Design of a 16-bit RISC Processor Using VHDL

Article *in* International Journal of Engineering and Technical Research · April 2017

DOI: 10.17577/IJERTV6IS040284

3 authors, including:

Karansingh Pramodsingh Thakor
Indian Institute of Technology Bombay
5 PUBLICATIONS **11** CITATIONS

SEE PROFILE

Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 04, April-2017

Design of a 16-bit RISC Processor Using VHDL

Karansingh P. Thakor, Ankushkumar Pal
Department of EXTC
Xavier Institute of Engineering
Mumbai, India

Prof. Madhura Shirodkar
Department of EXTC
Xavier Institute of Engineering
Mumbai, India

***Abstract*— This paper targets the design and implementation of a 16-bit RISC Processor using VHDL (Very High Speed Integrated Circuit Hardware Description Language). As IC chip design involves complex computations and intense usage of resources, by using an HDL we can save resources and time by implementing it using the software approach. The implementation strategies have been borrowed from the popular MIPS architecture to a certain extent. The processor has 16-bit arithmetic and logical instruction set which has been designed and simulated. The instruction set is extremely simple and it gives an insight into the kind of hardware that would be required to execute the instructions accordingly. The ALU, instruction register, program counter, register file, control unit and memory have been integrated in the proposed processor. All the modules in the design are coded in VHDL to ease the description, verification, simulation and hardware implementation. The blocks are designed using the behavioral approach.**

Keywords — *RISC Processor, MIPS, SPU, ALU, ISA, VHDL, PC, Opcode, clock, timing, FPGA, instruction, LOAD, JZ, JNZ, ADD, ALB, AGB, waveform, RTL Schematic, 16-bit, Xilinx ISE, Isim.*

I. INTRODUCTION

Processors are divided into 3 categories 8-bits, 16-bits and 32-bits depending upon the demand of performance, cost, power and programmability. 16-bit processors have higher performance and power than 8-bit processors and lower power consumption than 32-bit processors. They are often used in 16-bit applications such as disk driver controller, cellular communication and airbags. A RISC processor uses load-store architecture, fixed length instructions and pipelining. In load-store architecture, load instruction reads data from memory and writes it to a register, data-processing instructions process data available in registers and write the result to a register and store instruction copies data from register to memory.

This paper investigates the methodology of soft-core processor development. The software used was Xilinx ISE 14.5. The target family chosen was Spartan 6 FPGA with device XC6SLX9 and package CSG324. The simulation was performed in ISim.

II. INSTRUCTION SET

The first step was to design the Instruction Set Architecture (ISA). The instruction set contains instructions supported by the processor. The first nine instructions perform arithmetic and logical instructions. The memory read instruction reads the data from the memory address which is specified in a register (Rs) and writes the data word to the register mentioned in the instruction (Rd). The memory write instruction writes the data in the specified register (Rs) to the target address which is also specified in a register (Rt). There is an unconditional jump that jumps to a memory address that is calculated by adding the current value in the Program Counter and the 8-bit offset specified in the instruction. This gives the target branch address. A similar procedure is followed by the Conditional Jumps but the address is calculated beforehand as an optimistic operation and is stored in a special register, so that if the condition is true then the branch target address is written into the Program Counter. If the condition is false then the Program Counter will not be modified and execution will continue as if no branching took place. There is also a no operation (NOP) instruction. The complete instruction set is given in Table I.

