Christopher Lall

CSc 342/343 – Professor Gertner

Lab Assignment – BEQ_BNE_J

Spring 2022

Due 4/24/22

I will neither give nor reieve unauthorized assistance on this TEST. I will only use one computing device to perform this TEST, I will not use cell while performing this TEST.

Chris L

Table of Contents

Objective:

This lab's goal is to teach me how to utilize a comparator to determine whether or not a condition has been met. Through this lab, I learned how the MIPS processor's BNE, BEQ, and J functions. To get to this answer, we had to employ many components such as a pc adder, mux, pc register, general register, instruction register, controller, and finally a test bench to determine if our design was operating properly.

Components/VHDL Code:

2:1 Mux

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3
 4  entity Lall_2to1_Mux is
 5     port ( D0, D1 : in STD_LOGIC_VECTOR(31 downto 0);
 6            SEL    : in STD_LOGIC;
 7            OUT1   : out STD_LOGIC_VECTOR(31 downto 0));
 8  end Lall_2to1_Mux;
 9
10  architecture behavioral of Lall_2to1_Mux is
11  begin
12     OUT1(0) <= (D0(0) AND (NOT SEL)) OR (D1(0) AND SEL);
13     OUT1(1) <= (D0(1) AND (NOT SEL)) OR (D1(1) AND SEL);
14     OUT1(2) <= (D0(2) AND (NOT SEL)) OR (D1(2) AND SEL);
15     OUT1(3) <= (D0(3) AND (NOT SEL)) OR (D1(3) AND SEL);
16     OUT1(4) <= (D0(4) AND (NOT SEL)) OR (D1(4) AND SEL);
17     OUT1(5) <= (D0(5) AND (NOT SEL)) OR (D1(5) AND SEL);
18     OUT1(6) <= (D0(6) AND (NOT SEL)) OR (D1(6) AND SEL);
19     OUT1(7) <= (D0(7) AND (NOT SEL)) OR (D1(7) AND SEL);
20      OUT1(8) <= (D0(8) AND (NOT SEL)) OR (D1(8) AND SEL);
21     OUT1(9) <= (D0(9) AND (NOT SEL)) OR (D1(9) AND SEL);
22     OUT1(10) <= (D0(10) AND (NOT SEL)) OR (D1(10) AND SEL);
23     OUT1(11) <= (D0(11) AND (NOT SEL)) OR (D1(11) AND SEL);
24     OUT1(12) <= (D0(12) AND (NOT SEL)) OR (D1(12) AND SEL);
25     OUT1(13) <= (D0(13) AND (NOT SEL)) OR (D1(13) AND SEL);
26     OUT1(14) <= (D0(14) AND (NOT SEL)) OR (D1(14) AND SEL);
27     OUT1(15) <= (D0(15) AND (NOT SEL)) OR (D1(15) AND SEL);
28      OUT1(16) <= (D0(16) AND (NOT SEL)) OR (D1(16) AND SEL);
29     OUT1(17) <= (D0(17) AND (NOT SEL)) OR (D1(17) AND SEL);
30     OUT1(18) <= (D0(18) AND (NOT SEL)) OR (D1(18) AND SEL);
31     OUT1(19) <= (D0(19) AND (NOT SEL)) OR (D1(19) AND SEL);
32     OUT1(20) <= (D0(20) AND (NOT SEL)) OR (D1(20) AND SEL);
33     OUT1(21) <= (D0(21) AND (NOT SEL)) OR (D1(21) AND SEL);
34      OUT1(22) <= (D0(22) AND (NOT SEL)) OR (D1(22) AND SEL);
35     OUT1(23) <= (D0(23) AND (NOT SEL)) OR (D1(23) AND SEL);
36     OUT1(24) <= (D0(24) AND (NOT SEL)) OR (D1(24) AND SEL);
37     OUT1(25) <= (D0(25) AND (NOT SEL)) OR (D1(25) AND SEL);
38     OUT1(26) <= (D0(26) AND (NOT SEL)) OR (D1(26) AND SEL);
```

Figure 1. 2:1 mux vhdl code pt 1

