EE 224
Course Project: IITB CPU

Satush Parikh 21D070062
Vansh Agarwal 210070091

November 30, 2022

Contents

1 Introduction

IITB CPU is an 8-register, 16-bit computer system i.e. it can process 16 bits at a time and uses point to point communication.
The output is 19 bits, MSB is out_flag which signifies when output is starting to be displayed. Carry flag, Zero flag then R0 to R7 are displayed sequentially.
Our project consists of the following files

- ALU.vhdl : Arithmetic logic unit capable of finding Sum,Difference,NAND on 16 bit numbers and finding equality between two vectors.
- DUT.vhdl & Testbench.vhdl which have been used are the same as used in the lab course.We have made tracefile on our own which mainly serves purpose of mimicing clock .
- RF.vhdl Register file is capable of reading two registers at a time,writing into one register at once.
- IITB_CPU.vhdl which includes mainly the code for the designed states and state transitions, with control signals clock, reset and control_o.

2 FSM Design

1. S0: This is the common first state of all the instructions and fetches the data pointed out by Program counter.
2. S1 : In this state we are reading the data into T1 and T2(signals) contained by two registers which are indicated in the instruction format
3. S2: We are adding T1 & T2 and modifying Carry Flag.
4. S3: Here, we are checking for Zero Flag modification.
5. S4: We are writing the data into the required register.
6. S5: This state makes sure that Register R7 always stores the Program Counter.
7. S6: Same as S2 but difference is we are doing NAND and not modifying Carry flag.

8. S7: Adding content of Reg A with Im(sign extended) and storing result in Reg B. This sign extends immediate bits appropriately by padding it with 0s or 1s.
9. S8: Writes the result into the register
10. S9: Similar to S5(but some other instructions' final state)
11. S10: Checks equality between content of two registers
12. S11: Subtracts 1 from the Program Counter
13. S12: Branches PC to PC+Imm(Sign extended)
14. S13: Makes sure that Register R7 always stores the Program Counter.
15. S14: Stores PC in reg A
16. S15: Same as S12 by sign extends by different length
17. S16: Updates PC
18. S17: Stores 9 bits immediates into signals MSB with other LSBs assigned to 0s.
19. S18: Memory address is computed by adding immediate 6 bits with content of reg B.
20. S19: Loads value from Memory
21. S20: Stores value from Reg A into Memory
22. J_i : Store required memory address in T2.
23. K_i : Input contents of T2 as memory address.
24. L_i : Write memory data into respective register and input values into ALU for updating variable multiple.
25. M_i : Updating multiple.
26. S22: Makes sure that Register R7 always stores the Program Counter.

27. N_i : Store required memory address in T2 and read respective signal in register file.

28. O_i : Write data into memory and input values into ALU for updating multiple.

29. P_i : Updating multiple.

30. S23: Makes sure that Register R7 always stores the Program Counter.

31. S24: Writes data into appropriate register

32. S25: Makes sure that Register R7 always stores the Program Counter.

33. X_i : Helps in linking content of registers to output vector after an instruction has been completed to see result.

3 State Diagrams

- ADD,ADC,ADZ

$$S0- > S1- > S2- > S3- > S4- > S5$$

- NDC,NDZ,NDU

$$S0- > S1- > S6- > S3- > S4- > S5$$

- ADI

$$S0- > S1- > S7- > S3- > S8$$

- JAL

$$S0- > S1- > S11- > S14- > S15- > S13$$

- BEQ

$$S0- > S1- > S10- > S11- > S12- > S13$$

- LHI

$$S0- > S17- > S24- > S25$$

- JLR

$$S0- > S1- > S11- > S14- > S16- > S13$$

- LW

$$S0- > S1- > S18- > S19- > S3- > S4- > S5$$

- SW

$$S0- > S1- > S18- > S20$$

- LM

$$S0- > S1- > Ji- > Ki- > Li- > Mi- > S22$$

- SM

$$S0- > S1- > Ni- > Oi- > Pi- > S23$$

4 Simulation Results

4.1 ADD

```
    Memory:

--  0 => "0000_001_010_011_000" ,--ADD Adds content of register 1 & 2 and stores i


In RF.vhdl

signal R1 : std_logic_vector(15 downto 0) := "0000000000000001";
signal R2 : std_logic_vector(15 downto 0) := "0000000000000010";
signal R3 : std_logic_vector(15 downto 0) := "0000000000000000";

Other signals can be taken as x \0000"
```