TABLE I. INSTRUCTION SET

	Instruction	Opcode				Operation
1	ADD	0	0	0	0	Rd = Rs1 + Rs2
2	SUB	0	0	0	1	If RS1 > RS2 Then Rd = RS1 – RS2 Else Rd = RS2 – RS1
3	AND	0	0	1	0	Rd = Rs1 & Rs2
4	OR	0	0	1	1	Rd = Rs1 | Rs2
5	NOT	0	1	0	0	Rd = ~ Rs
6	XOR	0	1	0	1	Rd = Rs1 ^ Rs2
7	CMP (Equal	0	1	1	0	If Rs1 = Rs2 Then Equal = 1, else Equal =0 If R1 = 0 Then AZ = 1, else AZ=0 If Rs2 =0 Then BZ =1, else BZ =0 If Rs1> Rs2 Then AGB = 1, else AGB = 0 If Rs1< Rs2 Then ALB = 1, else ALB = 0
8	SHIFT LEFT	0	1	1	1	Rd=Rs1 <<1
9	SHIFT RIGHT	1	0	0	0	Rd=Rs1 >>1
10	LOAD	1	0	0	1	Rd = Mem[Rs1]
11	STORE	1	0	1	0	Mem[Rs1] = Rs2
12	JUMP	1	0	1	1	PC = PC+Offset
13	NOP	1	1	0	0	No operation
14	JZ	1	1	0	1	PC = PC+Offset if Rd == 0
15	JNZ	1	1	1	0	PC = PC+Offset if Rd != 1
16	LOAD 8-BIT IMMEDIATE	1	1	1	1	Rd = 8-bit Immediate

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 04, April-2017

III. INSTRUCTION FORMAT

After deciding the instructions, the next step was to decide the instruction format. The uniform and orderly placement of the fields in the instruction is called instruction format. The instruction words are of 16-bits each. For uniformity, the operation code (opcode) was kept as 4-bits for all the instructions. So there are 16 instructions. The register file has 16 registers of 16-bits each. In order to address 16 registers, we need a 4-bit register address field in an instruction.

A. Arithmetic and Logical Instructions

The arithmetic and logical instructions have a common format : the opcode, the destination register address (Rd), the first source register address (Rs1) and the second source address (Rs2). This format applies to two operand instructions like AND, SUB etc. For single operand instructions like NOT only a single source register address is required. Hence the 4-bits for source register 1 are unused and we use the address in the field for the second register.

Opcode (4-bits)	Destination Register (Rd) Address (4-bits)	Source Register 1 Address (4-bits) (Rs1)	Source Register 2 Address (4-bits) (Rs2)

Fig. 1. Format of an Arithmetic or Logical Instruction

B. Compare Instruction

The compare instruction compares the two source addresses (Rs1 and Rs2) and outputs a word that is stored in the destination register (Rd). Each bit represents some characteristic of the data. There are five conditions and each of the first five bits of the output word correspond to a single condition. If any bit is set it means that, that condition is true. The interpretation of each bit is given in Table II. The instruction also sets the five flag bits of the processor. These flag bits are used for the conditional jump instructions to decide if that condition is true.

Opcode (4-bits)	Destination Register Address (4-bits) (Rd)	Source Register 1 Address (4-bits) (Rs1)	Source Register 2 Address (4-bits) (Rs2)

Fig. 2. Format of Compare Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	X	ALB	AGB	BZ	AZ	EQUAL

Fig. 3. Format of Compare Instruction Output word

TABLE II. INTERPRETATION OF COMPARE OUTPUT WORD

Bit	Abbreviation	Meaning	
0	EQUAL	0	Rs1 and Rs2 are equal
		1	Rs1 and Rs2 are unequal
1	AZ	0	Rs1 is NON-ZERO
		1	Rs1 is ZERO
2	BZ	0	Rs2 is NON-ZERO
		1	Rs2 is ZERO
3	AGB	0	Rs1 is not greater than Rs2
		1	Rs1 is greater than Rs2
4	ALB	0	Rs2 is not greater than Rs1
		1	Rs2 is greater than Rs1

C. Memory Access Instrucions

There are two memory access instructions, a memory read and a memory write. The memory read instruction (or LOAD instruction) loads a 16-bit word into the register address specified in Rd. The memory address is stored in the first source register (Rs1). The memory write (or STORE) instruction takes the contents in the specified source register (Rs2) and writes it to a memory location stored in the first source register (Rs1).