```
39     OUT1(27) <= (D0(27) AND (NOT SEL)) OR (D1(27) AND SEL);
40      OUT1(28) <= (D0(28) AND (NOT SEL)) OR (D1(28) AND SEL);
41     OUT1(29) <= (D0(29) AND (NOT SEL)) OR (D1(29) AND SEL);
42     OUT1(30) <= (D0(30) AND (NOT SEL)) OR (D1(30) AND SEL);
43     OUT1(31) <= (D0(31) AND (NOT SEL)) OR (D1(31) AND SEL);
44
45  end behavioral;
```

Figure 2. 2:1 mux vhdl code pt 2

32bit Register:

```
1   -- Program Counter
2   LIBRARY IEEE;
3   USE IEEE.STD_LOGIC_1164.ALL;
4
5   entity Lall_32Bit_Register is
6       port (
7           Lall_CLK  : in std_logic;
8           Lall_WREN : in std_logic;
9           Lall_RDEN : in std_logic;
10          Lall_CHEN : in std_logic;
11          Lall_DATA : in std_logic_vector(31 downto 0);
12          Lall_Curr_PC    : out std_logic_vector(31 downto 0));
13  end Lall_32Bit_Register;
14
15  architecture arch of Lall_32Bit_Register is
16      signal Lall_New_PC : std_logic_vector(31 downto 0);
17
18  begin
19      process (Lall_CLK)
20      begin
21          if (rising_edge(Lall_CLK) and Lall_WREN = '1')
22              then Lall_New_PC <= Lall_DATA;
23          end if;
24      end process;
25
26      process(Lall_RDEN, Lall_CHEN, Lall_New_PC)
27      begin
28          if (Lall_RDEN = '1' and Lall_CHEN = '1')
29              then Lall_Curr_PC <= Lall_New_PC;
30          elsif(Lall_CHEN = '0')
31              then Lall_Curr_PC <= (others => 'Z');
32          end if;
33      end process;
34  end arch;
```

Figure 3. 32bit register vhdl code

Data Memory:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Lall_DataMemory is
    generic (K:integer:=32; -- Number of bits per word
        A:integer:=5); -- Number of address bits
    port(
        clock: in std_logic;
        wren : in std_logic :='0'; -- Set to 0 to initially read from memory :: Set 1 to initially write to memory
         data: in std_logic_vector(K-1 downto 0);
        address: in std_logic_vector(A-1 downto 0);
        q : out std_logic_vector(K-1 downto 0));
end Lall_DataMemory;

architecture arch of Lall_DataMemory is
    type mem is array(0 to 15) of std_logic_vector(K-1 downto 0);
    signal mem_array: mem;

    begin
    process(clock)
    begin
     if(clock'event and clock = '1') then
         if (wren = '1') then
             mem_array(to_integer(unsigned(address))) <= data;
         end if;
     end if;
     end process;

     q <= mem_array(to_integer(unsigned(address)));

 end arch;
```

Figure 4. data mem vhdl code

Instruction memory:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Lall_InstructionMemory is
    generic (K:integer:=32; -- Number of bits per word
        A:integer:=5); -- Number of address bits
    port(
        clock: in std_logic;
        wren : in std_logic :='0'; -- Set to 0 to initially read from memory :: Set 1 to initially write to memory
             data: in std_logic_vector(K-1 downto 0);
        address: in std_logic_vector(A-1 downto 0);
        q : out std_logic_vector(K-1 downto 0));
end Lall_InstructionMemory;

architecture arch of Lall_InstructionMemory is
    type mem is array(0 to 31) of std_logic_vector(K-1 downto 0);
    signal mem_array: mem;

    begin
    process(clock)
    begin
      if(clock'event and clock = '1') then
     if (wren = '1') then -- Write TO memory
        mem_array(to_integer(unsigned(address))) <= data;
     elsif (wren = '0') then -- Read FROM memory
        mem_array <= ( 0 => "00111100000000010001000000000001", -- lui MIPS Instruction
                       1 => "00001000000100000000000000001100", -- j MIPS Instruction
                       2 => "00000010001100101000000000100000", -- add MIPS Instruction
                       3 => "00000010001100101000000000100010", -- sub MIPS Instruction
                       4 => "00010110000100110000000000000010", -- bne MIPS Instruction
                  others =>"00000000000000000000000000000000");
        q <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
     end if;
        end if;
        q <= mem_array(to_integer(unsigned(address)));
     end process;


end arch;
```

Figure 5. instruction mem vhdl code

Instruction Register:

