Journal of Engineering and Development
www.jead.org
Vol. 20, No.02, March 2016
ISSN 1813-7822

MIPS CPU DESIGN AND IMPLEMENTATIONBASED CYCLONE II FPGA BOARD

Dr. Ibtesam R. K.Al-Saedi*

Assist Prof., Communication Engineering Department, University of Technology, Baghdad, Iraq.

(Received:23/4/2015; Accepted: 16/09/2015)

Abstract: The aim of this work is to design and implement a simple MIPS CPU by using Cyclone II FPGA without complex control unit. MIPS- Processor has been studied and realized to design, simulate and implement its components by using VHDL and FPGA Board under Quartus version "8.1 software packages. This design in a modern FPGA environment has done and used to finally realize the hardware components (RF, PC, ALU, RAM, ROM and Multiplexer). Also, the design has used a module principle to implement the components of the Microprocessor which provides high flexibility in expanding the hardware and software units of its components when there is a need to change the structure of the design without a complex central control unit. This method of design provides high flexibility especially for embedded systems which are planted in a variety of applications. The success of the design has been tested through the work of the processor as an integrated in all components under its instructions with simple control unit.

Keywords: *MIPS, Computer Architecture, VHDL, Simulation*

تصميم وبناء معالج مبس مبنيا على نسق مصفوفة البوابات المبرمجة نوع سايكلون2

الخلاصة:الهدف من هذا العمل تصميم وبناء نوع من المعالجات الدقيقة MIPS(مليون ايعاز في الثانية) نوع RISC (الكومبيوترات ذات ايعازات مقننة) بدون وحدة سيطرة معقدة باستخدامCyclone II FPGA Board. تم دراسة، تصميم، محاكاة وبناء هذا النوع من المعالجات الدقيقة باستخدام برمجيات محاكاة المكونات المادية VHDL ولجميع مكونات المعالج وبناء ايعازاته باستخدام منصة المحاكاة البرمجية المعروفة لهذا النوع من البرمجة 8.1Quartus.صمم معالج المبس في بيئة حديثة لل FPGAوببرنامج المحاكاة الذي في نهايتة تم إدراك المكونات المادية لوحدات المعالج الأساسية(RF, PC, ALU, RAM, ROM و Multiplexer) باستخدام مبدأ الوحدات التركيبية (Module) التي استخدمت في التصميم و البناء حيث وفرت مرونة عالية عند تغيير او توسيع بنية التصميم او عند توسيع الوحدات المادية والبرمجية لمكونات المعالج حسب المهام المطلوبة ودون استخدام وحدة سيطرة مركزية معقدة . هذه الطريقة في التصميم ومن خلال توفير مرونة عالية في توسيع مهام محددة تجعلها مناسبة للانظمة المطمورة او والمزروعة Embedded Systems و لتطبيقات متنوعة . تم اختبار نجاح التصميم من خلال عمل المعالج كوحدة متكاملة بجميع مكوناته تحت جميع الايعازات بوحدة السيطرة البسيطة.

1. Introduction

One of the important abstractions that a programmer uses is to instruction set

* drkarhiy@yahoo.com

architecture (ISA). MIPS microprocessor (Million Instruction Per Second) measures of microprocessor speed that uses 32 bit wide registers and data paths. The MIPS CPU is one of the RISC(Reduced Instruction Set Computer) CPUs, born out of a particularly fertile period of academic research and development. In general, RISC processors typically support fewer and much simpler instructions [1, 2].

Today, most of the MIPS parts that are shipped go into some sort of embedded application. MIPS Technologies is a provider of synthesizable, licensable 32-bit processor cores offered with a range of features and capabilities that address diverse market segments including home entertainment (e.g. DTV and set-top boxes), home networking (e.g.xDSL and WiFi), personal entertainment (e.g. digital cameras and portable media players) and microcontrollers (MCUs). MIPS also licenses its 32- and 64-bit architectures to system-on-chip (SoC) developers [3].

The efficient of design always is how to make the process less complexity especially when there is a need for a specific task. A single purpose processor is this type of processor that can be designed to execute exactly one program. An embedded designer creates a single-purpose processor by designing a custom digital circuit, advantages and disadvantages are more or less the opposite of the general-purpose processor and they are used for in a wide variety of applications for the same task [4].

Nowadays hardware is more and more like software. It can easily be programmed and integrated with other components by using a modular design. This means that hardware components are designed as separate parts and can be put together in one device. In this way the device is only programmed for the control and communication of the different components. Also, this way has many advantages like components can easily be replaced when broken and adding new functions (upgrading) is very simple.
However, there are some kind of a pioneer tools or standardization to reach for low design cost and fast product development like VHDL (hardware description language) and FPGA (field programmable gate array).

