable port (1 bit, input, denoted as PMem.LE): enable the load, i.e. if it is 1, then the entry specified by the address port will be load with the value specified by the load instruction input port and the instruction port is supplied with the same value; otherwise, the entry specified by the address port will be read out on instruction port, and value on instruction load port is ignored.
Load address port (8 bit, input, denoted as PMem.LA): specify which instruction entry is loaded.
Load instruction port (12 bit, input, denoted as PMem.LI): the instruction that is loaded.

For example, if the address point is supplied with 8'b0000_0011 and enable is set to 1, the fourth entry is read out on instruction port.
Note that program load only takes effect on clock rising edge, while instruction read out happens all the time.[1]

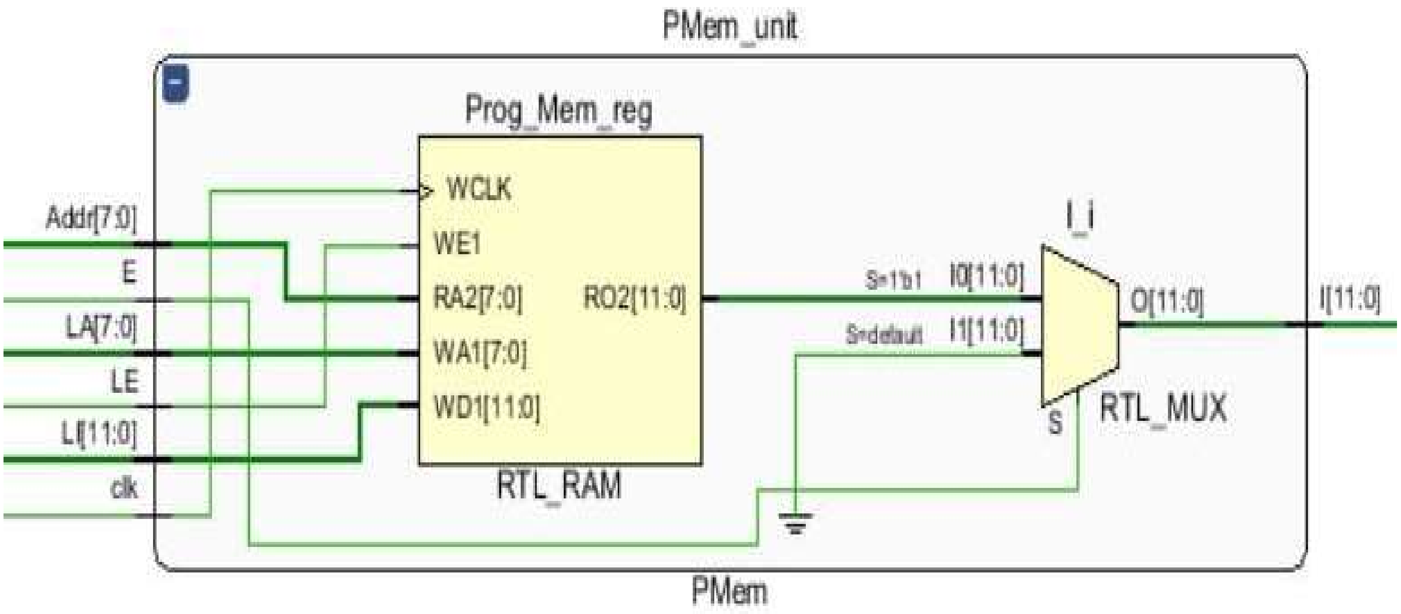


Fig 3.2: Implementation of Program Memory

3.4: DATA MEMORY

The microcontroller has a 16 entry data memory, denoted as DMem. Each entry is 8 bits, the ith entry is denoted as Dmem[i]. The program memory has the following input/output ports.