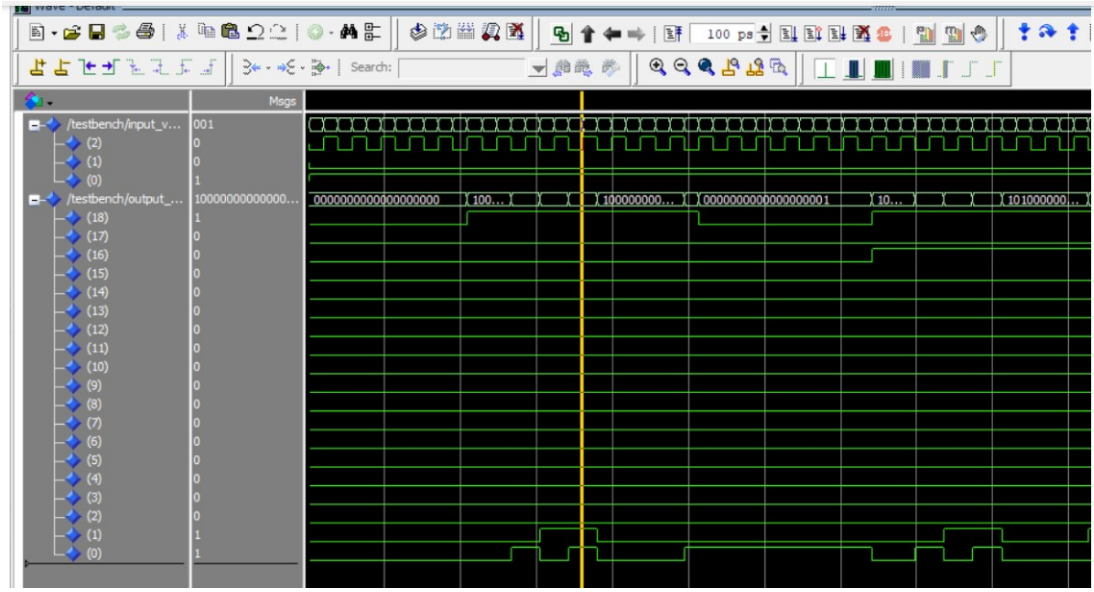


Figure 1: ADD

4.2 ADC

```
    Memory:

  0 => "0000_001_010_011_010" ,--ADC Adds content of register 1 & 2 and stores in

  In RF.vhdl

signal R1 : std_logic_vector(15 downto 0) := "0000000000000001";
signal R2 : std_logic_vector(15 downto 0) := "0000000000000010";
signal R3 : std_logic_vector(15 downto 0) := "0000000000000000";

Other signals can be taken as x \0000"
```

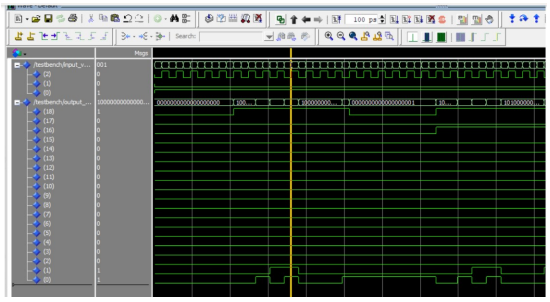


Figure 2: ADC if carry flag is 1

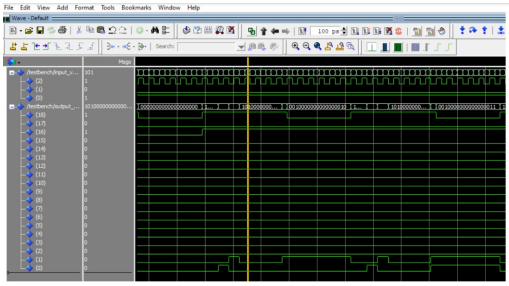


Figure 3: ADC if carry flag is 0

4.3 ADZ

```
    Memory:

  0 => "0000_001_010_011_001" ,--ADZ Adds content of register 1 & 2 and stores in

  In RF.vhdl

signal R1 : std_logic_vector(15 downto 0) := "0000000000000001";
signal R2 : std_logic_vector(15 downto 0) := "0000000000000010";
signal R3 : std_logic_vector(15 downto 0) := "0000000000000000";

Other signals can be taken as x \0000"
```

