Advances in Engineering Design Technology

Journal homepage: www.nipesjournals.org.ng

Enhancement of a Synthesized Single Cycle MIPS Processor on Altera DE2 FPGA Development Board

S.U. Mustapha[a], S.S. Ahmad[b]

[a]Bayero University, Kano, No. 2 Hassan Gwarzo Road, Kano, Nigeria.

[b]King Fahad University of Petroleum and Minerals, Dammam, Saudi Arabia

Corresponding Author: sumustapha.phy@buk.edu.ng

ARTICLE INFORMATION

Article history:
Received 04 May 2021
Revised 02 July 2021
Accepted 09 August 2021
Available online 31 August 2021

Keywords:
Altera, DE-2, MIPS, Synthesis, Verilog

https://doi.org/10.37933/nipes.a/3.2.2021.6

https://nipesjournals.org.ng

ABSTRACT

The MIPS processor continues to be one of the most popular processors in the field of embedded systems mainly because of its speed and reduced instructions. This work presents a single cycle MIPS synthesized on Altera DE2 development board. It is further enhanced by developing extra sets of instructions. The design is tested by running Mips Assembly Language (MAL) programs on the model. Results indicate that the enhanced model has more executable instructions and has an on board maximum frequency of 25MHz which is good for softcore processor.

1. Introduction

The Microprocessor without Interlocked Pipeline Stages (MIPS) is highly popular in field of a embedded systems design. It is a Reduced Instruction Set Computer (RISC) processor that supports fewer and simpler instructions than their counter parts (Complex Instruction Set Computer (CISC) processors). The idea behind the reduction of instruction is to reduce hardware size and increase the speed of operation as shown by research [1, 2]. The MIPS processor has a load-store architecture and has a 5-Stage instruction execution stage [3]. This means that it requires two instructions to be executed in order to access memory. Although it is pipelined, it solves the problem of hardware interlocking by implementing solutions in software leaving a user with a fast system that has a simple instruction set [4].

MIPS processors are designed to be deployed as VLSI softcore processors [5]. A primary target hardware for deploying it is a Field Programmable Gate Array (FPGA) such as Altera DE-2. MIPS being softcore, needs to be synthesized onto a chosen hardware. This is achieved by translating the whole architecture using a hardware description language. The beauty of this is that both 32bit and 64bit MIPS can be synthesized on an FPGA. Although this has been attempted and done by some researchers, there is a still a gap in the number of instructions that can be performed by the synthesized processor [6]. A key thing to keep in mind is that when a processor is successfully synthesized onto a hardware, the native programming language of the synthesized processor can be on the hardware. For example, if an arduino is synthesized on an FPGA, one can use C++ programs written to be used by an arduino. The input and output pins mapping and architectural design will be carried using the destination language (Verliog). The only difference is that the hardware will be an FPGA. In a nutshell, it is just simply implementing a software based arduino on an FPGA.

In this work, a detailed implementation ANDI (AND Immediate), NOR (NOR), and LHW (Load Half Word) will be presented and discussed. The new set of instructions were written in verilog HDL and simulated using Signal-Tap analysis tool in Quartus-II software. The improved system was tested by verifying both hardware and software functionality. The design details and results will be discussed next sections.

2. Methodology

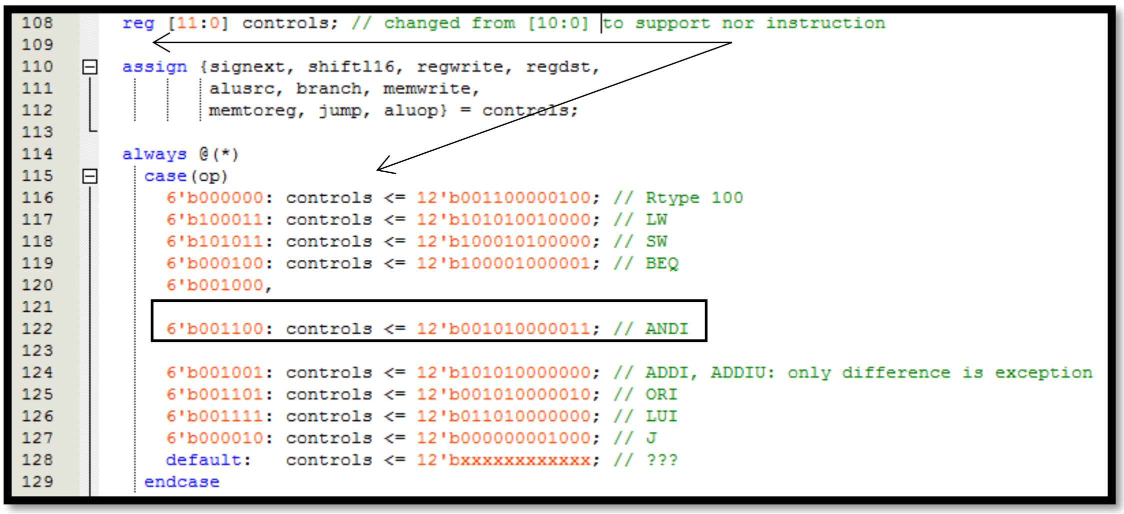
The MIPS design presented only implements a limited number of MIPS instructions. The instructions NOR, LH and ANDI were added to the existing design file by making some modifications in the supplied design file. The changes done for implementing each instruction will be presented. The MIPS processors has three prominent instruction types [7], the R-type which passes values within registers, the I-type which takes a register and an immediate value and a J-type instruction which are used when a jump is to be performed.