Opcode (4-bits)	Destination Register Address (4-bits) (Rd)	Source Register 1 Address (4-bits) (Rs1)	Unused (4-bits)

Fig. 4. Format of Memory Read (LOAD) instruction

Opcode (4-bits)	Unused (4-bits)	Source Register 1 Address (4-bits) (Rs1)	Source Register 2 Address (4-bits) (Rs2)

Fig. 5. Format of Memory Write (STORE) instruction

D. Unconditional Jump

The unconditional jump will change the branch of execution of the program and will load the previously calculated target address into the Program Counter. The 8-bit offset is provided in the first 8-bits of the instruction. The next four bits are unused.

Opcode(4-bits)	Unused (4-bits)	8-bit offset (Immediate)

Fig. 6. Format Unconditional (JMP) instruction

E. Conditional Jumps – JZ and JNZ

As stated before, the compare instruction also sets the flags of the processor. These flags are used to decide whether the condition to be checked for each of these jumps is true or not.

Opcode (4-bits)	Address of Register to be checked (4-bits)	8-bit offset (Immediate)

Fig. 7. Format Conditional (JZ & JNZ) instruction

- JZ – Jump if Zero
  This instruction checks whether the register specified the 4-bits after the opcode has a zero value or not. If it is, then the condition is true and the branch target address will be written into the program counter. If false the program counter remains unaffected.

- JNZ – Jump if Non-Zero
  JNZ has exactly the opposite condition as JZ, it checks if the register specified is not zero. But the instruction format is the same.

F. Load 8-Bit Immediate

This instruction is useful to load 8-bit immediate data into the registers. For now, the processor treats the 8-bit immediate as an unsigned integer and only appends 8 zeroes (i.e. 0x00) ahead of the number to make it 16-bit. This is only for the preliminary stage. A sign-extension unit would be much more efficient for this purpose. The format is given in Figure 8.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 04, April-2017

Opcode (4-bits)	Destination Register Address (4-bits) (Rd)	8-bit offset (Immediate)

Fig. 8. Format of LI instruction

IV. ARCHITECTURE

After the instructions and their formats were finalized, the architecture could finally be developed. The architecture was developed in stages. The processor was initially simple in design in stage 1, so it was called simple processing unit or SPU for short. But in the further stages, more of the instructions, were implemented and in stage 3 all the instructions were implemented. The stages are described below.

A. *SPU Version 1*

It is a simple processor. It is simply a test module which cannot be used as a part of any system. There was not much functionality built into this processor. Only 3 blocks were implemented:

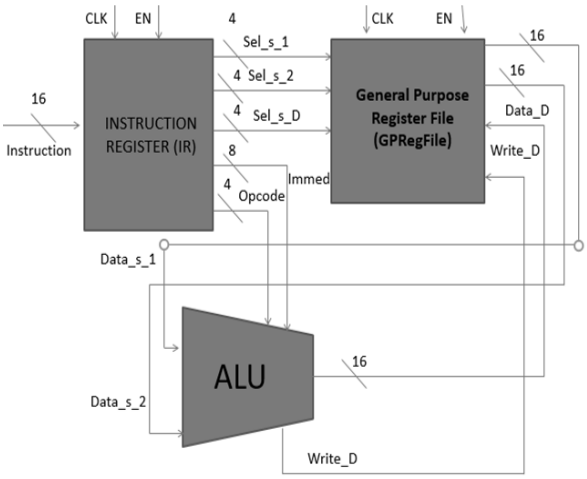

Fig. 9. SPU Version 1 architecture

- Instruction Register (IR)
  The ISA is uniform as bit ranges of instructions are shared. The decoder takes the bit fields, separates them accordingly and feeds them to the other units it is connected to.

- ALU
  Works like a standard ALU in this version

- General Purpose Register File (GPRF)
  On each clock cycle, it updates the source 1 and 2 register outputs when given the selection inputs for them and write to the destination register if enabled.

There are certain drawbacks of this version of the processor. Instructions have to be fed directly to the IR, latency periods need to be inserted between instructions and it cannot interface to memory or perform branching.