A FPGA has many more components than it should have so in that way it can be programmed for various applications. It can also be used as an interface for the communication of several hardware components [5, 6].

2. Cyclone II FPGA Starter Board

Although microprocessors can be implemented in an FPGA design, it is great choice to use some tools can offer many facilities such as density logic inside a single chip and reprogrammable of the design.

In reprogramming feature, the design can be changed after the hardware has been manufactured and the high level design software optimized the usage of limited resources.

In this work, the Cyclone II FPGA Starter Development Board (Fig. 1) has chosen to provide integrated features that enable users to develop and test designs, range from simple circuits to various multimedia projects, all without the need to implement complex application programming interfaces (APIs), host control software, or SRAM/SDRAM/flash memory controllers..

To implement the design with Cyclone II FPGA, it is necessary to understand basic design steps about Quartus board by using schematic editor and HDL, compiling the design, pin assignment, and downloading the design into the FPGA board [7, 8].

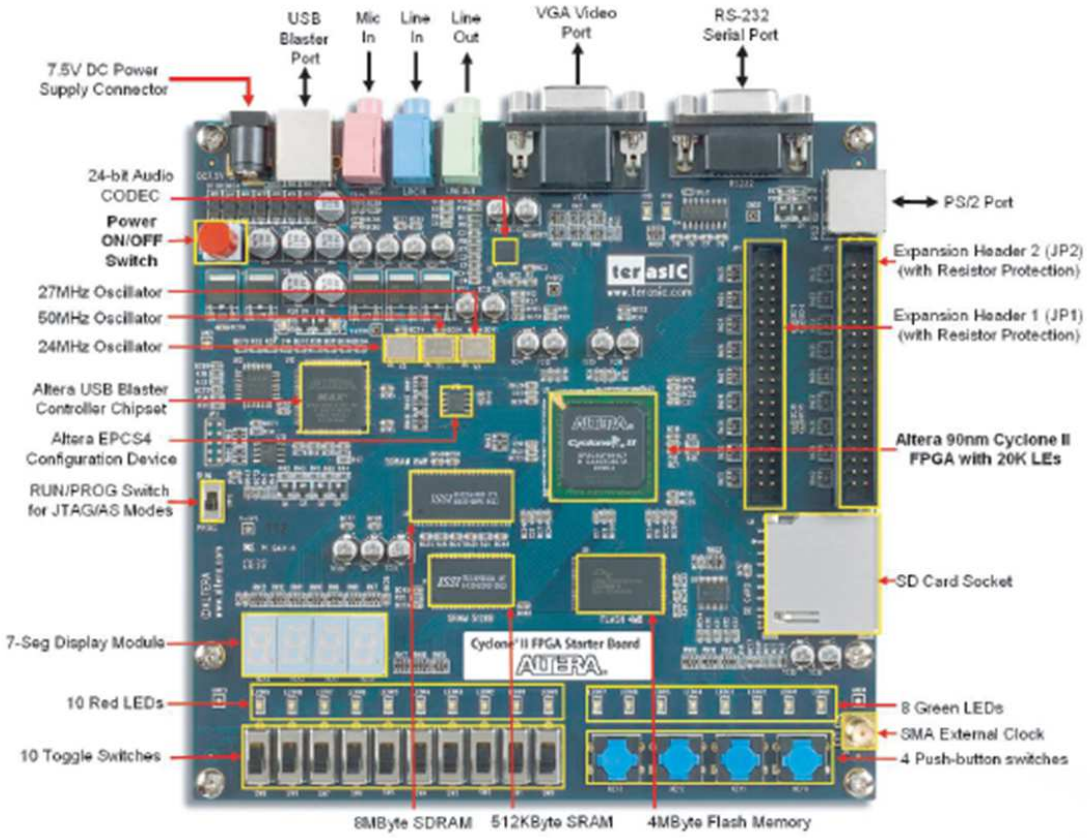

Figure 1. Cyclone II FPGA Starter Development Board[7]

3. Design Architecture

The Architecture for MIPS CPU design is a simple a RISC processor. The design is implemented with the concepts of the modules which represent here five components (Memory, Register File (RF), Arithmetic Logic Unit (ALU), Programming Counter (PC) and Multiplexes). The advantage of this design is ability to recognize both hardware and software in simple way of construction. The block diagram in Fig.2 describes the architecture for designed CPU.