2.1 ANDI (AND Immediate) Instruction

The instruction ANDI is an I-type instruction that uses a source register (Rs), target register (Rt) and immediate value (Imm). It has the structure.

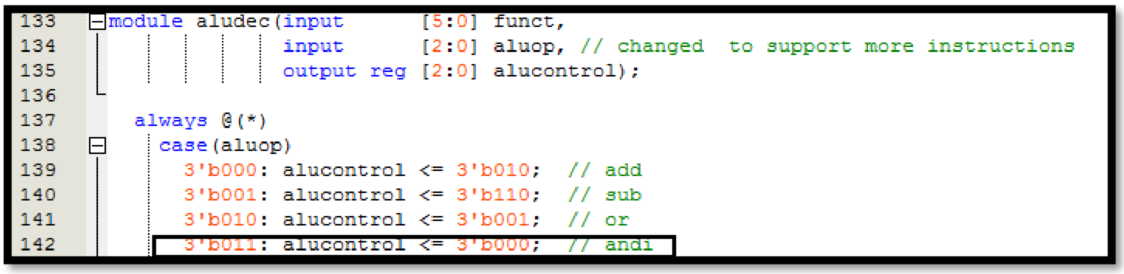
andi \$Rt, \$Rs, Imm #Rt:= Rs & Imm (1)

It performs a logical and with the contents of the source register and the immediate value and place the result in the target register. The additions were made to implement the **andi** instruction are shown in Figures 1 and 2.



```
reg [11:0] controls; // changed from [10:0] to support nor instruction

assign {signext, shiftl16, regwrite, regdst,
        alusrc, branch, memwrite,
        memtoreg, jump, aluop} = controls;

always @(*)
  case(op)
    6'b000000: controls <= 12'b001100000100; // Rtype 100
    6'b100011: controls <= 12'b101010010000; // LW
    6'b101011: controls <= 12'b100010100000; // SW
    6'b000100: controls <= 12'b100001000001; // BEQ
    6'b001000,

    6'b001100: controls <= 12'b001010000011; // ANDI

    6'b001001: controls <= 12'b101010000000; // ADDI, ADDIU: only difference is exception
    6'b001101: controls <= 12'b001010000010; // ORI
    6'b001111: controls <= 12'b011010000000; // LUI
    6'b000010: controls <= 12'b000000001000; // J
    default:   controls <= 12'bxxxxxxxxxxxx; // ???
  endcase
```

Figure 1: Decoding the opcode of the andi instruction



```
module aludec(input      [5:0] funct,
              input      [2:0] aluop, // changed  to support more instructions
              output reg [2:0] alucontrol);

  always @(*)
   case(aluop)
     3'b000: alucontrol <= 3'b010;  // add
     3'b001: alucontrol <= 3'b110;  // sub
     3'b010: alucontrol <= 3'b001;  // or
     3'b011: alucontrol <= 3'b000;  // andi
```

Figure 2: Adding the andi instruction to the ALU decoder

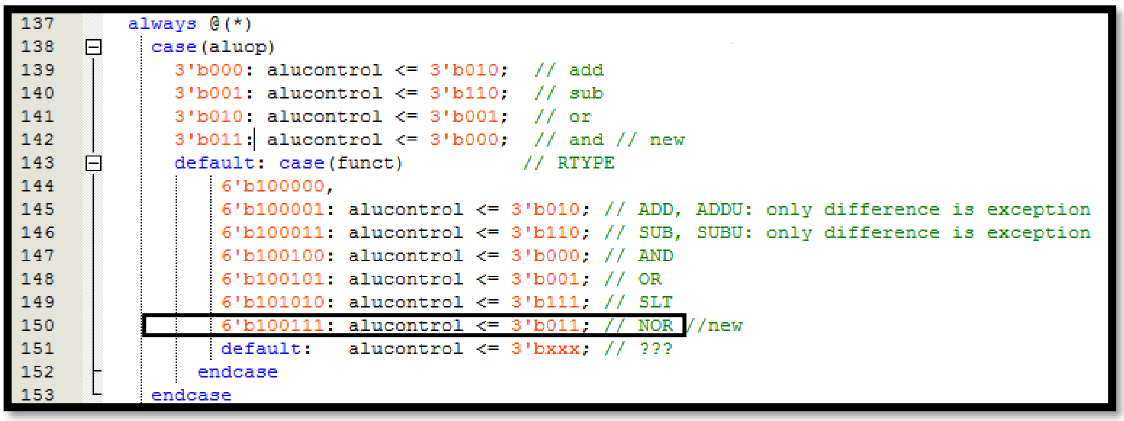
For the andi instruction, the result should be 0 extended, written to a register and sent to the ALU for computation. This is why the control signals are written as 12’b001010000011. The last three LSB of the control signal denote the ALU operation to be performed.