B. *SPU Version 2*

This version is self-sustaining and can fetch instructions on its own. The SRAM module acts as memory. No latency periods are needed. Inspiration was taken from the multi-cycle MIPS datapath for this implementation. New components were added to facilitate this such as the Control Unit (CU), Program Counter (PC) and some multiplexers. The processor uses several components in a single cycle and the processor uses a 5-state finite state machine to control these components. In each state the CU generates the control signals for each module in the different states. The five states are: START, FETCH, DECODE, EXECUTE & WRITEBACK.

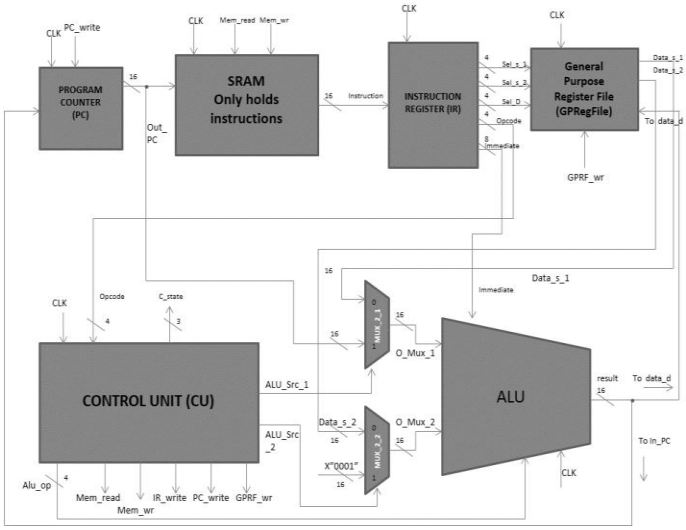

Fig. 10. SPU Version 2 architecture

There were no proper means to write instructions to memory as no user interface was created and there was no means for branching, but it successfully performed arithmetic and logical instructions at a good speed. A VHDL testbench was created for this processor. A small example program could be typed into the SRAM module and for simulation of the testbench.

The state diagram is shown below in Figure 11. It only describes the execution of arithmetic and logical instructions.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 04, April-2017

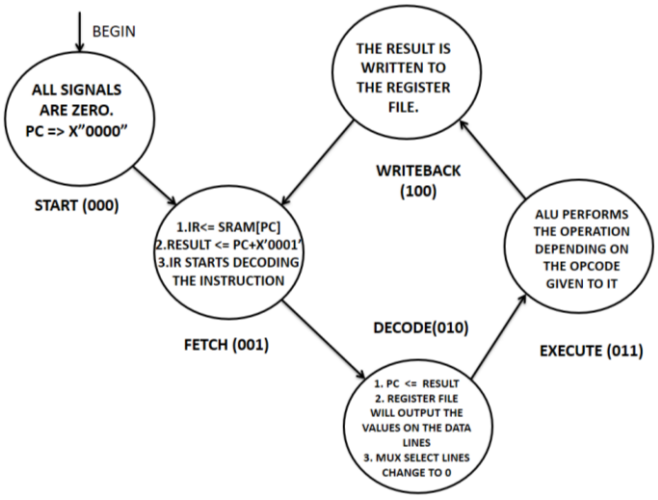

Fig. 11. SPU Version 2 state diagram

C. *SPU Version 3*

Version 3 implements all 16 instructions. The architecture was modified. The Control Unit was modified the most as it had to provide more control signals than before. Only registers and multiplexers were added as these are low on real-estate and are also cheaper to implement as compared to completely new modules. The extra control signals that were added were for the branching and memory write instructions. The control signals are given in the Table III. The Figure 12 shows the architecture of the processor.