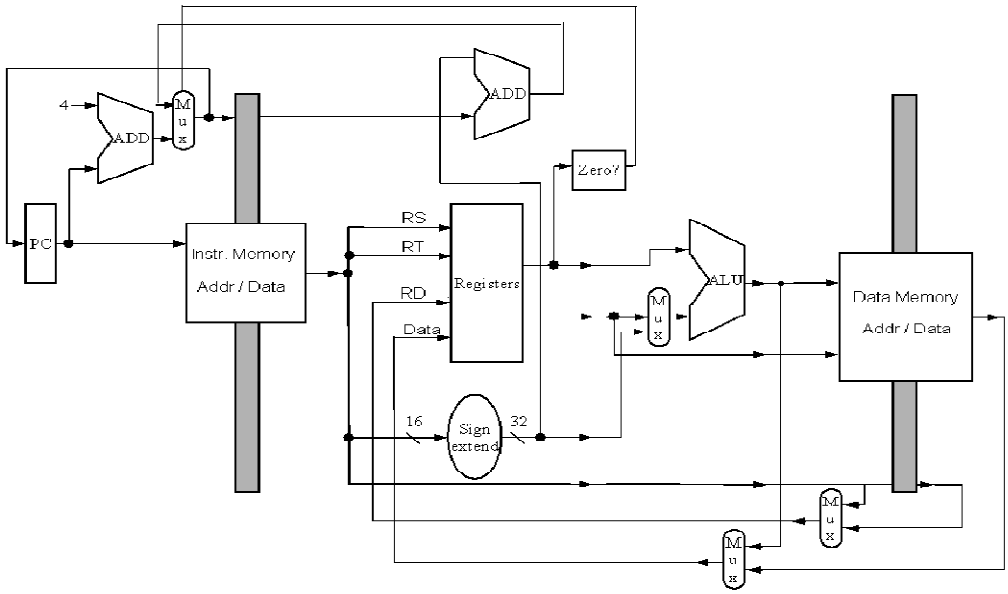

Figure 2: Architecture Block Diagram [9]

The standard design flow has applied for each module as well as for the final CPU which includes: Design entry (Schematic entry and High level language), Synthesis (Translating design into logic elements), Simulation (Validate design logic) and Fitting (Implementing logic using FPGA resources). The design has implemented under platform Quartus version "8.1 "software package [10].

2.1 Component Descriptions for CPU Designed

Memory: This memory is divided into two parts:

1) ROM holds the instructions. It has 2 inputs address [8..0] &clk and 1 output q [31..0].
2) RAM holds data. It has 4 input [address[8..0] , clk, data[31..0] and Wren }and 1 output.

***Register File (RF)*:** This structure contains all of the registers numbered #0- #31. 32bit register used to store the output of the memory (instructions and Data). It has 8 inputs [clk, rst, opcode, rsAdd, rtAdd, rdAddr, dataIn and dataLoad] and 2 outputs with the ability to see the contents of any register. The loading of this register file is controlled directly by the program.

ALU: The core of the MIPS CPU has 7 inputs [clk. rst, opcode, funct, rs, rt and imm Value] and 7 outputs [aluOut, WE,Zero, Carry, equal. Greater Than and Less Than]. A zero flag is set if the value of the result [32 bits], another output, is all zeros. The ALU can perform a number of arithmetic operations, such as add and subtract, and some of logical operations, such as AND, OR and XOR. The opcode selects the operation directly.

PC Counter: Usually the program is executed in the straight line fashion; the next instruction to be executed is the one that follows the previous one currently being

executed. But sometimes, it is needed to conditionally or unconditionally jump to some other part of the program (e.g., functions, loops, etc.) by the program control instructions. This module has 10 inputs [opcode, addr, immaddr, zero, carry, EQ, GT,LT, clk, reset and 1 output PCout.

Multiplexes: This module connects the data pathway to check the output on the same nodes output. This module has been created to check the output for each cycle, depending on the current instruction.

The components RF, PC, ALU, RAM, ROM and Multiplexer are programmed by using VHDL and FPGA Board under Quartus version "8.1.[10]. ALU was more complex programing part in whole design because it represents the core of process that need a special attention (Appendix).

2.2 MIPS Instruction Set Descriptions with the CPU Designed [11]

There are several distinct "classes" of instructions which describes its instructions sets:

a. Arithmetic/logical/shift/comparison
b. Load/store
c. Branch
d. Jump

Also, there are three instruction formats (encoding) for MIPS

a. R-type (6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit shamt, 6-bit function code)
b.

31-26	25-21	20-16	15-11	10-6	5-0
opcode	*rs*	*rt*	*rd*	shamt	function

c. I-type (6-bit opcode, 5-bit rs, 5-bit rt, 16-bit immediate)
d.

31-26	25-21	20-16	15-0
opcode	*rs*	*rt*	*imm*

e. J-type (6-bit opcode, 26-bit pseudo direct address)
f.