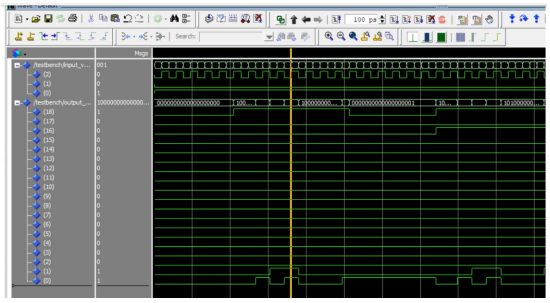


Figure 4: ADZ if Zero flag is 1

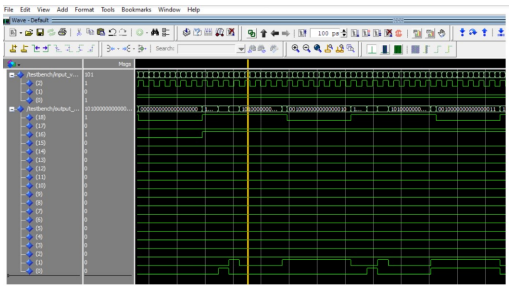


Figure 5: ADZ if Zero flag is 0

4.4 ADI

```
Memory:

0 => "0001001010111110",--ADI Adds content of register 1 & 2 and stores in reg

Other memory commented
In RF.vhdl
signal R1 : std_logic_vector(15 downto 0) := "0000000000000001";
signal R2 : std_logic_vector(15 downto 0) := "0000000000000010";

Other Registers can be taken as x \0000"
```

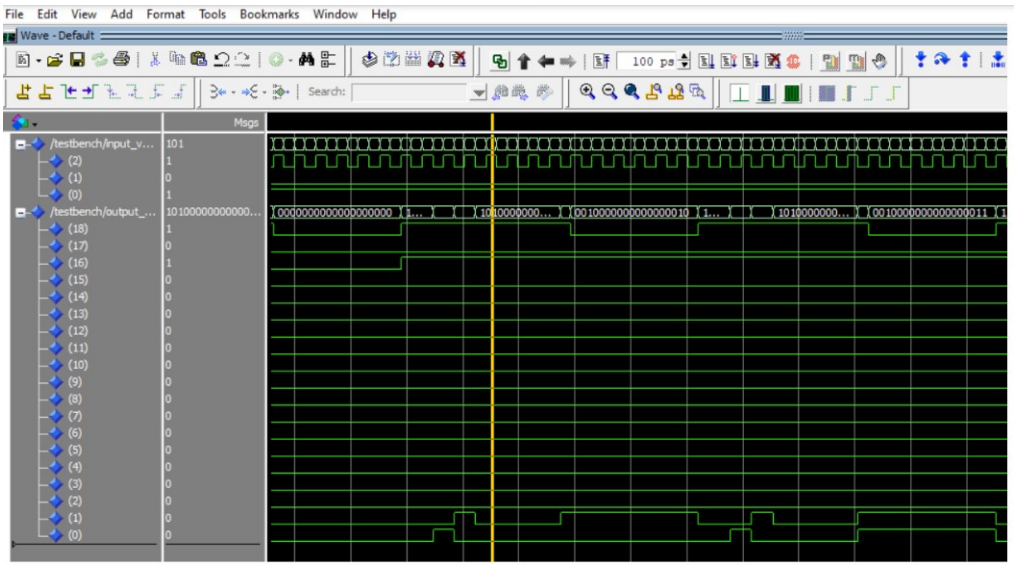


Figure 6: ADI

4.5 NDU

```
-- for NDU,NDC,NDZ testcase we will initialise registers to following values
-- signal R1 : std_logic_vector(15 downto 0) := "0000000000000011";
-- signal R2 : std_logic_vector(15 downto 0) := "0000000000000101";
-- signal R0,R3,R4,R5,R6,R7:std_logic_vector(15 downto 0)
                                                    := "0000000000000000";

0 => "0010001010011000" ,--NDU nands content of reg 1 & reg 2 and stores in re
```