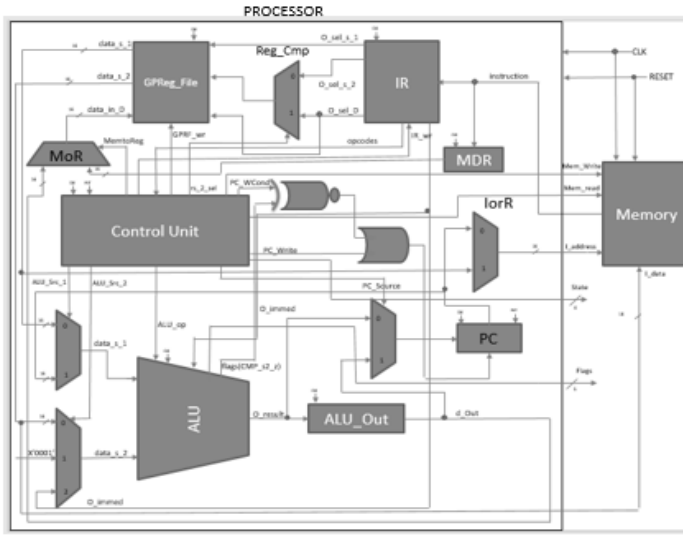

Fig. 12. SPU Version 3 architecture

The processor is enclosed in the black box. The memory is only an external unit. This version can interface with the memory using control signals Mem_write, Mem_read and the respective lines for sending and receiving address and data from memory. Some new states were also added to handle the memory read, write, conditional and unconditional branching instructions.

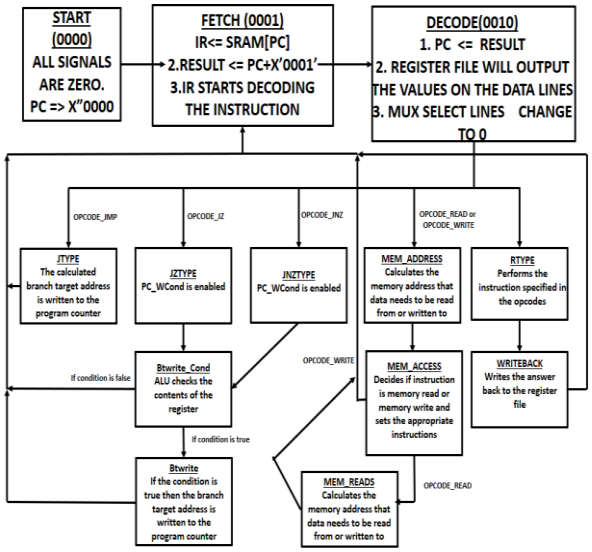

Fig. 13. SPU Version 3 state digram

TABLE III. CONTROL SIGNALS USED BY THE PROCESSOR

Control Signal	Value	Function
GPRF_wr	0	Disables write to register file
	1	Enables write to register file
IR_wr	0	Disables write to instruction decoder
	1	Enables write to instruction decoder
PC_Write	0	Disables write to Program Counter
	1	Enables write to Program Counter
PC_WCond	0	Disables write to Program Counter for conditional jumps
	1	Enables write to Program Counter for conditional jumps
PC_Source	0	Passes the calculated next address to the Program Counter
	1	Passes the branch target address to the Program Counter for an unconditional jump or if the condition is true for a conditional jump
ALU_op	0000-1111	Takes value passed from Control Unit to decide what action takes place
ALU_Src_1	0	Passes data in source register 1 to ALU
	1	Passes PC_addr to ALU during FETCH cycle
ALU_Src_2	00	Passes data in source register 2 to ALU
	01	Passes X'0001' to ALU
	10	Passes 8-bit Immediate to ALU
Rs2_select	0	Passes the address of source register 2 when the instruction is not a conditional jump
	1	Passes the address of the register to be checked for a conditional jump
MemtoReg	0	Passes the data calculated by ALU to register file
	1	Passes the data stored in a memory location during a memory read instruction
IorD	0	Passes the Program Counter contents as the next address
	1	Passes the memory address stored in source register 1 for a memory write instruction
Mem_Write	0	Disables write to Memory
	1	Enables write to Memory
Mem_Read	0	Disables read from Memory
	1	Enables read from Memory
Cu_state	0000-1111	Tells us what state the processor is in

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 04, April-2017

V. SIMULATION RESULTS

A. Top-Level Module

A top-level module that contains the instances of the processor, an external memory and an address decoder was created.