2.2 NOR Instruction

The instruction nor is an R-type instruction that uses a source register (Rs), target register (Rt) and a destination register (Rd), shift amount and function code (100111). All R-type instructions have and opcode of 000000. The nor instruction has the structure

nor $Rd, $Rs, $Rt #Rd:= ~ (Rs | Rt) (2)

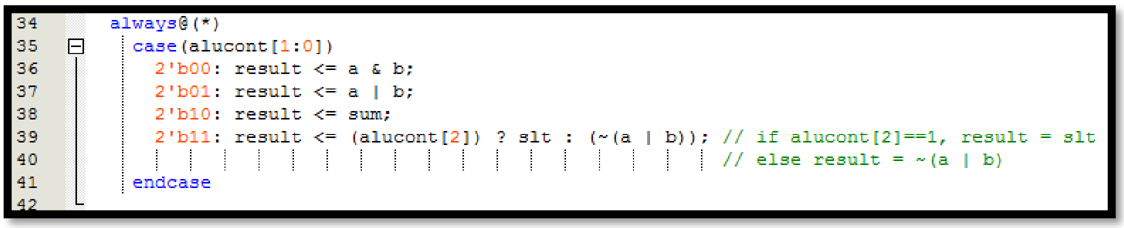
It performs a nor with the contents of the source register and the target register and place the result in the destination register. The changes made to implement the **nor** instruction are shown in Figures 3 and 4



```
  always @(*)
   case(aluop)
     3'b000: alucontrol <= 3'b010;  // add
     3'b001: alucontrol <= 3'b110;  // sub
     3'b010: alucontrol <= 3'b001;  // or
     3'b011: alucontrol <= 3'b000;  // and // new
     default: case(funct)          // RTYPE
         6'b100000,
         6'b100001: alucontrol <= 3'b010; // ADD, ADDU: only difference is exception
         6'b100011: alucontrol <= 3'b110; // SUB, SUBU: only difference is exception
         6'b100100: alucontrol <= 3'b000; // AND
         6'b100101: alucontrol <= 3'b001; // OR
         6'b101010: alucontrol <= 3'b111; // SLT
         6'b100111: alucontrol <= 3'b011; // NOR //new
         default:   alucontrol <= 3'bxxx; // ???
       endcase
   endcase
```

Figure 3: Decoding the opcode of the andi instruction

The function code of the nor instruction was added to the given instructions function codes as indicated in Figure 3. It utilizes the unused combination in the given ALU decoder (011).



```
  always@(*)
   case(alucont[1:0])
     2'b00: result <= a & b;
     2'b01: result <= a | b;
     2'b10: result <= sum;
     2'b11: result <= (alucont[2]) ? slt : (~(a | b)); // if alucont[2]==1, result = slt
                                                      // else result = ~(a | b)
   endcase
```

Figure 4: Utilizing the decoder to create a nor instruction.

For all R-type instructions, the result is 0 extended, written to the destination register and the operation to be performed depends on the function code. Thus, all control signals will have the 12'b001100000100.

2.3 LH (Load Half word) Instruction

The LH is an I-type instruction that uses a source register (Rs), target register (Rt) and an immediate offset. It loads half of word (16bits) from any memory address specified and sign extends it. It has the opcode of 100001. The LH instruction has the structure.

lh $Rt, Imm($Rs) # Rt:= [Mem$Rs + Imm] (3)

2.3.1 Adding the LH instruction

A new module was created to extract half word from a word and sign extend it based on the trigger of a newly added control signal. It was also designed to support loading either the upper half or the lower half word based on the instruction. The ASM chart of the module is shown in Figure 5.