> **Enable port** (1 bit, input, denoted as Dmem.E): enable the device, i.e. if it is 1, then the entry specified by the address port will be read out or written in; otherwise nothing is read out or written in.
>
> **Write enable port(1 bit, input, denoted as Dmem.WE):** enable the write, i.e. if it is 1, then the entry specified by the address port will be written with the value specified by the data input port and the data output port is supplied with the same value; otherwise, the entry specified by the address port will be read out on data output port, and value on data input port is ignored.
>
> **Address port (4 bit, input, denoted as Dmem.Addr):** specify which data entry is read out, connected to IR[3:0].
>
> **Data input port (8 bit, input, denoted as Dmem.DI)**: the value that is written in, connected to ALU.Out.
>
> **Data output port (8 bit, output, denoted as Dmem.DO):** the data entry that is read out, connected to MUX2.In1.

For example, if the address point is supplied with 8'0000_0011, data input port is supplied with 8'0000_0000, enable is set to 1, and write enable is set to 1, the fourth entry of the data memory is written with value 0 and the data output port shows 8'0000_0000.
As another example, if the address point is supplied with 8'0000_0011, data input port is supplied with 8'0000_0000, enable is set to 1, while write enable is set to 0, the fourth entry of the data memory is read out on data output port.[1]

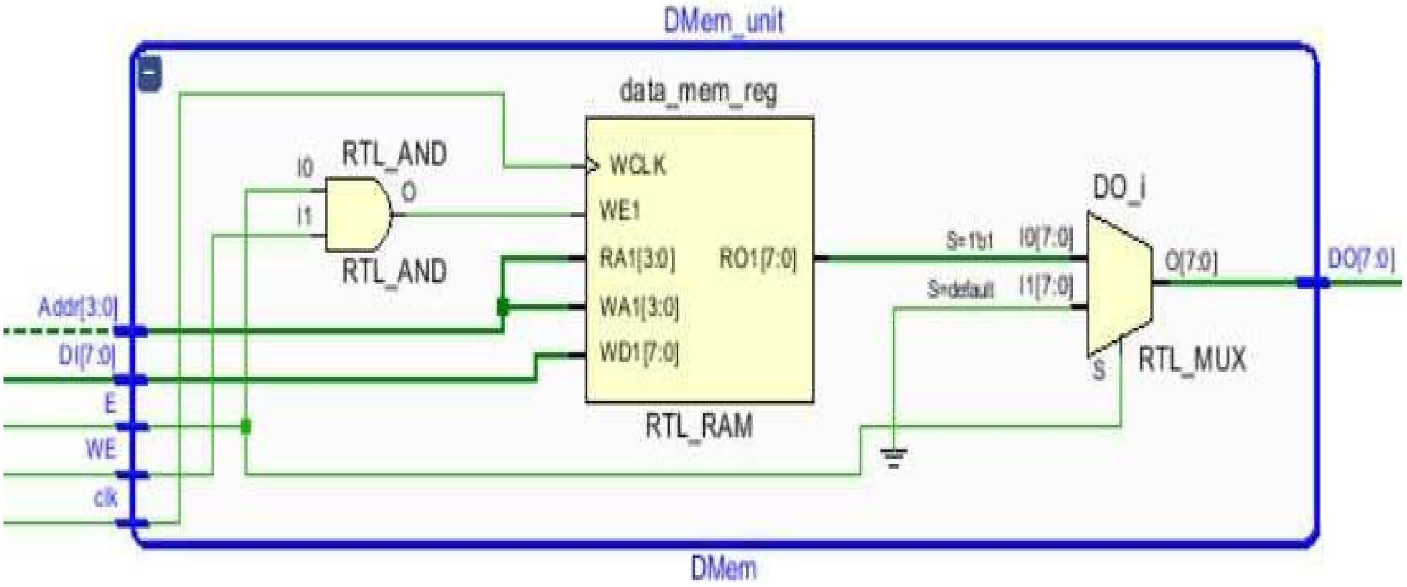


Fig 3.3: Implementation of Data Memory

3.5: PC ADDER

PC adder is used to add PC by 1, i.e. move to the next instruction. This component is pure combinational. It has the following port.

Adder input port (8 bit, input, denoted as Adder.In): connected to PC.
Adder output port (8 bit, output, denoted as Adder.Out): connected to MUX1.In2.