The module SPU_3_top is the processor block. All the components of the processor are instantiated in this top-level block. The RTL schematic is shown in Figure 14.

As very large memories take a long time to synthesize and utilize too much of the FPGAs resources a small memory with 16 addresses was chosen for simulation purposes, so an address decoder was used that could convert a 16-bit address to a 4-bit address. The memory module contains the test program to be executed. The program is typed into the memory before simulation.

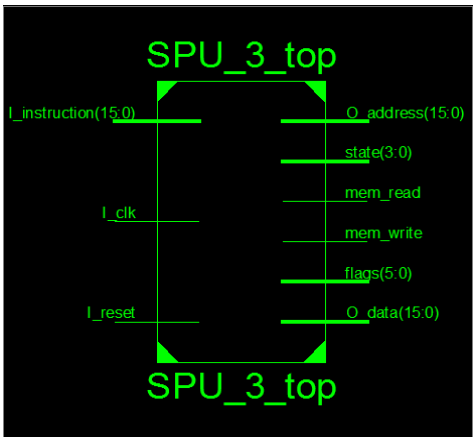

Fig. 14. RTL Schematic of SPU_3_top processor

The RTL schematic is expanded below and it shows us the different blocks in the processor.

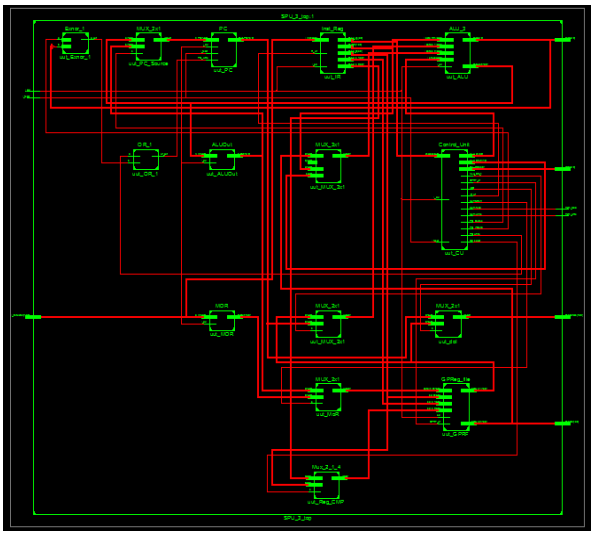

Fig. 15. Expanded RTL Schematic of SPU_3_top

B. Device Utilization

TABLE IV. RESOURCES UTILIZED BY PROCESSOR

Slice Logic Utilization	Utilization	
Number of Slice Registers	150 of 11,400	1%
Number of Slice LUTs	201 of 5,720	3%
Number of IOBs	62 of 200	1%
Number of occupied Slices	69 of 1,430	4%

C. Simulation in ISim

The example program given in Table V contains a single instruction of each type. A clock of 2us was used.

- LOAD 8-BIT IMMEDIATE
  The LI instruction takes 4 cycles to execute. The instruction here is xF001. It loads Immediate x0001 into register R0. Waveform is shown in Figure 16. Here it takes 8us to execute.

- ADD
  The ADD instruction takes 4 cycles to execute. The instruction here is x0301. It adds the value in registers R0 and R1 and places the answer in R3. Waveform is shown in Figure 17. All 9 arithmetic and logical instructions in the instruction set take the same amount of time for execution.

TABLE V. EXAMPLE PROGRAM IN MEMORY

Address	Instruction	Machine Code
$(0000)_{16}$	LI R0 x01	$(F001)_{16}$
$(0001)_{16}$	LI R1 x02	$(F102)_{16}$
$(0002)_{16}$	LI R2 x0A	$(F20A)_{16}$
$(0003)_{16}$	ADD R3, R0, R1	$(0301)_{16}$
$(0004)_{16}$	LI R5 x0D	$(F50D)_{16}$
$(0005)_{16}$	JZ R4 x03	$(D403)_{16}$
$(0006)_{16}$	0000	$(0000)_{16}$
$(0007)_{16}$	0000	$(0000)_{16}$
$(0008)_{16}$	ADD R4, R0, R2	$(0402)_{16}$
$(0009)_{16}$	STORE R5, R4	$(A054)_{26}$
$(000A)_{16}$	JMP x005	$(B005)_{16}$
$(000B)_{16}$	0000	$(0000)_{16}$
$(000C)_{16}$	0000	$(0000)_{16}$
$(000D)_{16}$	0000	$(0000)_{16}$
$(000E)_{16}$	0000	$(0000)_{16}$
$(000F)_{16}$	LOAD R8, R2	$(9820)_{16}$