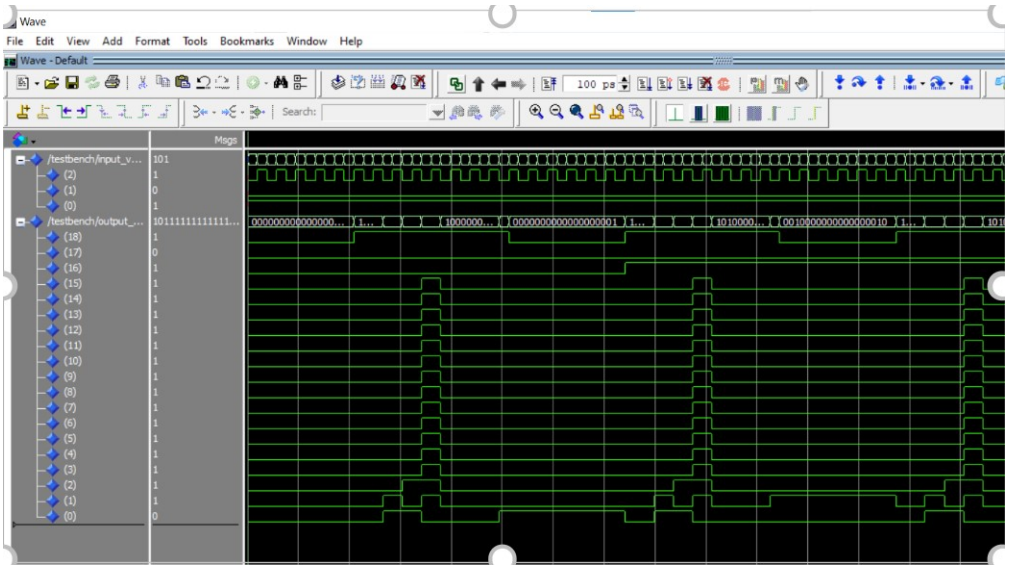


Figure 7: NDU

4.6 NDC

```
-- for NDU,NDC,NDZ testcase we will initialise registers to following values
-- signal R1 : std_logic_vector(15 downto 0) := "0000000000000011";
-- signal R2 : std_logic_vector(15 downto 0) := "0000000000000101";
-- signal R0,R3,R4,R5,R6,R7:std_logic_vector(15 downto 0)
                                                    := "0000000000000000";

0 => "0010001010011010" ,--NDU nands content of reg 1 & reg 2 and stores in re
```

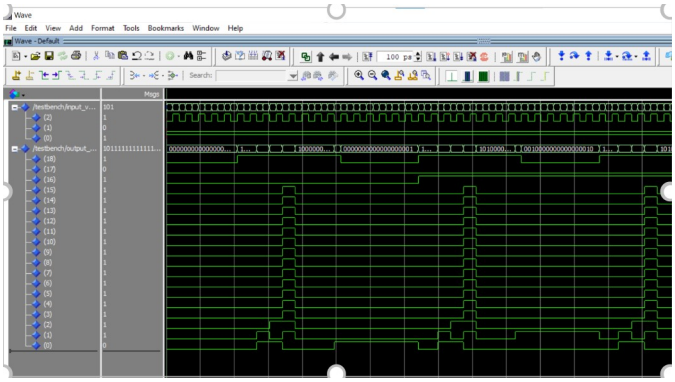


Figure 8: NDC if carry flag is 1

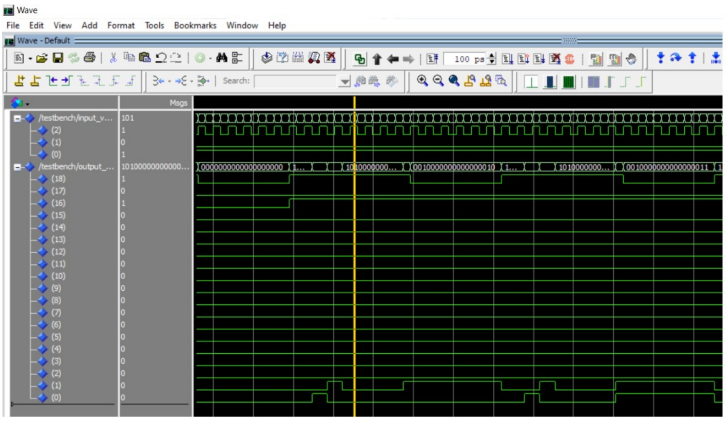


Figure 9: NDC if carry flag is 0

4.7 NDZ

```
-- for NDU,NDC,NDZ testcase we will initialise registers to following values
-- signal R1 : std_logic_vector(15 downto 0) := "0000000000000011";
-- signal R2 : std_logic_vector(15 downto 0) := "0000000000000101";
-- signal R0,R3,R4,R5,R6,R7:std_logic_vector(15 downto 0)
                                                    := "0000000000000000";

0 => "0010001010011001" ,--NDU nands content of reg 1 & reg 2 and stores in re
```