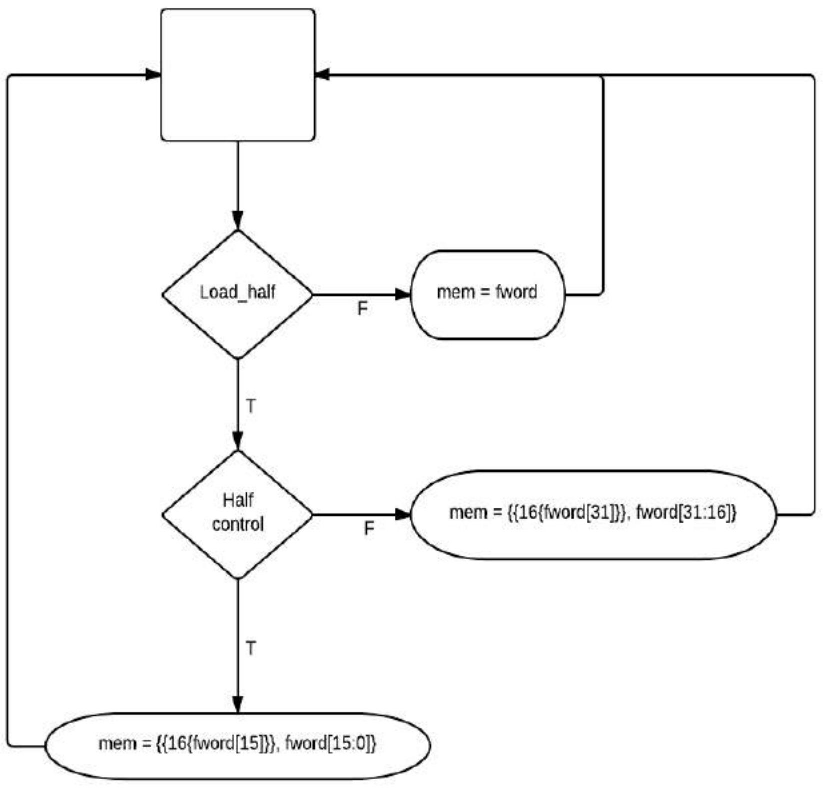
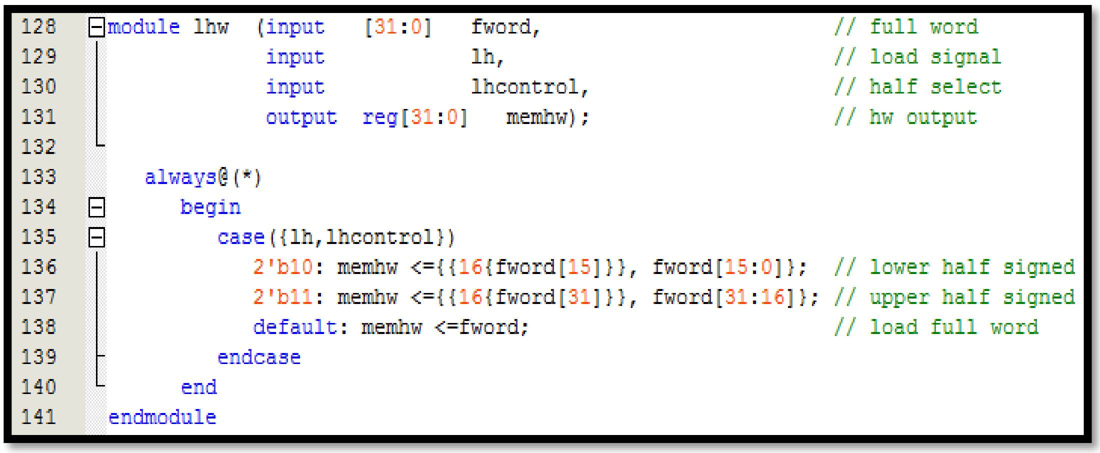

Load_half
F
mem = fword
T
Half control
F
mem = {{16{fword[31]}}, fword[31:16]}
T
mem = {{16{fword[15]}}, fword[15:0]}

Figure 5: ASM chart of the Load Half module

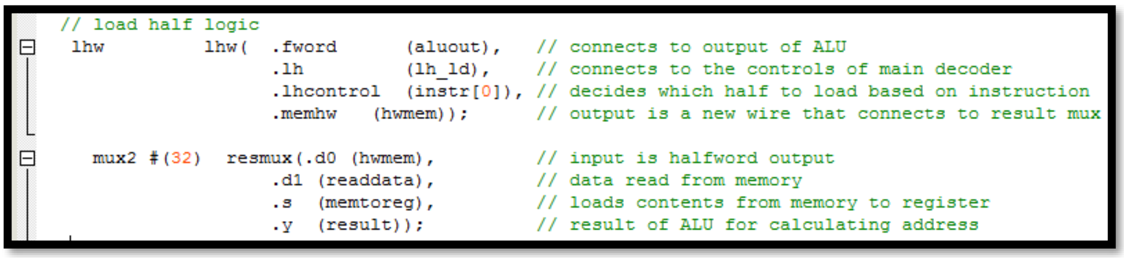
2.3.2 Verilog code of the LH module and other changes



```verilog
module lhw  (input   [31:0]    fword,                              // full word
             input             lh,                                 // load signal
             input             lhcontrol,                          // half select
             output  reg[31:0]   memhw);                           // hw output

   always@(*)
      begin
         case({lh,lhcontrol})
            2'b10: memhw <={{16{fword[15]}}, fword[15:0]};  // lower half signed
            2'b11: memhw <={{16{fword[31]}}, fword[31:16]}; // upper half signed
            default: memhw <=fword;                         // load full word
         endcase
      end
endmodule
```