31-26	25-0
opcode	*pseudodirect jump address*

In this work, the instructions groups of the CPU that adopted in this design are explained in Table 1.

Table 1.Describtionof the instructions groups [9]

Name	Description	Opcode 6 bits	RS 5 bits	RT 5 bits	RD 5 bits	SA 5 bits	Funct 6 bits
Logic Instructions Register-Type							
OR		010111	Src	Src	Drc		000000
AND		010111	Src	Src	Drc		000001
XOR		010111	Src	Src	Drc		000010
XNOR		010111	Src	Src	Drc		000011
Arithmetic Instructions Register-Type							
SHL	Shift left	010000	Src	Src	Drc		
SHR	Shift right	010001	Src	Src	Drc		
ROL	Rotate left	010010	Src	Src	Drc		
ROR	Rotate right	010011	Src	Src	Drc		
SUB	Subtract	010100	Src	Src	Drc		
ADD	Addition	010101	Src	Src	Drc		
Arithmetic Instructions Immediate-Type							
SHLI	Shift left	011000	Src	Drc	i		
SHRI	Shift right	011001	Src	Drc	i		
ROLI	Rotate left	011010	Src	Drc	i		
RORI	Rotate right	011011	Src	Drc	i		
SUBI	Subtract	011100	Src	Drc	i		
ADDI	Addition	011101	Src	Drc	i		
Branch Instructions							
JMP	Jump	100000	address				
BEQ	Branch equal	100010	Src	Src	i		
BNEQ	Branch not equal	100011	Src	Src	i		
BGT	Branch greater than	100100	Src	Src	i		
BLT	Branch less than	100101	Src	Src	i		
BC	Branch on carry	100110			i		
BZ	Branch zero	100111	Src		i		
Memory Instructions Register-Type							
LDD		111000	Src	Drc	i		
STO		111001	Src	Src	i		
Other Instructions							
NOP		000000					

Notes: Src is a source register, Drc is the destination register, 'i' is a 16 bit constant, "address" is a 24 bit constant.

2.3 Cycle description for CPU Designed

Each of the three cycles must perform certain tasks. Once completed, the results of the cycle's calculations are stored in registers or in memory depended on the program. The basic duty of each cycle is as follows:

1. **Instruction Fetch –** This cycle is the same for all instructions, simply program counter (PC) sends the address to the memory (ROM) to fetch the instruction, put it in the Register File (RF) and increment the PC.
2. **Instruction Decode and Register Fetch –** To minimize the time of instruction decode, it is more efficient to perform tasks by participation between ALU and

RF. Therefore, to decoding the instruction, the registers rs and rt are read and stored send to ALU. These measures reduce the maximum number of clock cycles. This operation is the same for all instructions too.

3. **Execution –** In this operation of the CPU is determined by the particular instruction that is being executed. This operation been performance by participation between ALU, RF and Memory according to the following possibility:
   - Arithmetic-logical instruction – ALUOut<= rs op rt.
   - Branch or a Jump – if (Zero, Carry , Equal, Grater Than , Less Than), PC <= ALUOut.
   - Memory Events (read or write) – Depend on the Instruction which can be in two state : if the instruction is a load, a data word is retrieved from memory directly and written into DataMemOut of IR. Or if the instruction is a store, the ALUOut put in DataIn of RF.

3. VHDL Coding for CPU Designed

According to the Modules listed above, VHDL code was written to perform the combined functions in as simplistic form as possible. Each module has been as a stand alone component. The Source VHDL Coding of the modules PC, RF and ALU were written according to the function description for each one.

This design used Quartus8.1 Software under windows as a platform to write the VHDL programs for the modules PC, RF and ALU. By using the facilities like MegaWizard plug In_Managr which is available in Quartus8.1; it was easy to build the RAM. ROM and Multiplexer Modules. The designs have been tacked account all the possible states to avoid the overlapping during the excision of the instructions and minimize the hardware.

4. Simulating the designed CPU

Before implementing the design Module in the FPGA chip on the DE board, it is deserve to simulate it to ascertain its correctness. Quartus8.1 software includes a simulator which can be used to simulate the behaviour and performance of units designed for implementation in Altera's programmable logic devices.

The simulator allows the user to apply test vectors as inputs to the designed circuit and to observe the outputs generated in response. In addition to being able to observe the simulated values on the I/O pins of the unit, it is also possible to probe the internal nodes in the circuit. The simulator makes use of the Waveform Editor which makes it easy to represent the desired signals as waveforms. This work presents the use of VHDL to simulate of simple RSIC microprocessor in the level synthesis system.