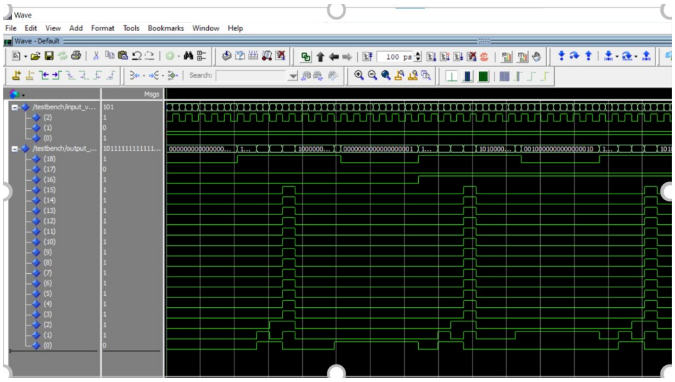


Figure 10: NDZ if zero flag is 1

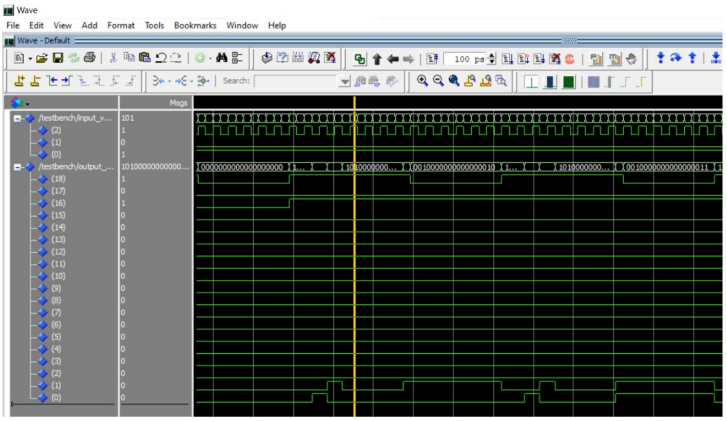


Figure 11: NDZ if zero flag is 0

4.8 LHI

```
-- for LHI load higher immediate,placing 9 bits immediate into MSBs of reg A &
-- signal R0 : std_logic_vector(15 downto 0) := "0000000000000001";
-- signal R1 : std_logic_vector(15 downto 0) := "0000000000000010";
-- signal R2 :std_logic_vector(15 downto 0) := "0000000000000100";
-- signal R3 : std_logic_vector(15 downto 0) := "0000000000001000";
-- signal R4 : std_logic_vector(15 downto 0) := "0000000000010000";
-- signal R5 : std_logic_vector(15 downto 0) := "0000000000100000";
-- signal R6 : std_logic_vector(15 downto 0) := "0000000001000000";
-- signal R7 : std_logic_vector(15 downto 0) := "0000000010000000";

0 => "0011001000111111" ,--LHI
```