Figure 6: Verilog code of the load half word module

The load half word module was then connected to the existing datapath and controller after it was instantiated. The module instantiation is shown in Figure 7 while the RTL view is shown in Figure 8.



```verilog
  // load half logic
  lhw        lhw(  .fword      (aluout),    // connects to output of ALU
                   .lh         (lh_ld),     // connects to the controls of main decoder
                   .lhcontrol  (instr[0]),  // decides which half to load based on instruction
                   .memhw    (hwmem));      // output is a new wire that connects to result mux

    mux2 #(32)  resmux(.d0  (hwmem),        // input is halfword output
                   .d1  (readdata),         // data read from memory
                   .s   (memtoreg),         // loads contents from memory to register
                   .y   (result));          // result of ALU for calculating address
```

Figure 7: load half word module instantiation in mips.v file

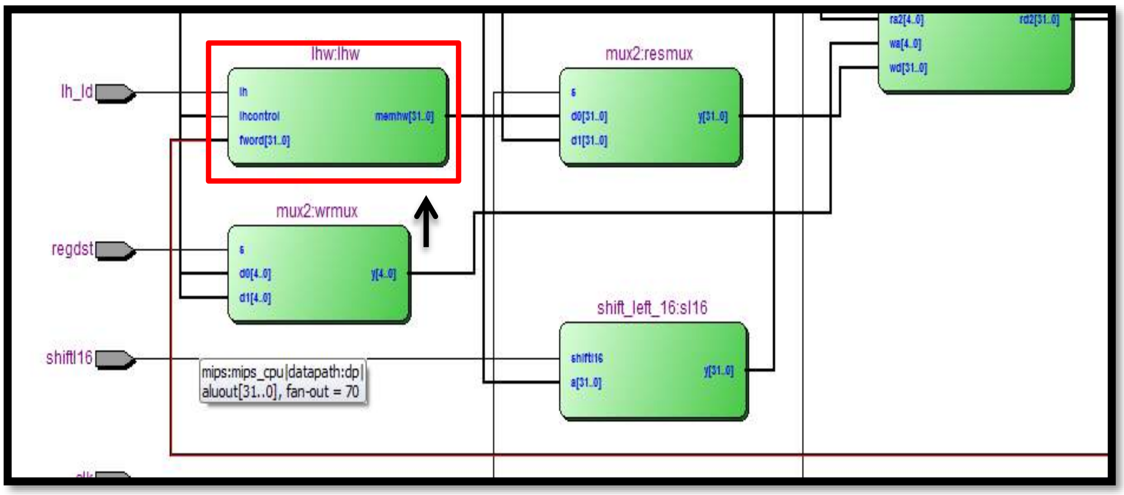
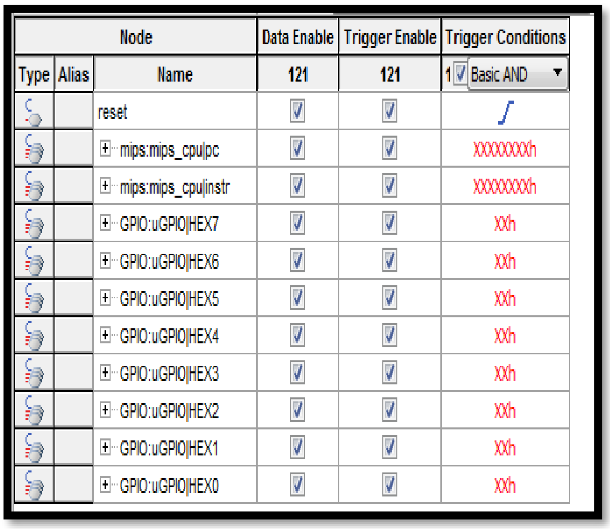


Figure 8: Partial part RTL view of the load half module (lhw)

3. Results and Discussion

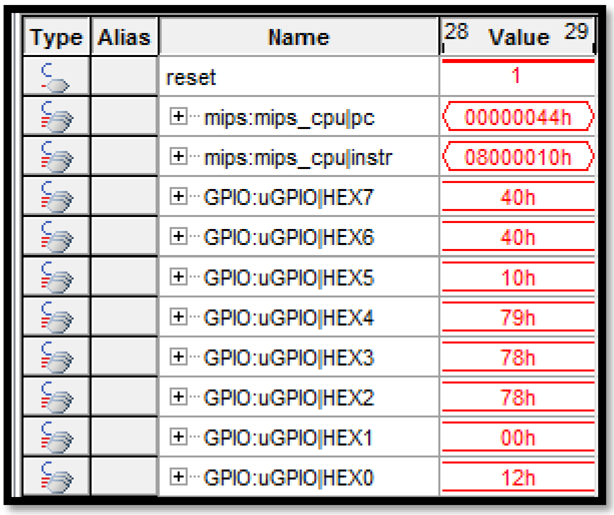
3.1 Signal Tap Result for Testing the functionality of the synthesized MIPS Processor

Signal tap was used to verify the result for correctness. It was also used to view the signals from the MIPS system. The signals that were used to analyse the written code are Program Counter (PC), instruction register (Instr) and Hex0 – Hex7. In addition, a reset signal was used to trigger the data acquisition. The setup is shown Figure 9 and the result is in Figure 8.