This work is depended on the principle of module design that makes design more flexible when they need any extension for some facilities later. After design and simulating for each unite RF, ALU, PC, RAM, ROM and Multiplexer, It was implemented the Block diagram for the components of CPU. Fig. 3 shows the details connection for the design according the facilities which are available in Quartus 8.1.

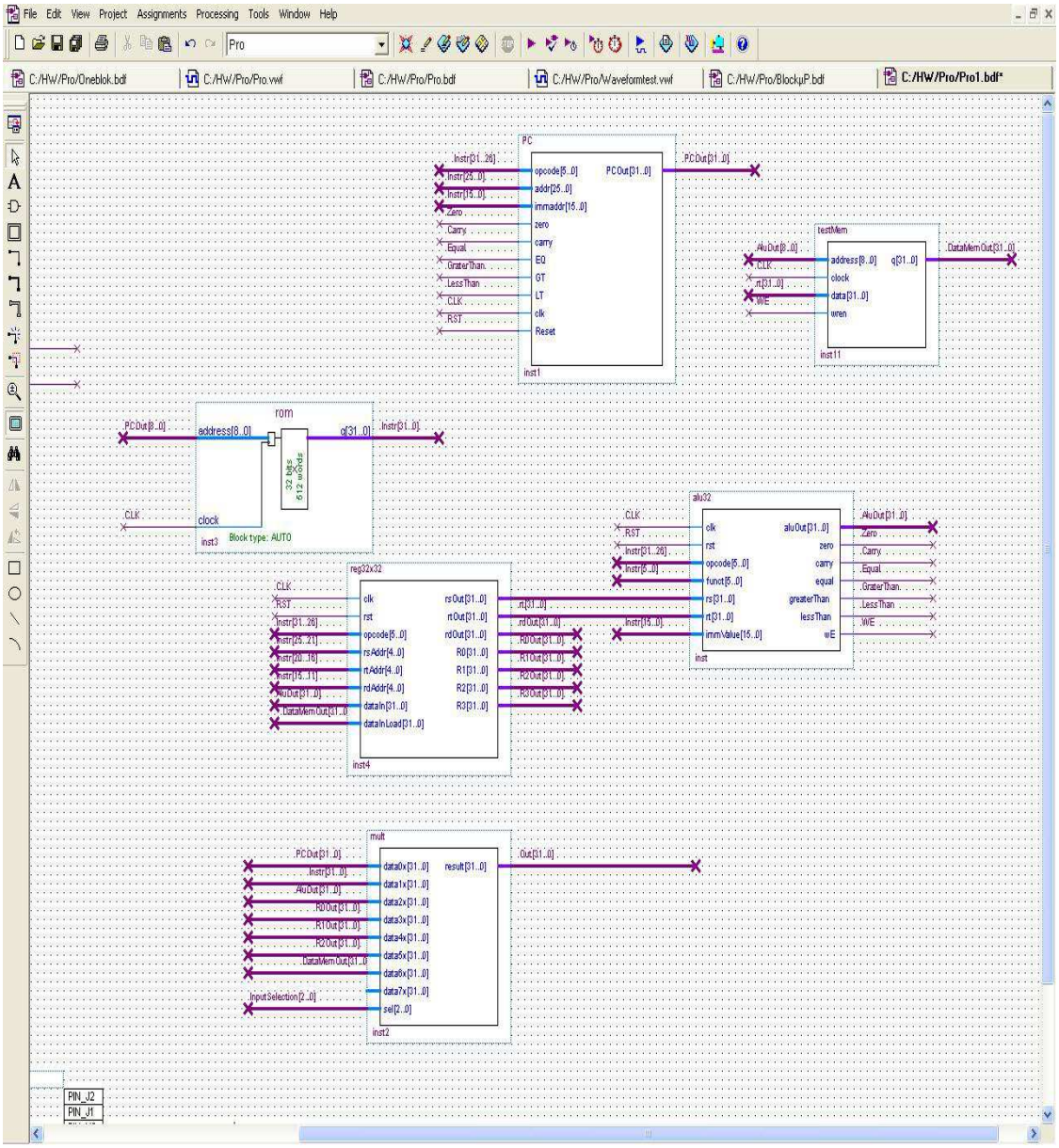

Figure 3: Block Diagram for final CPU designed which implemented and programed its components under Quartus 8.1.

5. Results of Simulation

After succeed of the coding operation for the block designed is processed by several Quartus II tools that analyze the code, synthesize the design circuit and generate an implementation of it. These tools were performed by compiler operation. The report of compilation operations showed in table 2.