```
-- Instruction Register (IR)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity Lall_Instruction_Register is
    port( Instruction : in std_logic_vector(31 downto 0);
          Opcode : out std_logic_vector(5 downto 0); -- 6 bit Opcode
          Rs, Rt : out std_logic_vector(4 downto 0); -- 5 Bit addresses
          Immediate : out std_logic_vector(15 downto 0)); -- 16 Bit Immediate
end Lall_Instruction_Register;

architecture arch of Lall_Instruction_Register is
begin

Opcode <= Instruction(31 downto 26); -- Needed?
Rs <= Instruction(25 downto 21);
Rt <= Instruction(20 downto 16);
Immediate <= Instruction(15 downto 0);

end arch;
```

Figure 6. instruction reg vhdl code

Register file:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Lall_RegisterFile is
   port(
      clock: in std_logic;
      wren : in std_logic;
      rst  : in std_logic;
      data : in std_logic_vector(31 downto 0);
      waddress  : in std_logic_vector(4 downto 0);
      raddress1 : in std_logic_vector(4 downto 0);
      raddress2 : in std_logic_vector(4 downto 0);
      q1 : out std_logic_vector(31 downto 0);
      q2 : out std_logic_vector(31 downto 0));
end Lall_RegisterFile;

architecture arch of Lall_RegisterFile is
   type reg is array(0 to 31) of std_logic_vector(31 downto 0);
   signal reg_array: reg := (
      0 => X"00C3A23B",
      5 => X"3FFD02E1",
      9 => X"00C3A23B",
      13 => X"FFFFEEEE",
      21 => X"FFFFEEEE",
      27 => X"224CCC80",
      others => X"00000000");

   begin

   process(rst, clock)
   begin
    if (rst = '1') then
      reg_array <= (others => (others => '0'));
    elsif (clock'event and clock = '1') then
      if (wren = '1') then
         reg_array(to_integer(unsigned(waddress))) <= data;
      end if;
    end if;
       q1 <= reg_array(to_integer(unsigned(raddress1)));
       q2 <= reg_array(to_integer(unsigned(raddress2)));
    end process;
```

Figure 7. reg file vhdl code

NAL Unit:

```
1   Library IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.NUMERIC_STD.ALL;
4   USE WORK.Lall_Components_Package.ALL;
5
6   entity Lall_NAL_Unit is -- A unit to compute the next address of the next instruction via BEQ, BNE, J MIPS instruction
7      port ( Clock1, Clock2 : in std_logic;
8             Instruction : in std_logic_vector(31 downto 0); -- 32 Bit instruction
9             PC_In : in std_logic_vector(31 downto 0);
10            InstrAddr : out std_logic_vector(31 downto 0));
11  end Lall_NAL_Unit;
12
13  architecture arch of Lall_NAL_Unit is
14
15  signal Opcode : std_logic_vector(5 downto 0); -- Do I need this?
16  signal Rs, Rt : std_logic_vector(4 downto 0); -- Registers address ports (indicies) of RS, Rt
17  signal Imm    : std_logic_vector(15 downto 0); -- 16 Bit Immediate
18  signal BusA, BusB : std_logic_vector(31 downto 0);-- Output data registers Rs and Rt to be compared
19  signal EqualCond : std_logic; -- Condition determining whether a branching occurs (i.e. PcSrc)
20  signal PC_Out : std_logic_vector(31 downto 0); -- 32 Bit register
21  signal ImmExt : std_logic_vector(31 downto 0); -- Signed Exetended 32-Bit Imm
22  signal Addr1, Addr2 : std_logic_vector(31 downto 0); -- For calculating the two possible Instruction Addresses
23  signal NextAddr : std_logic_vector(31 downto 0); -- Final Address to be chosen by 2-1 Mux
24
25  begin
26
27  -- Instruction Register to output Opcode, Rs, Rt Indicies, and Immediate field value
28  U0: Lall_Instruction_Register port map(Instruction, Opcode, Rs, Rt, Imm);
29  -- Register File (values for CLK, WREN, RST, and Wraddress don't matter since we are not writing to any register)
30  U1: Lall_RegisterFile port map(Clock2, '0', '0', X"00000000", "00000", Rs, Rt, BusA, BusB);
31  -- Compare if BusA and BusB of Reg file are equal and set EqualCond flag accordingly
32  U2: Lall_Comparator port map(BusA, BusB, EqualCond);
33  -- Sign extend the 16-Bit Immediate field
34  U3: Lall_Sign_Ext port map (Imm, ImmExt);
35  -- Program Counter (All that matters is Clock, PC_In are provided. WREN = RDEN = CHEN = 1)
36  U4: Lall_32Bit_Register port map(Clock1, '1', '1', '1', PC_In, PC_Out);
37  -- Compute the two possible addresses
38  U5: Lall_LPM_Adder port map(PC_Out, X"00000004", Addr1); -- PC + 4
39  U6: Lall_LPM_Adder port map(Addr1, ImmExt, Addr2); -- PC + ImmExt + 4
40  -- Select the appropriate address and store it in NextAddr (Do I need to feed it to PC Reg Again?)
41  U7: Lall_2to1_Mux port map(Addr1, Addr2, EqualCond, NextAddr);
42  InstrAddr <= NextAddr; -- Address of next instruction
43  end arch;
```