Type	Alias	Name	Data Enable	Trigger Enable	Trigger Conditions
			121	121	1 ✓ Basic AND
		reset	✓	✓	ʃ
		mips:mips_cpu|pc	✓	✓	XXXXXXXXh
		mips:mips_cpu|instr	✓	✓	XXXXXXXXh
		GPIO:uGPIO|HEX7	✓	✓	XXh
		GPIO:uGPIO|HEX6	✓	✓	XXh
		GPIO:uGPIO|HEX5	✓	✓	XXh
		GPIO:uGPIO|HEX4	✓	✓	XXh
		GPIO:uGPIO|HEX3	✓	✓	XXh
		GPIO:uGPIO|HEX2	✓	✓	XXh
		GPIO:uGPIO|HEX1	✓	✓	XXh
		GPIO:uGPIO|HEX0	✓	✓	XXh

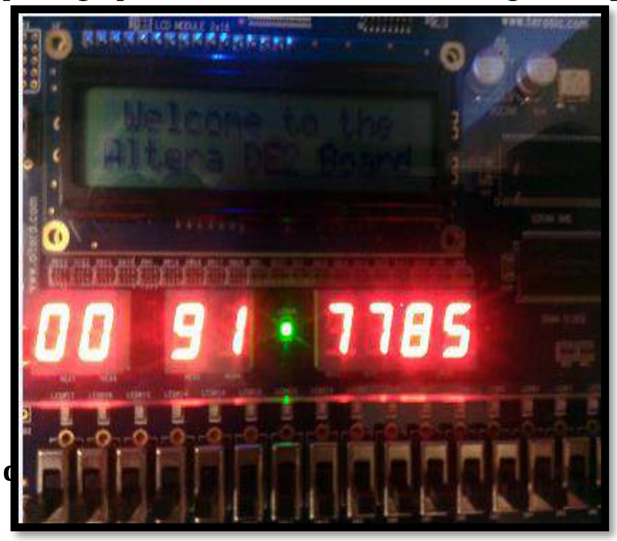
Figure 9: Signal tap analyser setup for verifying Hex values displayed

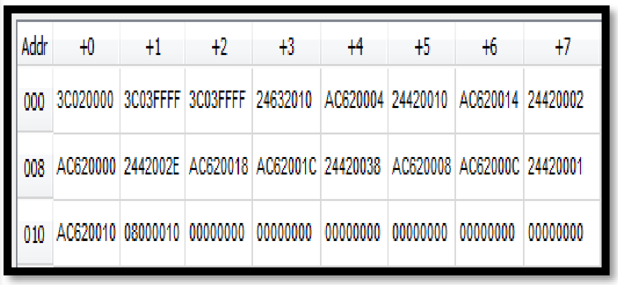


Type	Alias	Name	28 Value 29
		reset	1
		mips:mips_cpu|pc	00000044h
		mips:mips_cpu|instr	08000010h
		GPIO:uGPIO|HEX7	40h
		GPIO:uGPIO|HEX6	40h
		GPIO:uGPIO|HEX5	10h
		GPIO:uGPIO|HEX4	79h
		GPIO:uGPIO|HEX3	78h
		GPIO:uGPIO|HEX2	78h
		GPIO:uGPIO|HEX1	00h
		GPIO:uGPIO|HEX0	12h

Figure 10: Signal tap analyser result for displaying ID number

The result displayed in Figure 10 was taken after the MAL instruction was excuted. It shows the Hex values that display the ID number 00917785 on 8 seven segment displays. The number displayed is **00917785** and it corresponds to **(40h, 40h, 10h, 79h, 78h, 78h, 00h, 12h)**. The photograph of the board as shown in Figure 11 proves the functionality of the written program.





Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	3C020000	3C03FFFF	3C03FFFF	24632010	AC620004	24420010	AC620014	24420002
008	AC620000	2442002E	AC620018	AC62001C	24420038	AC620008	AC62000C	24420001
010	AC620010	08000010	00000000	00000000	00000000	00000000	00000000	00000000

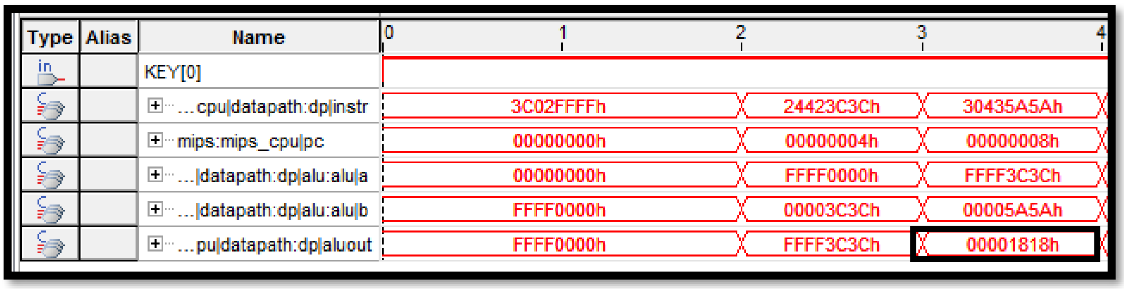
Figure 12: Memory Instantiation file setup for

c

Figure 11: Photograph of ID number displayed

The result obtained when the machine code was written into memory instantiation file and programmed to the DE2 board meets the requirement as shown in Figure 11. The memory instantiation file in Figure 12 consists of 18 machine codes that contain 18 instructions after decoded by the processor. Each instruction is word aligned and the program counter is incremented by 4 after each instruction is executed.