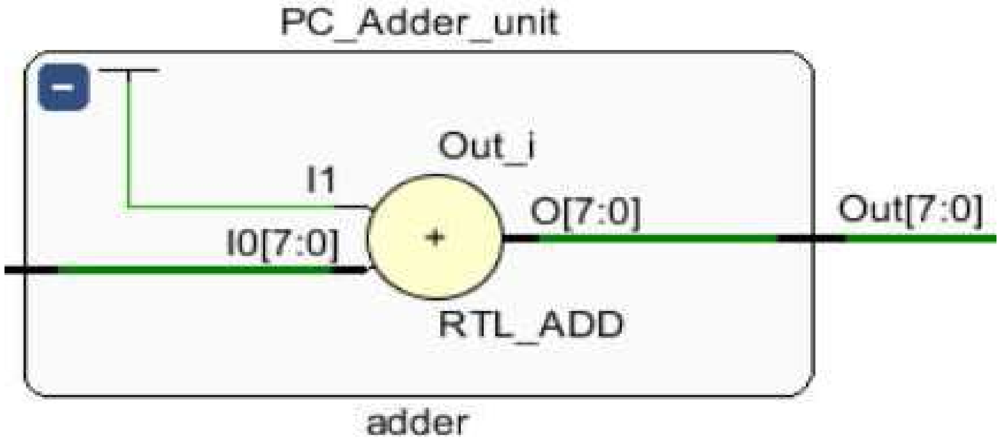


Fig 3.4: Implementation of PC Adder

3.6: INSTRUCTION MUX

MUX1 is used to choose the source for updating PC. If the current instruction is not a branch or it is a branch but the branch is not taken, PC is incremented by 1; otherwise PC is set to the jumping target, i.e. IR [7:0]. It has the following port.

MUX1 input 1 port (8 bit, input, denoted as MUX1.In1): connected to IR [7:0].
MUX1 input 2 port (8 bit, input, denoted as MUX1.In2): connected to Adder.Out.
MUX1 selection port (1 bit, input, denoted as MUX1.Sel): connected to control logic.
MUX1 output port (8 bit, output, denoted as MUX1.Out): connected to PC.

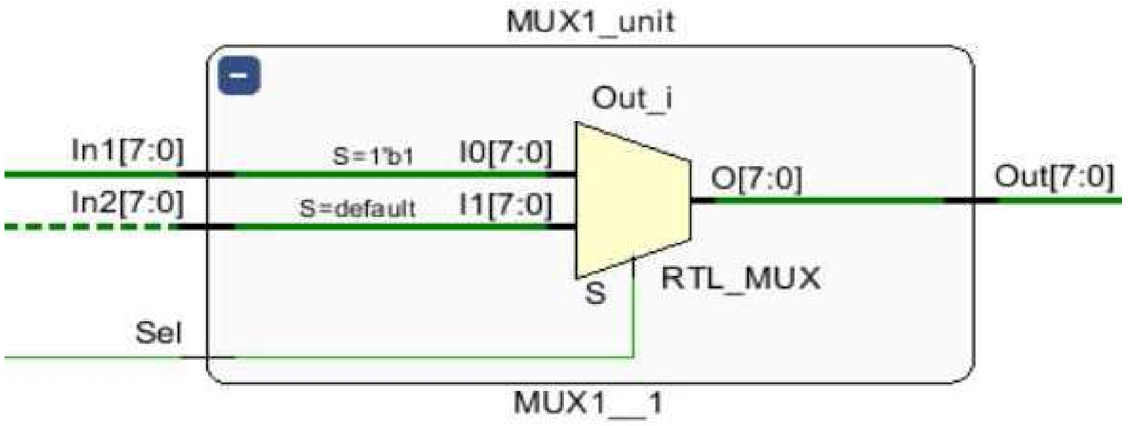


Fig 3.5: Implementation of Instruction MUX

3.7: ALU

ALU is used to do the actual computation for the current instruction. This component is pure combinational. The Arithmetic-Logic Unit was the first designed block. It works doing 8-bit and bit-oriented operations between the accumulator register W and another data that can come from of RAM or the instruction register. It has a carry (c) input for arithmetic and rotating operations, a 4-bit selector for the ALU to decode 18 operations; output flags: carry (c), digit carry (dc) and zero (z), which go to the STATUS register. It has the following port. The mode of ALU is listed in the following table.