Table 2.The report of compilation operation

Flow Status	Successful - Fri Nov 14 14:50:17 2008
Quartus II Version	8.0 Build 215 05/29/2008 SJ Web Edition
Revision Name	Pro
Top-level Entity Name	Pro1
Family	Cyclone II
Device	EP2C20F484C6
Timing Models	Final
Met timing requirements	N/A
Total logic elements	2,529
Total combinational functions	2,529
Dedicated logic registers	1,063
Total registers	1063
Total pins	269
Total virtual pins	0
Total memory bits	32,768
Embedded Multiplier 9-bit elements	0
Total PLLs	0

6. Simulation and Waveform

The design can be simulated in two ways. The typical way is a functional simulation which is used to verify the function correctness of the circuit as it is being designed. This operation tasks much less time, because the simulation can be performed simply by using the logic expression. But timing simulation is more complicated because it takes all propagation delays into account. It is necessary to create the required enlist before running the function simulation. Fig. 4 illustrates the result of timing simulation for the CPU.

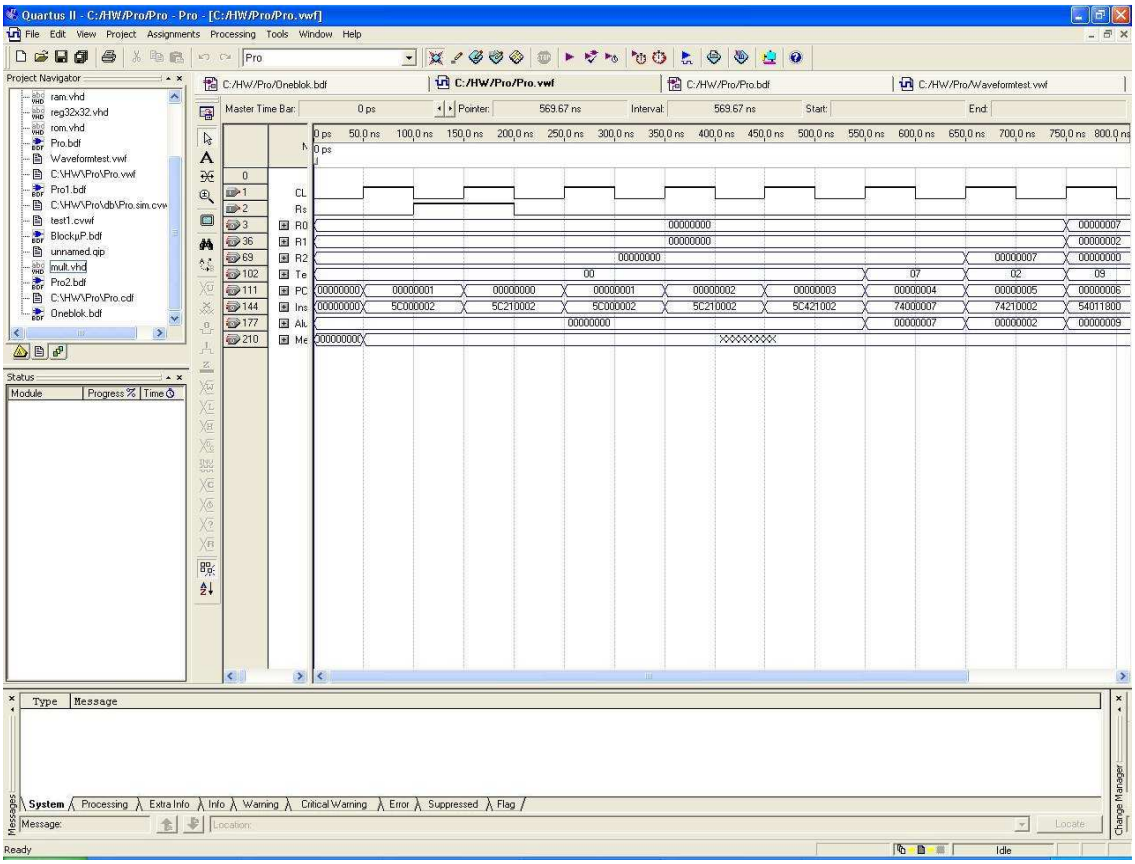

Figure 4.The result of timing simulation waveform shows the behaviour of the design under normal conditions.