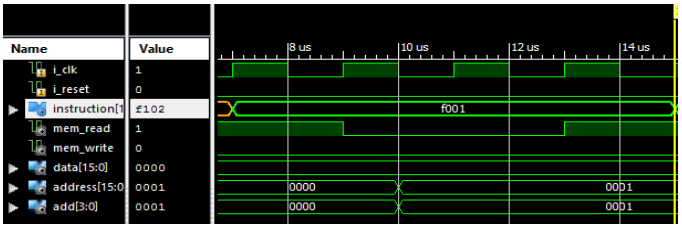

Fig. 16. Waveform of LI instruction

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 04, April-2017

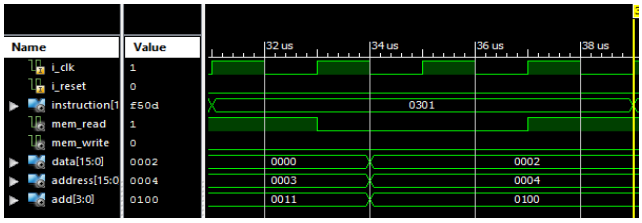

Fig. 17. Waveform of ADD instruction

- JZ – Conditional Jump

The JZ instruction takes 5 cycles to execute, irrespective if condition is true or false. The instruction here is xD403. It checks the value in register 4 and if the condition is true it updates the program counter with the new address which offsets the current address by 3. As the current address is x0005 and the condition is true, the next address will be x0008. The JNZ instruction executes in the same manner. Waveform is shown in Figure 18.

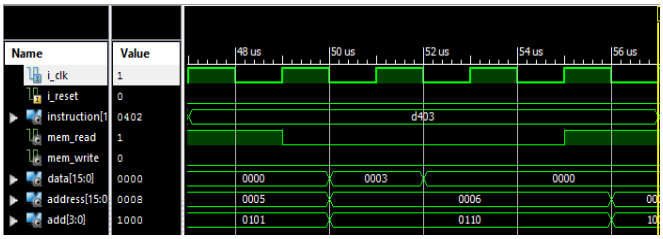

Fig. 18. Waveform of JZ instruction

- STORE – Memory Write

The STORE instruction writes the data in a register to a given memory location. It takes 5 cycles to execute. Here the instruction is xA054. It writes the data stored in register 4 to the memory address stored in register 5. The processor goes through the MEM_ADDRESS and MEM_ACCESS states before finally performing the write. Waveform is shown in Figure 19.

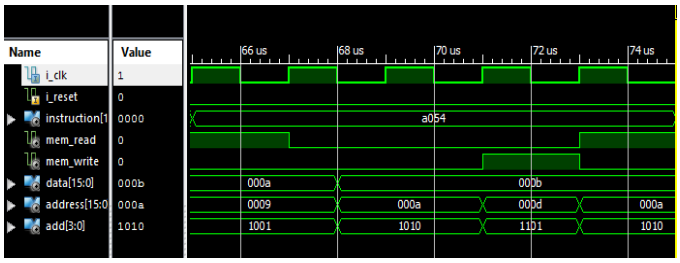

Fig. 19. Waveform of STORE instruction

- JMP – Unconditional Jump

The instruction will cause a jump to the next address formed by adding the current address in the program counter to the 8-bit offfset in the instruction. It takes 3 cycles to execute. Here the instruction is xB005. The current address is x000A after adding the offset we get the new address as x000F. The execution will now jump to that memory location. The instruction at that address is x9820. Waveform is shown in Figure 20.