ALU operand 1 port (8 bit, input, denoted as ALU.Operand1): connected to Acc.
ALU operand 2 port (8 bit, input, denoted as ALU.Operand2):connected to MUX2.Out.
ALU enable port (1 bit, input, denoted as ALU.E): connected to control logic.
ALU mode port (4 bit, input, denoted as ALU.Mode): connected to control logic.
Current flags port (4 bit, input, denoted as ALU.CFlags): connected to SR.
ALU output port (8 bit, output, denoted as ALU.Out): connected to DMem.DI.
ALU flags port (4 bit, output, denoted as ALU.Flags): the Z (zero), C (carry), S (sign), O (overflow) bits, from MSB to LSB, connected to status register.

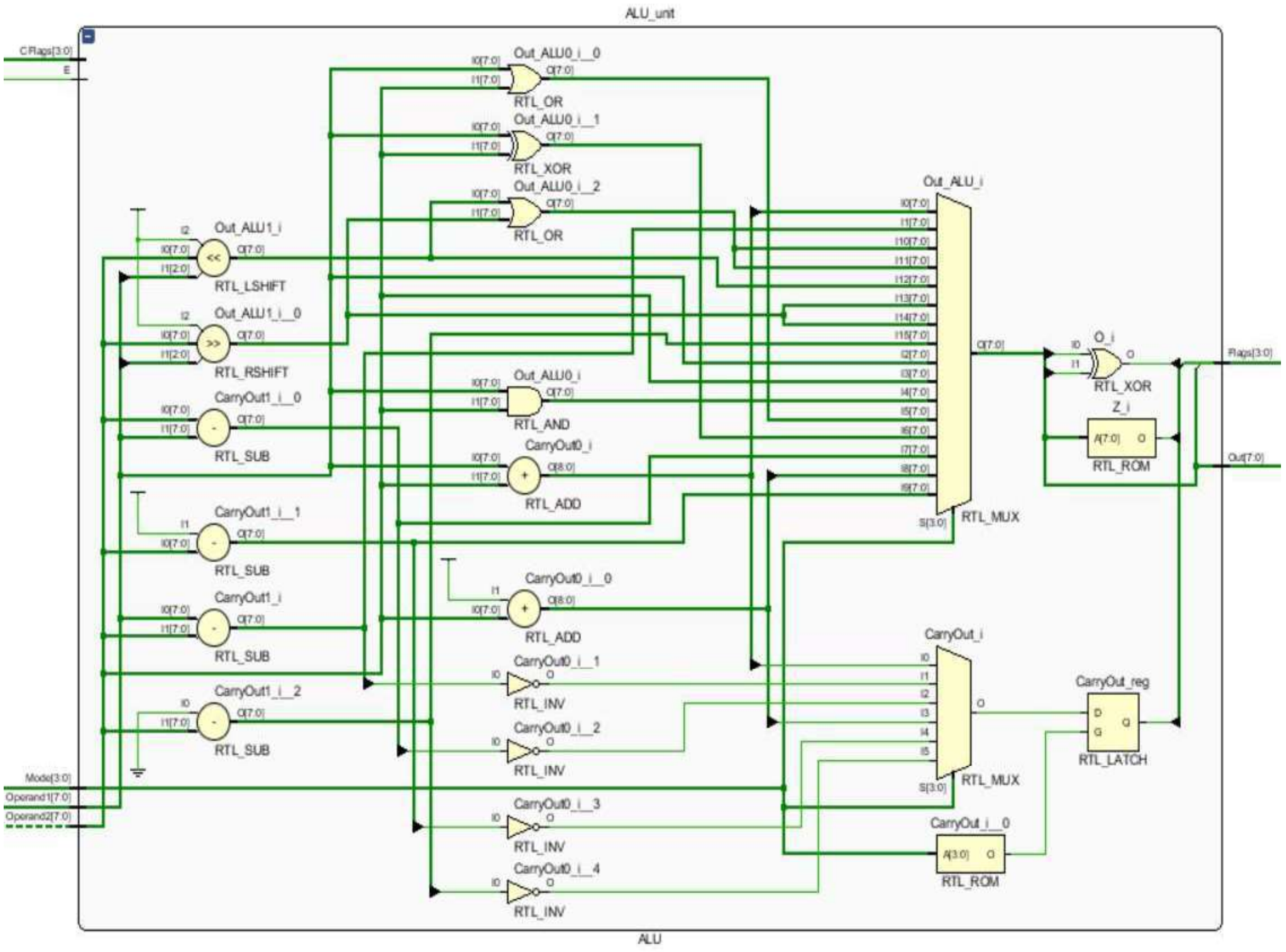


Fig 3.6: Implementation of ALU

3.8: CONTROL LOGIC UNIT

The Control Unit is a big Finite State Machine (FSM) that decodes the 14-bit operation code. It controls all the blocks of the system in order to execute the microinstructions of each instruction. Control signal is derived from the current state and current instruction. The control logic component is purely combinational. There are in total 12 control signals, listed as follows.

- PC.E: enable port of program counter (PC);
- Acc.E: enable port of accumulator (Acc);
- SR.E: enable port of status register (SR);
- IR.E: enable port of instruction register (IR);
- DR.E: enable port of data register (DR);
- PMem.E: enable port of program memory (PMem);
- DMem.E: enable port of data memory (DMem);
- DMem.WE: write enable port of data memory (DMem);
- ALU.E: enable port of ALU;
- ALU.Mode: mode selection port of ALU;
- MUX1.Sel: selection port of MUX1;
- MUX2.Sel: selection port of MUX2;