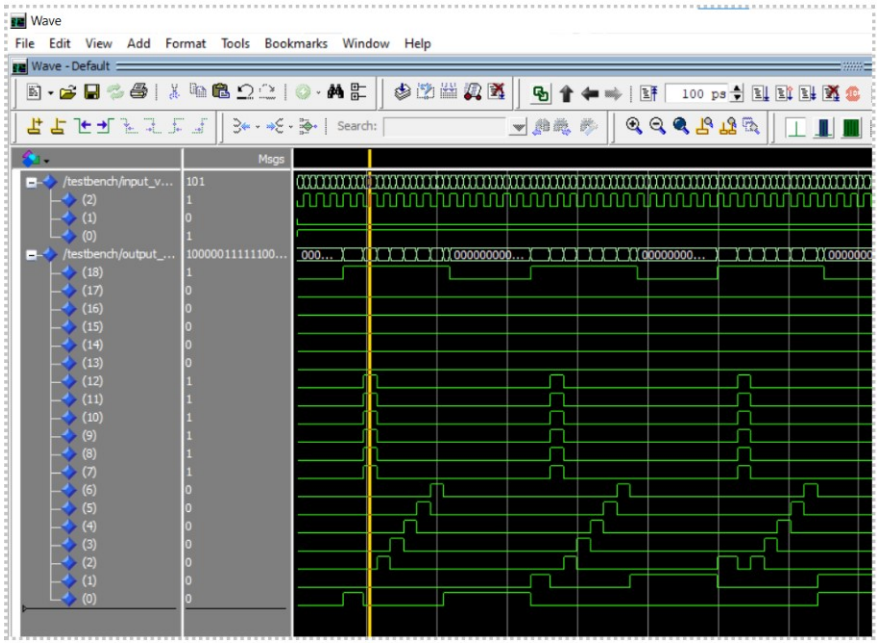


Figure 12: LHI

4.9 LW

```
   -- for LW load ,loading value from memory into reg A.
   -- Memory address is computed by adding immediate 6 bits with content of reg B

   -- signal R0 : std_logic_vector(15 downto 0) := "0000000000000001";
   -- signal R1 : std_logic_vector(15 downto 0) := "0000000000000000";
   -- signal R2 :std_logic_vector(15 downto 0) := "0000000000000100";
   -- signal R3 : std_logic_vector(15 downto 0) := "0000000000001000";
   -- signal R4 : std_logic_vector(15 downto 0) := "0000000000010000";
   -- signal R5 : std_logic_vector(15 downto 0) := "0000000000100000";
   -- signal R6 : std_logic_vector(15 downto 0) := "0000000001000000";
   -- signal R7 : std_logic_vector(15 downto 0) := "0000000010000000";
   --here we will be adding Mem(conv_int(regB+SE 6))
   -- 0 => "0100001000000010" ,--LW   here 6 immediate bits correspond to 2
-- 3=>  "0000000001111111",
```

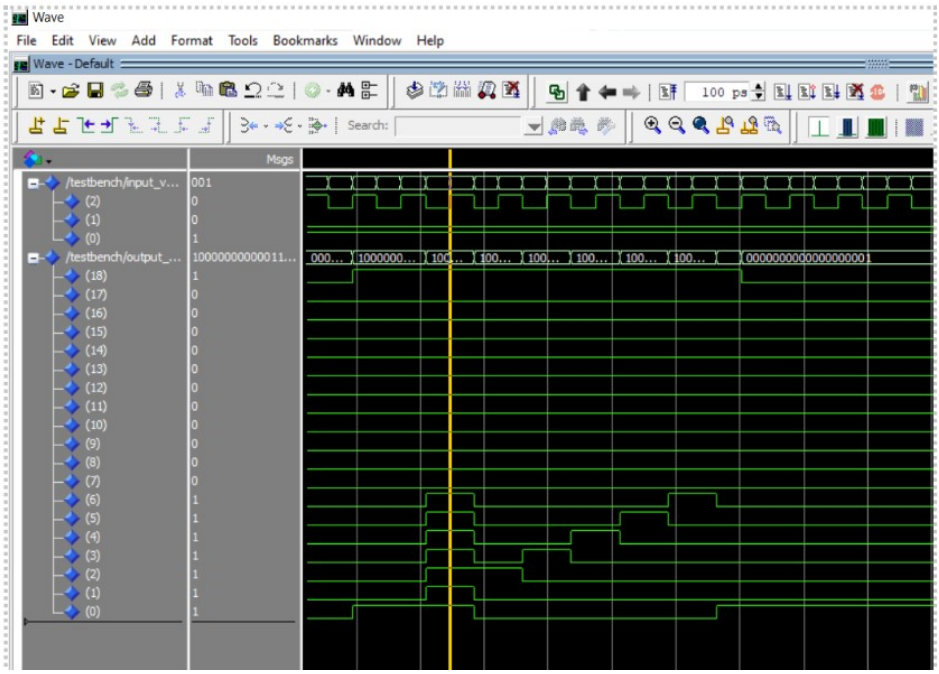


Figure 13: LW

4.10 LM

```
--   Memory for LM
0=> "0110000001111111",
1 => "1111111111111111" ,
2 => "1111111111111110" ,
3 => "1111111111111100" ,
4 => "1111111111111000" ,
5 => "1111111111110000" ,
6 => "1111111111100000" ,
7 => "1111111111000000" , --here we will be adding Mem(conv_int(regB+SE 6))
signal R0 : std_logic_vector(15 downto 0) := "0000000000000001";
signal R1 : std_logic_vector(15 downto 0) := "0000000000000000";
signal R2 :std_logic_vector(15 downto 0) := "0000000000000100";
signal R3 : std_logic_vector(15 downto 0) := "0000000000001000";
signal R4 : std_logic_vector(15 downto 0) := "0000000000010000";
signal R5 : std_logic_vector(15 downto 0) := "0000000000100000";
signal R6 : std_logic_vector(15 downto 0) := "0000000001000000";
signal R7 : std_logic_vector(15 downto 0) := "0000000010000000";
```