3.2 Signal Tap Test Result for andi instruction

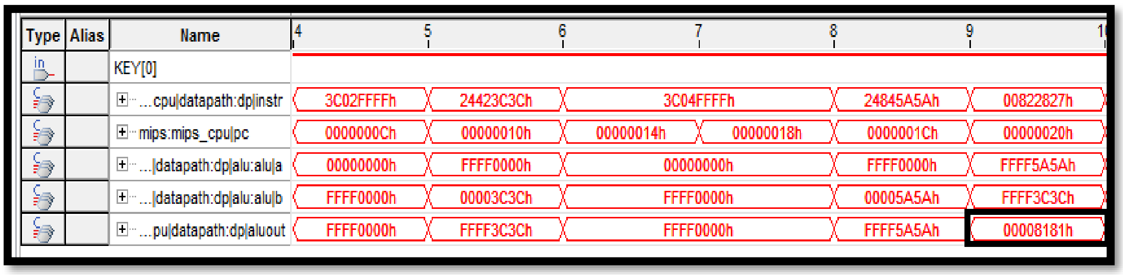


Type	Alias	Name	0–1	2	3
in		KEY[0]			
		...cpu|datapath:dp|instr	3C02FFFFh	24423C3Ch	30435A5Ah
		mips:mips_cpu|pc	00000000h	00000004h	00000008h
		...|datapath:dp|alu:alu|a	00000000h	FFFF0000h	FFFF3C3Ch
		...|datapath:dp|alu:alu|b	FFFF0000h	00003C3Ch	00005A5Ah
		...pu|datapath:dp|aluout	FFFF0000h	FFFF3C3Ch	00001818h

Figure 13: Test result for andi instruction

The signals used for testing the **andi** instruction are Program Counter (PC), instruction register (Instr), value a, value b, ALU output and reset. The values (a&b) used are 0xFFFF_3C3C and 0x0000_5A5A. The manual computation of the andi result is 0x00001818 and it tallies with the result obtained for the manual computation as in Figure 13.

3.3 Signal Tap Test Result for nor instruction



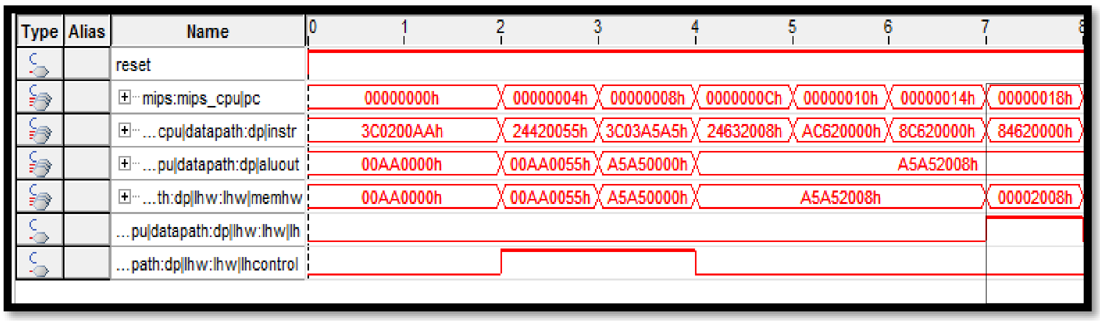
Type	Alias	Name	4	5	6	7	8	9
in		KEY[0]						
		...cpu|datapath:dp|instr	3C02FFFFh	24423C3Ch	3C04FFFFh	3C04FFFFh	24845A5Ah	00822827h
		mips:mips_cpu|pc	0000000Ch	00000010h	00000014h	00000018h	0000001Ch	00000020h
		...|datapath:dp|alu:alu|a	00000000h	FFFF0000h	00000000h	00000000h	FFFF0000h	FFFF5A5Ah
		...|datapath:dp|alu:alu|b	FFFF0000h	00003C3Ch	FFFF0000h	FFFF0000h	00005A5Ah	FFFF3C3Ch
		...pu|datapath:dp|aluout	FFFF0000h	FFFF3C3Ch	FFFF0000h	FFFF0000h	FFFF5A5Ah	00008181h

Figure 14: Test result for nor instruction

The signals used for testing the **or** instruction are Program Counter (PC), instruction register (Instr), value a, value b, ALU output and reset. The values (a&b) used are 0xFFFF_3C3C and 0x0000_5A5A. The manual computation of the NOR is 0x00008181 and it tallies with the result obtained for the manual computation as in Figure 14.