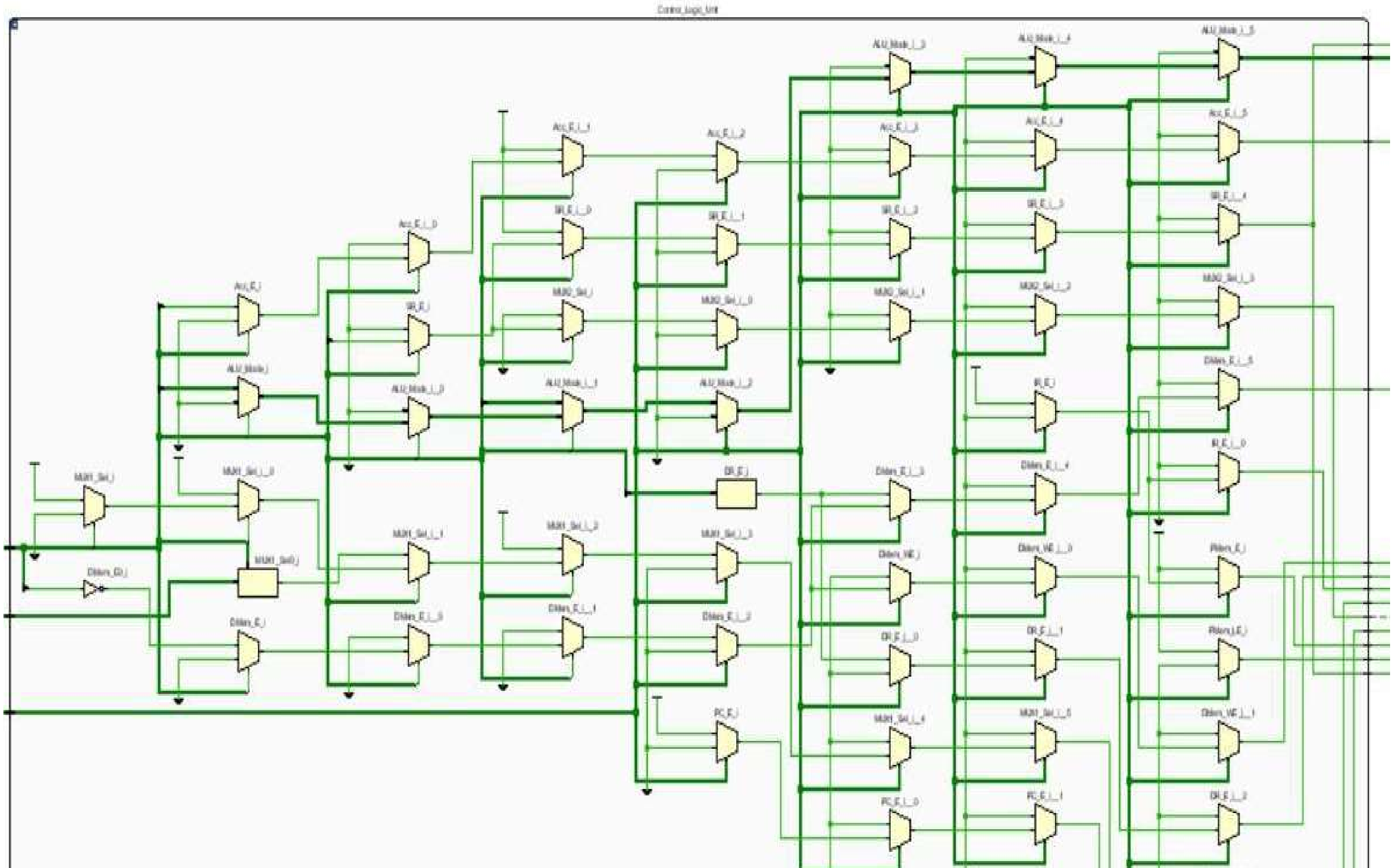


Fig 3.7: Implementation of Control Logic Unit

CHAPTER 4: IMPLEMENTATIONAND RESULTS

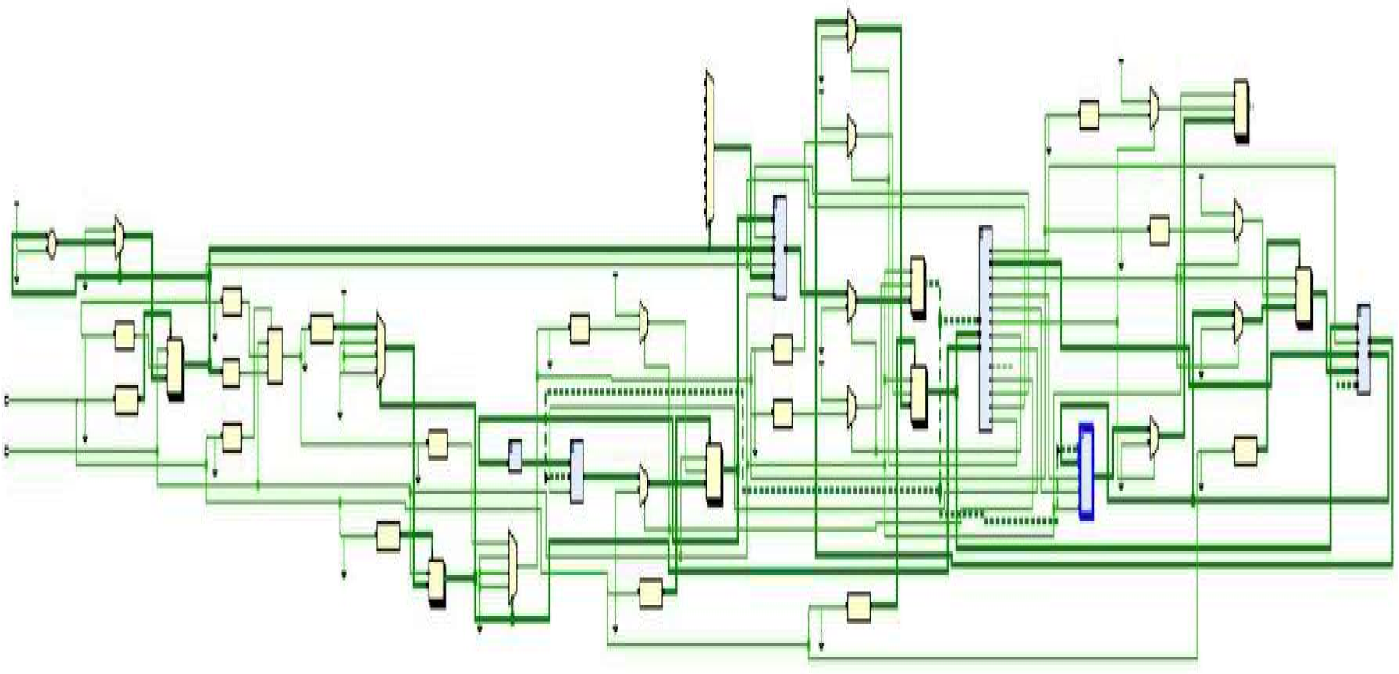


Fig 4.1: Implementation of complete microcontroller

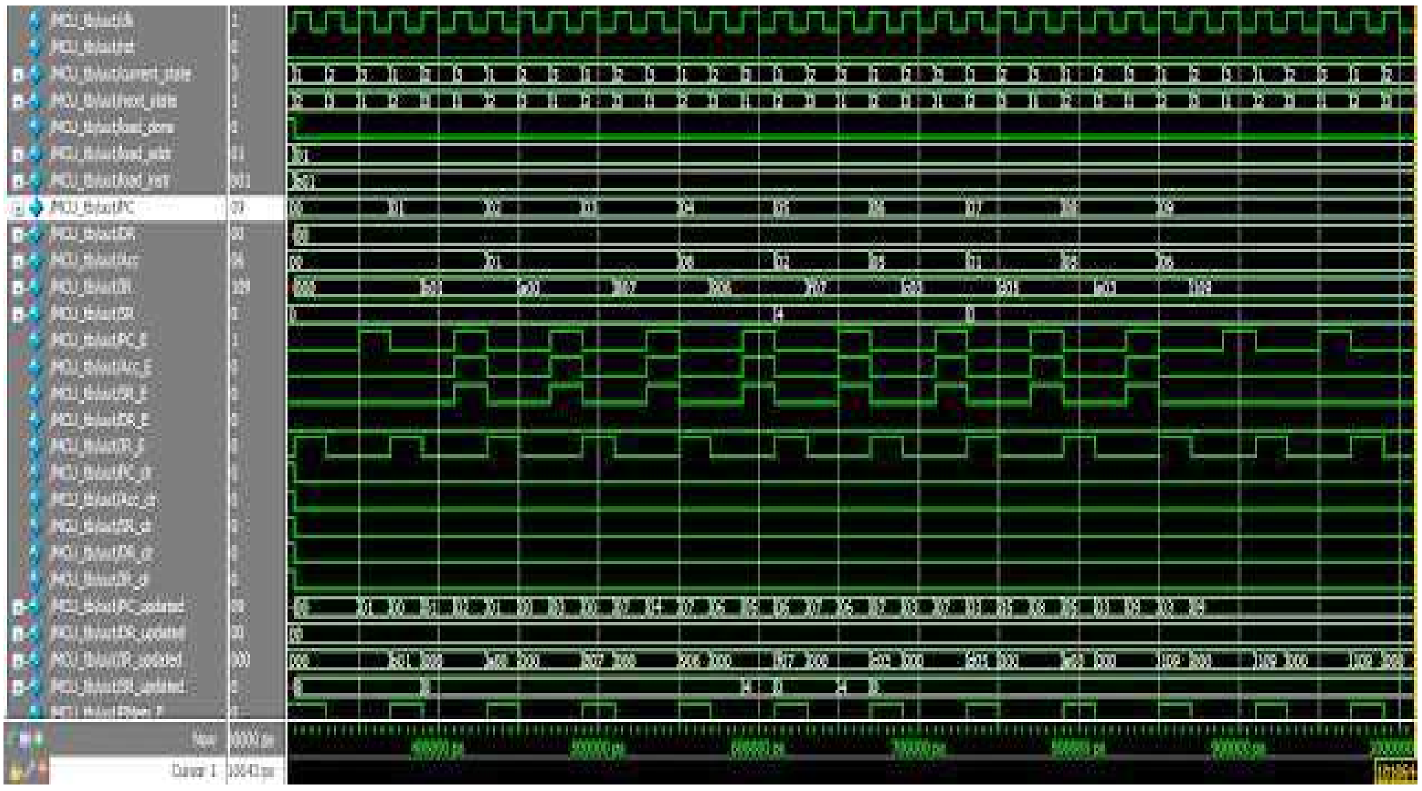


Fig 4.2: Timing Diagram for addition of two 8-bit operands using FPGA

CHAPTER 5: **ADVANTAGES OF IMPLEMENTATION**

1. Microcontrollers built using FPGAs can be re-programmed for more operations and to handle more tasks, as compared to conventional microcontrollers.
2. Any programming errors in the FPGA can instantly be removed by reprogramming, thus reducing time and wastage of money
3. Quantity of hardware required will be less leading to less space requirement ad better efficiency.
4. Due to parallel tasking and workflow, the microcontroller can be faster than a conventional microcontroller.
5. Unlike ASIC which are fixed once programmed, FPGAs are programmable at software level at any time. Hence FPGA IC can be re-programmed or reused any number of times. FPGA can also be programmed from remote locations.[1]