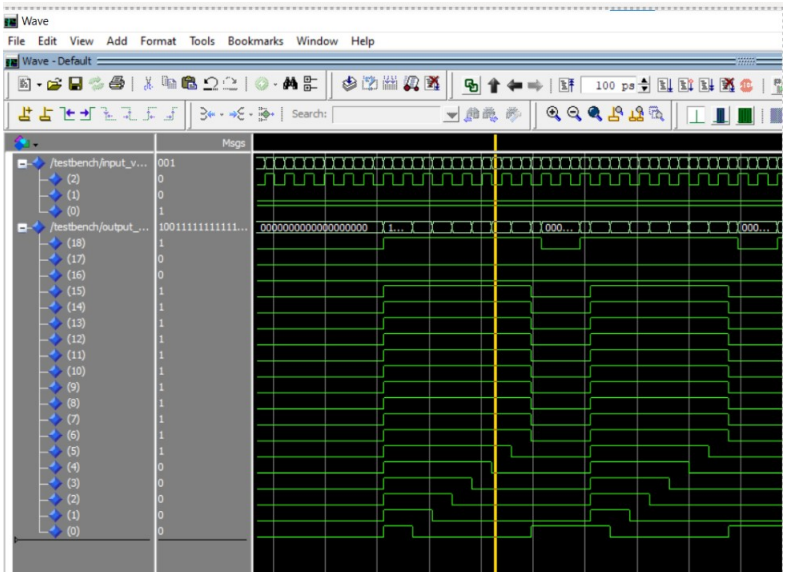


Figure 14: LM

4.11 BEQ

```
-- signal R0 : std_logic_vector(15 downto 0) := "0000000000000001";
-- signal R1 : std_logic_vector(15 downto 0) := "0000000000000010";
-- signal R2 :std_logic_vector(15 downto 0)  := "0000000000000100";
-- signal R3 : std_logic_vector(15 downto 0) := "0000000000001000";
-- signal R4 : std_logic_vector(15 downto 0) := "0000000000010000";
-- signal R5 : std_logic_vector(15 downto 0) := "0000000000100000";
-- signal R6 : std_logic_vector(15 downto 0) := "0000000001000000";
-- signal R7 : std_logic_vector(15 downto 0) := "0000000010000000";   --Memory
--  0 => "1000_000_000_000_110",
```

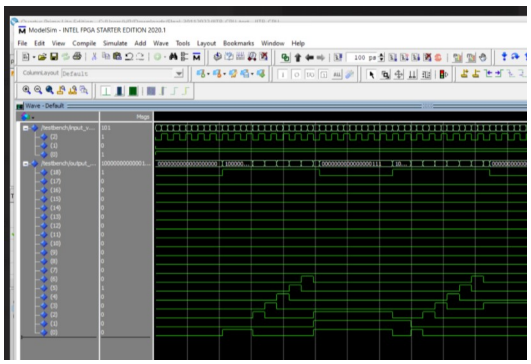


Figure 15: BEQ when content of registers are equal

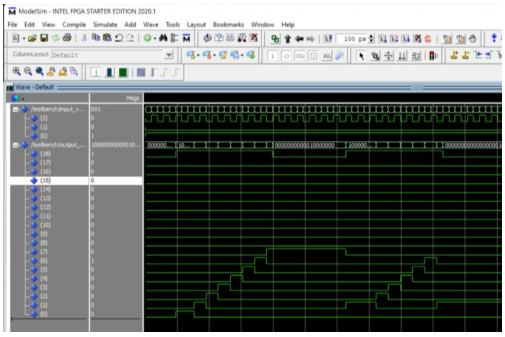
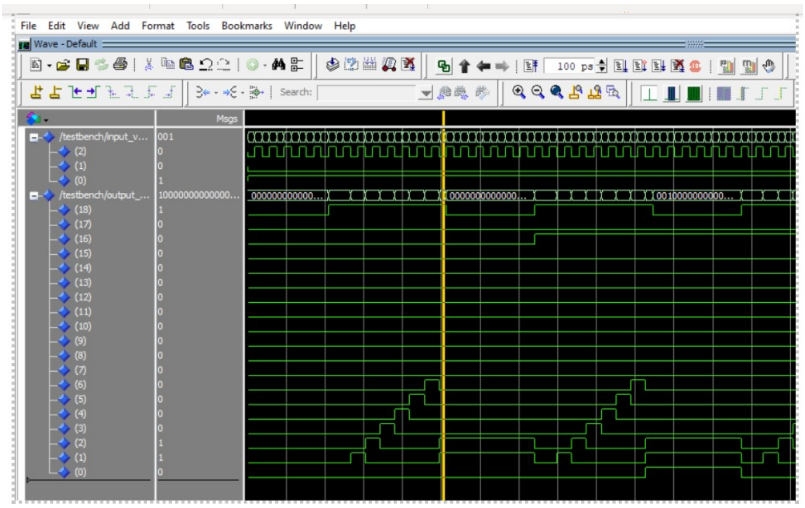