3.1.1 Signal Tap Logic Load Half Word Test Result

Signal tap logic analyser was used to test the load half word instruction. The design was tested to verify loading of a sign extended upper half word or lower half. The signals used to test are shown in Table 5. The functionality test results are shown in Figures 15-16.



Type	Alias	Name	0–1	2	3	4	5	6	7
		reset							
		mips:mips_cpu|pc	00000000h	00000004h	00000008h	0000000Ch	00000010h	00000014h	00000018h
		...cpu|datapath:dp|instr	3C0200AAh	24420055h	3C03A5A5h	24632008h	AC620000h	8C620000h	84620000h
		...pu|datapath:dp|aluout	00AA0000h	00AA0055h	A5A50000h	A5A52008h	A5A52008h	A5A52008h	
		...th:dp|lhw:lhw|memhw	00AA0000h	00AA0055h	A5A50000h	A5A52008h	A5A52008h	A5A52008h	00002008h
		...pu|datapath:dp|lhw:lhw|lh							
		...path:dp|lhw:lhw|lhcontrol							

Figure 15: Loading the lower half word

The full word as shown in Figure 15 is **A5A52008h**. The lower half word 2008 is loaded when the **load half** signal is given. The result is then sign extended with the MSB of the half word loaded.

Thus, the **memhw** indicates **00002008h**. The lower half word was loaded because the **lhcontrol** signal is low.

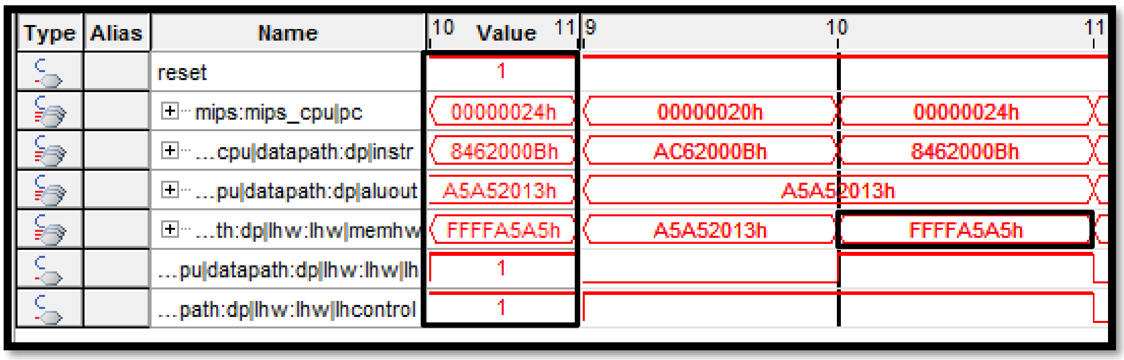


Figure 16: loading the upper half word

In figure 16, the full word is **A5A52013h**. The upper half word A5A5 gets loaded when the **load half** signal is given. The result is then sign extended with the MSB of the half word loaded. Thus, the **memhw** indicates **FFFFA5A5h**. The upper half word was loaded because the **lhcontrol** signal is high. It was noticed that the maximum attainable of operating frequency was 25MHz which is greater than the default Xilinx pico-blaze softcore processor that has an operating frequency of 16MHz.

4. Conclusion

In this paper, algorithmic state machines were translated to Verilog HDL to develop new instructions for a MIPS processor. ANDI, NOR and LHW instructions were added and tested using MIPS Assembly language. Results show that the synthesized processor works within a single cycle is capable of performing both R-type instructions where operation is done on two registers and I-type instructions were a register and immediate value is used. The results obtained also opened doors for developers and researchers interested in adding custom instructions that will suite their needs.

References

[1] E. d. Vries, "Introduction to the MIPS Processor," Electronic Design, Dublin, 2009.

[2] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, Cambridge: Morgan Kaufmann, 2017.

[3] M. Mano and C. R. Kime, Logic and Computer Design Fundamentals, New Jersey: Prentice Hall, 2008.

[4] G. Kane and J. Heinrich, MIPS RISC Architecture, New York: Prentice Hall, 1992.

[5] J. Hennessy, "MIPS: A Microprosessor Architecture," IEEE, Stanford, 1982.

[6] S. Suresh and R. Ganesh, "FPGA Implementation of MIPS RISC Processor," *International Journal of Engineering Research & Technology*, vol. 3, no. 1, pp. 1710-1714, 2014.

[7] MIPS Architecture for Programmers, Vols. I-A, Beijing: Imagination Technologies, 2016, pp. 1-10.

[8] P. Bhardwaj and S. Murugesan, "Design & Simulation Of A 32-Bit Risc Based Mips Processor Using Verilog," *IJRET: International Journal of Research in Engineering and Technology*, vol. 5, no. 11, pp. 166-172, 2017.

[9] Mohit N. Topiwala, N. Sarawathi, "I*mplementation of a 32-bit MIPS Based RISC Processor using Cadence*", International Conference on Advanced Communication Control and Computing Teclmologies (ICACCCT), 2014 IEEE.