The result of simulation illustrated that the design performed the factions (execute the test program) which is written as following below:

Memory.mif:

WIDTH=32;

DEPTH=512;

ADDRESS_RADIX=HEX;

DATA_RADIX=HEX;

CONTENT BEGIN

000 : 5C000002; XOR R1,R1,R1

001 : 5C210002; XOR R2,R2,R2

002 : 5C421002; XOR R3,R3,R3

003 : 74000007; Addi R1,R1,7

004 : 74210002; Addi R2,R2,2

```
    005 : 54011800;    Add   R3, R1, R2

    006 : E4010000;    STO R1,R2,0

007 :00000000;

008 :00000000;

009 :00000000;

00A :00000000;

00B :00000000;

00C :00000000

01A : 00000000;

    .

    .

    256: 00000000;

END;
```

7. Conclusions

The architecture for this CPU here is very straight forward and easy to implement. This relatively simple design was chosen to recognize the principle of RISC processer based FPGA . Given more applications for this design would be easier extended the design without changing the architecture and without a need for complex Control Unit. However, the VHDL coding of such a design would, we felt, take more time than what we think if there is not idea about the programming of VHDL under modern platform software package. The more important advantages for like this design that it will be not a need a complex control unit and easy to extend its applications especially with specific applications for embedded systems.

One of the important reason actually in the successful of final this work is the way of ALU architecture has been programmed and connected.

8. References

1. John P. Hayes,(1998) 'Computer architecture and Organisation', Tata McGraw-Hill, Third edition,.
2. Dominic Sweetman,(2006)"See MIPS run",2nd Edition ,ISBN-9780120884216, Printbook ,Release Date:.
3. MIPS TECHNOLOGY , 'An Introduction to the MIPS32® M14Kc™ Processor Core MD00689', Revision 01.00 October 2009 MIPS Technologies, Inc.955 East Arques Avenue Sunnyvale, CA 94085 (408) 530-5000 © 2009 MIPS Technologies, MIPS by imagination, https://www.mips.com/products/product-materials/whitepapers/.

4. Embedded Control Systems Design/ Processors, http://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Processors#single_purpose_processor.
5. Peter J. Ashenden,(1990)" The VHDL CookBook", 1St Edition , Dept. Computer Scince, University of Adelaide, July,.
6. VHDL, Verilog, and the Altera environment Tutorial, http://www.ece.tufts.edu/~hchang/ee129-f06/project/project2/Tutorial.pdf
7. Cyclone II FPGA Starter Development Board Reference Manual, ALTERA, document version1.0, document data October 2006. http://www.altera.com
8. Tutorial of ALTERA Cyclone II FPGA Starter Board htt://web.cecs.pdx.edu/~greenwd/Tutorial-altera-cyclone-board.pdf
9. .Prof Dr. Bernd Becker & Prof Dr. Paul Molitor,(2008)" Technische Informatik", Eine einführede Darstellung, Oldenbourg Verlag München Wien, Deuchland,.
10. Quartus II Simulation with VHDL Designs ftp://ftp.altera.com/up/pub/Tutorials/DE2/Digital_Logic/tut_simulation_vhdl.pdf
11. Chapter 3: MIPS Instruction Set Review ,www.cs.gmu.edu/~setia/cs365-S02/lec3.pdf

Appendix (ALU code)

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

---------------------------------------------------------------------------------

-- Definition of my 32 bit alu

entity alu32 is port(

   -- Inputs

clk, rst : in std_logic;

opcode : in std_logic_vector(5 downto 0);

funct : in std_logic_vector(5 downto 0);

rs, rt: in std_logic_vector(31 downto 0);

immValue: in std_logic_vector(15 downto 0);

   -- Outputs

aluOut : out std_logic_vector(31 downto 0);

zero, carry, equal, greaterThan, lessThan : out std_logic;
```

```
wE : out std_logic);
end alu32;
architecture Behavioral of alu32 is
---------------------------------------------------------------------------------
-- Signal Declorations for Logic and Arithmatic
signaliRT          : std_logic_vector(31 downto 0);
signalinternalXORRT : std_logic_vector(31 downto 0);
signal subtract32    : std_logic_vector(31 downto 0);
signalimmValueExt   : std_logic_vector(31 downto 0);
signalaluResult     : std_logic_vector(32 downto 0);
signallogicResult   : std_logic_vector(31 downto 0);
signalzeroVec       : std_logic_vector(31 downto 0);
signalsumOut : std_logic_vector(32 downto 0);
---------------------------------------------------------------------------------
-- Signal Declorations for Rotating
signal rotateL16, rotateL8, rotateL4, rotateL2, rotateL1 : std_logic_vector(31 downto 0);
signal rotateR16, rotateR8, rotateR4, rotateR2, rotateR1 : std_logic_vector(31 downto 0);

---------------------------------------------------------------------------------
-- Signal Declorations for Shifting
signal shiftL16, shiftL8, shiftL4, shiftL2, shiftL1 : std_logic_vector(31 downto 0);
signal shiftR16, shiftR8, shiftR4, shiftR2, shiftR1 : std_logic_vector(31 downto 0);
begin
---------------------------------------------------------------------------------
-- Decide what results really is with the opcode
with opcode(2 downto 0) select aluResult(31 downto 0) <=
shiftL1     when "000",
shiftR1     when "001",
rotateL1    when "010",
rotateR1    when "011",
```