Figure 16: BEQ when content of registers are NOT equal

4.12 JAL

```
--JAl for Jump and Link
-- signal R0 : std_logic_vector(15 downto 0) := "0000000000000001";
-- signal R1 : std_logic_vector(15 downto 0) := "0000000000000010";
-- signal R2 :std_logic_vector(15 downto 0)  := "0000000000000100";
-- signal R3 : std_logic_vector(15 downto 0) := "0000000000001000";
-- signal R4 : std_logic_vector(15 downto 0) := "0000000000010000";
-- signal R5 : std_logic_vector(15 downto 0) := "0000000000100000";
-- signal R6 : std_logic_vector(15 downto 0) := "0000000001000000";
-- signal R7 : std_logic_vector(15 downto 0) := "0000000010000000"; signal R0
--Memory for JAL
--  0 => "1000_000_000_000_110",
```



We see that the R7 which stores the PC has branched to PC+Imm (0+6)

Figure 17: JAL

4.13 JLR

```
-- -- JLR for Jump and Link Register
-- --according to given input PC R0 should contain
-- signal R0 : std_logic_vector(15 downto 0) := "0000000000000001";
-- signal R1 : std_logic_vector(15 downto 0) := "0000000000000011";
-- signal R2 :std_logic_vector(15 downto 0)  := "0000000000000100";
-- signal R3 : std_logic_vector(15 downto 0) := "0000000000001000";
-- signal R4 : std_logic_vector(15 downto 0) := "0000000000010000";
-- signal R5 : std_logic_vector(15 downto 0) := "0000000000100000";
-- signal R6 : std_logic_vector(15 downto 0) := "0000000001000000";
-- signal R7 : std_logic_vector(15 downto 0) := "0000000010000000";   -- Memory
--  0 => "1001_000_001_000000", --Memory
```

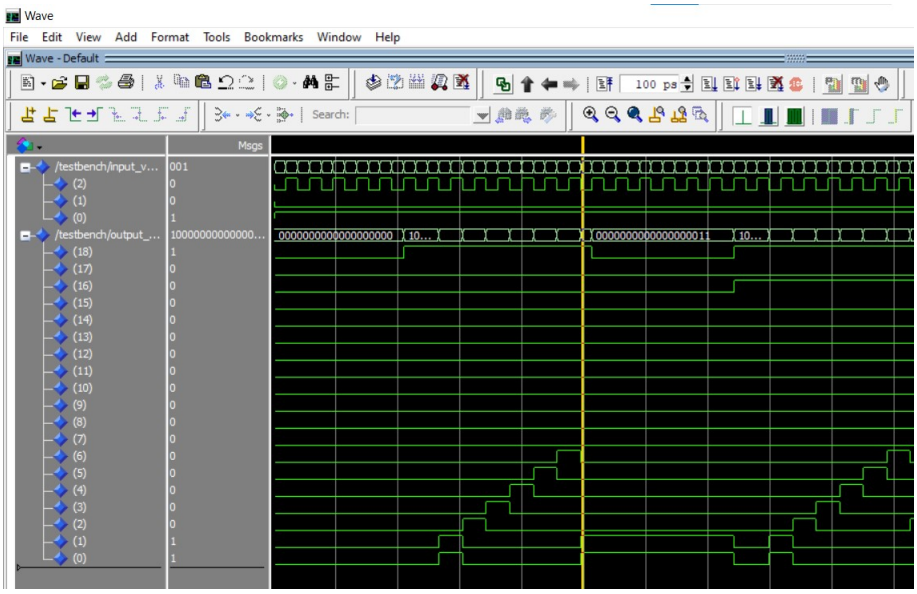


Figure 18: JLR