Figure 8. NAL unit vhdl code

Sign Ext:

```
1   Library IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.NUMERIC_STD.ALL;
4
5   Entity Lall_Sign_Ext is
6       port( A      : in std_logic_vector(15 downto 0);
7             Output: out std_logic_vector(31 downto 0));
8   end Lall_Sign_Ext;
9
10  architecture behvaioral of Lall_Sign_Ext is
11  begin
12      Output <= std_logic_vector(resize(signed(A), 32));
13  end behvaioral;
```

Figure 9. sign ext vhdl code

LPM Comparator:

```
37  LIBRARY ieee;
38  USE ieee.std_logic_1164.all;
39
40  LIBRARY lpm;
41  USE lpm.all;
42
43  ENTITY Lall_Comparator IS
44     PORT
45     (
46        dataa     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
47        datab     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
48        aeb       : OUT STD_LOGIC
49     );
50  END Lall_Comparator;
51
52
53  ARCHITECTURE SYN OF lall_comparator IS
54
55     SIGNAL sub_wire0  : STD_LOGIC ;
56
57
58
59     COMPONENT lpm_compare
60     GENERIC (
61        lpm_representation      : STRING;
62        lpm_type    : STRING;
63        lpm_width       : NATURAL
64     );
65     PORT (
66           dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
67           datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
68           aeb   : OUT STD_LOGIC
69     );
70     END COMPONENT;
71
72  BEGIN
73     aeb    <= sub_wire0;
```

Figure 10. LPM Comparator vhdl code pt 1

```
75      LPM_COMPARE_component : LPM_COMPARE
76      GENERIC MAP (
77          lpm_representation => "UNSIGNED",
78          lpm_type => "LPM_COMPARE",
79          lpm_width => 32
80      )
81      PORT MAP (
82          dataa => dataa,
83          datab => datab,
84          aeb => sub_wire0
85      );
```

Figure 11. LPM Comparator vhdl code pt 2

LPM Adder:

```
37  LIBRARY ieee;
38  USE ieee.std_logic_1164.all;
39
40  LIBRARY lpm;
41  USE lpm.all;
42
43  ENTITY Lall_LPM_Adder IS
44     PORT
45     (
46        dataa     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
47        datab     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
48        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
49     );
50  END Lall_LPM_Adder;
51
52
53  ARCHITECTURE SYN OF lall_lpm_adder IS
54
55     SIGNAL sub_wire0   : STD_LOGIC_VECTOR (31 DOWNTO 0);
56
57
58
59     COMPONENT lpm_add_sub
60     GENERIC (
61        lpm_direction     : STRING;
62        lpm_hint    : STRING;
63        lpm_representation      : STRING;
64        lpm_type    : STRING;
65        lpm_width      : NATURAL
66     );
67     PORT (
68           dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
69           datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
70           result   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
71     );
72     END COMPONENT;
73
74  BEGIN
75     result    <= sub_wire0(31 DOWNTO 0);
76
77     LPM_ADD_SUB_component : LPM_ADD_SUB
78     GENERIC MAP (
79        lpm_direction => "ADD",
```

Figure 12. LPM Adder vhdl code pt 1

```
80          lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
81          lpm_representation => "UNSIGNED",
82          lpm_type => "LPM_ADD_SUB",
83          lpm_width => 32
84      )
85      PORT MAP (
86          dataa => dataa,
87          datab => datab,
88          result => sub_wire0
89      );
```

Figure 13. LPM Adder vhdl code pt 2.