```
logicResult when "111",
sumOut(31 downto 0) when others;
-- All the logical functions here
withfunct(5 downto 0) select logicResult<=
rs OR rt   when "000000",
rs AND rt  when "000001",
rs XOR rt  when "000010",
rs XNOR rt when others;

-- Alu result for the carry flag
aluResult(32) <= sumOut(32) when opcode(2 downto 1) = "10" else '0';
-- Define aluOut
aluOut<= aluResult(31 downto 0);
--------------------------------------------------------------------------------
-- These are the flags for branching
zero<= '1' when rs = X"00000000" else '0';
equal<= '1' when rs = rt else '0';
greaterThan<= '1' when rs>rt else '0';
lessThan<= '1' when rs<rt else '0';
-- This is for the carry flag. It is held for one cycle
carryFlag:process(clk)
begin
ifclk'event and clk = '1' then
ifrst = '1' then
carry<= '0';
else
carry<= aluResult(32);
end if;
end if;
end process;
```

```
---------------------------------------------------------------------------------
-- Calculate if we are using rt or the Immediate value
immValueExt(31 downto 16) <= (others => (immValue(15)));
immValueExt(15 downto 0)  <= immValue;
iRT<= rt when opcode(3) = '0' else immValueExt;
---------------------------------------------------------------------------------
-- This is for the adder/subtractor
subtract32    <= (others => '1') when opcode(2 downto 0) = "100" else (others => '0');
internalXORRT<= iRT XOR subtract32;
sumOut<= ('0' &rs) + ('0' &internalXORRT) + subtract32(0);
---------------------------------------------------------------------------------
-- The next part here is for the shifting and rotating.
-- A zero vector used for shifting
zeroVec<= (others => '0');
-- Rotating and Shifting
-- Rotate Left
rotateL16 <= (rs(15 downto 0) &rs(31 downto 16))             when iRT(4)='1' else rs;
rotateL8  <= (rotateL16(23 downto 0) & rotateL16(31 downto 24)) when iRT(3)='1' else rotateL16;
rotateL4  <= (rotateL8(27 downto 0) & rotateL8(31 downto 28))   when iRT(2)='1' else rotateL8;
rotateL2  <= (rotateL4(29 downto 0) & rotateL4(31 downto 30))   when iRT(1)='1' else rotateL4;
rotateL1  <= (rotateL2(30 downto 0) & rotateL2(31))            when iRT(0)='1' else rotateL2;

-- Rotate Right
rotateR16 <= (rs(15 downto 0) &rs(31 downto 16))            when iRT(4)='1' else rs;
rotateR8  <= (rotateR16(7 downto 0) & rotateR16(31 downto 8)) when iRT(3)='1' else rotateR16;
rotateR4  <= (rotateR8(3 downto 0) & rotateR8(31 downto 4))   when iRT(2)='1' else rotateR8;
rotateR2  <= (rotateR4(1 downto 0) & rotateR4(31 downto 2))   when iRT(1)='1' else rotateR4;
rotateR1  <= (rotateR2(0) & rotateR2(31 downto 1))            when iRT(0)='1' else rotateR2;
-- Shift Left
shiftL16 <= (rs(15 downto 0) &zeroVec(15 downto 0))      when iRT(4)='1' else rs;
```

```
shiftL8  <= (shiftL16(23 downto 0) &zeroVec(7 downto 0)) when iRT(3)='1' else shiftL16;
shiftL4  <= (shiftL8(27 downto 0) &zeroVec(3 downto 0))  when iRT(2)='1' else shiftL8;
shiftL2  <= (shiftL4(29 downto 0) &zeroVec(1 downto 0))  when iRT(1)='1' else shiftL4;
shiftL1  <= (shiftL2(30 downto 0) &zeroVec(0))           when iRT(0)='1' else shiftL2;
-- Shift Right
shiftR16 <= (zeroVec(15 downto 0) &rs(31 downto 16))     when iRT(4)='1' else rs;
shiftR8  <= (zeroVec(7 downto 0) & shiftR16(31 downto 8)) when iRT(3)='1' else shiftR16;
shiftR4  <= (zeroVec(3 downto 0) & shiftR8(31 downto 4))  when iRT(2)='1' else shiftR8;
shiftR2  <= (zeroVec(1 downto 0) & shiftR4(31 downto 2))  when iRT(1)='1' else shiftR4;
shiftR1  <= (zeroVec(0) & shiftR2(31 downto 1))           when iRT(0)='1' else shiftR2;
wE<= '1' when opcode = "111001" else '0';
end Behavioral;
```