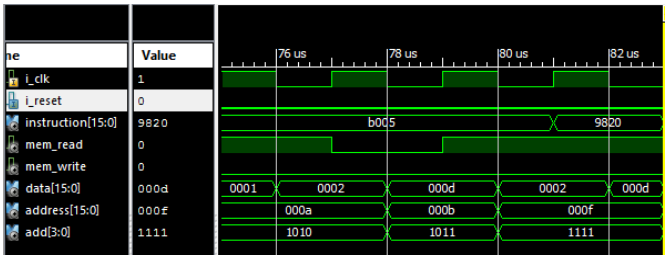

Fig. 20. Waveform of JMP instruction

- LOAD – Memory Read

The LOAD instruction will write the data stored in the memory address which is stored in the source register 1 to the destination register. It takes 4 cycles to execute. Here the instruction is x9820. It loads register 8 wth the data at address stored in register 2. The waveform is shown in Figure 21.

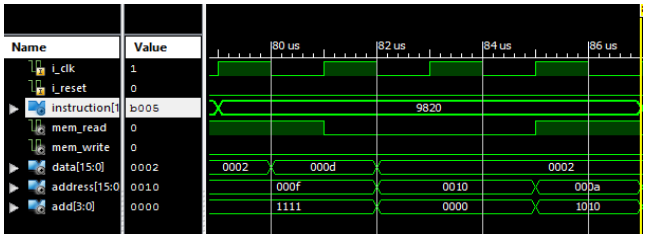

Fig. 21. Waveform of JMP instruction

VI. CONCLUSION

A 16-bit RISC processor has been realized. It can execute an instruction set with 16 instructions of different classes like arithmetic and logical, jumping - both conditional and unconditional jumps and memory interface instructions. The simulation output is compared with the expected results and the functionality is found correct.

The design can be improved in several ways to make it more sophisticated. A user interface can be developed so that a user can enter programs and write them to memory which will then be fetched for execution.

ACKNOWLEDGMENTS

The authors would like to thank their project guide, Prof. Madhura Shirodkar. Her insight and vision have made it possible for us to pursue and understand developments in the areas of Processor Design, VHDL programming and FPGAs.

The authors would also like to thank our Principal, Director, teachers and staff of the Xavier Institute of Engineering, Mumbai, India for providing all the facilities to conduct this project.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 04, April-2017

REFERENCES

[1] Vishwas V. Balpande, Abhishek B. Pande, Meeta J. Walke, Bhavna D. Choudhari, Kiran R. Bagade, "Design and Implementation of 16 Bit Processor on FPGA", IJARCSSE, January 2015.

[2] Nupur Gupta, Pragati Gupta, Himanshi Bajpai, Richa Singh, Shilpa Saxena "Analysis of 16 Bit Microprocessor Architecture on FPGA Using VHDL", IJAREEIE, April 2014.

[3] V. R. Gaikwad, "Design, Implementation and Testing of 16 bit RISC Processor", IOSR-JVSP, March 2013.

[4] Amanjyot Singh Johar, "16 bit Reduced Instruction Set Computer (RISC) Processor Design A Project Report" , Department of Electrical and Computer Engineering, University of Illinois at Chicago, September 2013.

[5] M.Kishore Kumar, MD.Shabeena Begum, "FPGA Based Implementation of a 32-bit RISC processor", IJERA, Setember 2011.

[6] David A. Patterson, David R. Ditzel, "The Case for The Reduced Instruction Set Computer".

[7] J.B. Nade, Dr. R. V. Sarwadnya, "The Soft Core Processors : A Review", IJIREEICE, December 2015.

[8] Anders Wallander, "A VHDL Implementation of a MIPS" , Department of Computer Science and Engineering, Lulea Teniska Universitet, January 2000

[9] Anjana R, Krunal Gandhi, "VHDL Implementation of a MIPS RISC Processor", IJARCSSE, August 2012.

[10] David A. Patterson, John L. Hennessey, Computer Organisation And Design, Third edition, 2007.