6. FPGA ICs are readily available which can be programmed using HDL code in no time. Hence the solution is available faster to the market.[1]
7. Unlike ASIC which requires huge NRE (Non Recurring Expenses) and costly tools, FPGA development is cheaper due to less costly tools and no NRE.
8. In FPGA design, software takes care of routing, placement and timing. This makes lesser manual intervention. The design flow eliminates complex and time consuming place and router, floor planning and timing analysis.[4]

CHAPTER 6: CONCLUSION:

There is in this project, without a doubt, an exploitation of the standard architecture that promises a large compatibility and the use of robust compilers that includes a lot of devices and libraries widely tested and certified. Additionally, communities on internet gives support to these kind of projects, thanks to the use of open code and free distribution of microcontroller design projects. Particularly, this project along with all its source code and documentation was shared in opencores.org website in order to be used or modified by anybody who needs it. The design of this type of architecture gives the versatility of managing the memory capacity because the dedicated blocks are easily scalable and modifiable from the same VHDL description.

On the other hand, the obtained result spends less resource within the FPGA, thanks to an optimization done developing combinatory blocks as the ALU, which has standard features. Like-wise, as the program memory as the data memory of the microcontroller have been mapped, in only one RAM block of hardware, predefined in the FPGA. The methodology of this microcontroller guaranties a parallel working with another type of microcontrollers or hardware modules of the same type, in order to be able to use in multitasking mode. This feature plus its reduced size, makes it possible to implement a lot of different microcontrollers in the same FPGA. According to the amount of logic gates of current FPGAs, one design can contain even hundreds of these microcontrollers. Under no circumstances, the objective of this project has been to replace a microcontroller; instead of this, this offers to the designer some design advantages of using Soft Core microcontrollers (under known and tested architectures), and encourage them to include these devices in their FPGA designs.

At an academic level, this project can be reused as tool for teaching some of the themes of the basic courses or even in advanced levels of subjects related with the digital design. With this system, the students can reach to comprehend of didactic way the behavior of the computational systems and like-wise include them in their own practice designs, taking advantage of FPGA development boards, which have well capacities and integrated peripherals. In the case of designing prototypes, adapted FPGAs can be used, with versatile connection interfaces.[5]

CHAPTER 7: BIBLIOGRAPHY

[1] https://www.fpga4student.com/2016/11/verilog-hdl-implementation-of-micro.html
[2]. https://www.fpga4student.com/2016/11/verilog-microcontroller-code.html
[3]. https://www.fpga4student.com/2016/11/verilog-code-for-microcontroller.html
[4]. https://learn.pantechsolutions.net/
[5]. https://codilime.com/blog/fpga-programming-how-it-works-and-where-it-can-be-used/