Components Package:

```
1   Library IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3
4   package Lall_Components_Package is
5
6   component Lall_LPM_Adder -- Adder to compute instruction addresses
7      PORT(
8          dataa    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
9          datab    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
10         result   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0));
11  end component;
12
13  component Lall_2to1_Mux -- 2 to 1 (32 Bit) multiplexer to select address
14      port ( D0, D1 : in STD_LOGIC_VECTOR(31 downto 0);
15           SEL     : in STD_LOGIC;
16           OUT1    : out STD_LOGIC_VECTOR(31 downto 0));
17  end component;
18
19  component Lall_Comparator -- 32 Bit comparator (compares contents of BusA and BusB data lines)
20      PORT(
21          dataa    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
22          datab    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
23          aeb      : OUT STD_LOGIC );
24  end component;
25
26  component Lall_32Bit_Register -- 32 Bit Register for PC
27      port ( Lall_CLK  : in std_logic;
28             Lall_WREN : in std_logic;
29             Lall_RDEN : in std_logic;
30             Lall_CHEN : in std_logic;
31             Lall_DATA : in std_logic_vector(31 downto 0);
32             Lall_Curr_PC : out std_logic_vector(31 downto 0));
33  end component;
34
35
36  component Lall_RegisterFile --
37     port( clock: in std_logic;
38           wren : in std_logic;
39           rst  : in std_logic;
40           data : in std_logic_vector(31 downto 0);
41           waddress  : in std_logic_vector(4 downto 0);
42           raddress1 : in std_logic_vector(4 downto 0);
43           raddress2 : in std_logic_vector(4 downto 0);
```

Figure 14. Components package vhdl code pt 1

```
44          q1 : out std_logic_vector(31 downto 0);
45          q2 : out std_logic_vector(31 downto 0));
46  end component;
47
48  component Lall_Instruction_Register
49     port( Instruction : in std_logic_vector(31 downto 0);
50           Opcode : out std_logic_vector(5 downto 0); -- 6 bit Opcode
51           Rs, Rt : out std_logic_vector(4 downto 0); -- 5 Bit addresses
52           Immediate : out std_logic_vector(15 downto 0)); -- 16 Bit Immediate
53  end component;
54
55  component Lall_Sign_Ext
56     port( A      : in std_logic_vector(15 downto 0);
57           Output: out std_logic_vector(31 downto 0));
58  end component;
59
60  end package;
```

Figure 15. Components package vhdl code pt 2

Predefined arbitrary data will be stored in the register file component at the arbitrary address indices Rs and Rt. The corresponding data will be transferred to outputs BusA and BusB based on the values of Rs and Rt (which are fed from the IR). WREN, RST, Waddress, and Data are all present as inputs, and behaviors are defined, but they will not be used in this lab. Because we aren't writing to any registers, the inputs could be eliminated if necessary.

The 32-Bit Comparator will compare the outputs of the Register File (i.e. BusA and BusB) to set a condition flag CondEqual either to a 1 or a 0.

The Program Counter is a 32-bit register with the primary purpose of updating the current PC's value. At the rising edge of the clock, the PC value will be updated.

The sign extender merely converts the 16-bit Immediate field to 32-bits, which is required to identify the following instruction's address. To retain the sign of the value, the remaining 16 bits will be filled with either 0's or F's depending on whether the most significant bit is positive (0–7) or negative (8–F).

Two adder components will be used to compute the two possible PC values as described earlier.

Another required component is a 2:1 Multiplexer which will use the CondEqual flag to select one of the two computed PC values to feed back into the PC and give the final PC value.

A package is created to store all components.

With the utilization of each component, we create a symbol so that we can realize the NAL unit

Simulation/Waveform:

BEQ:

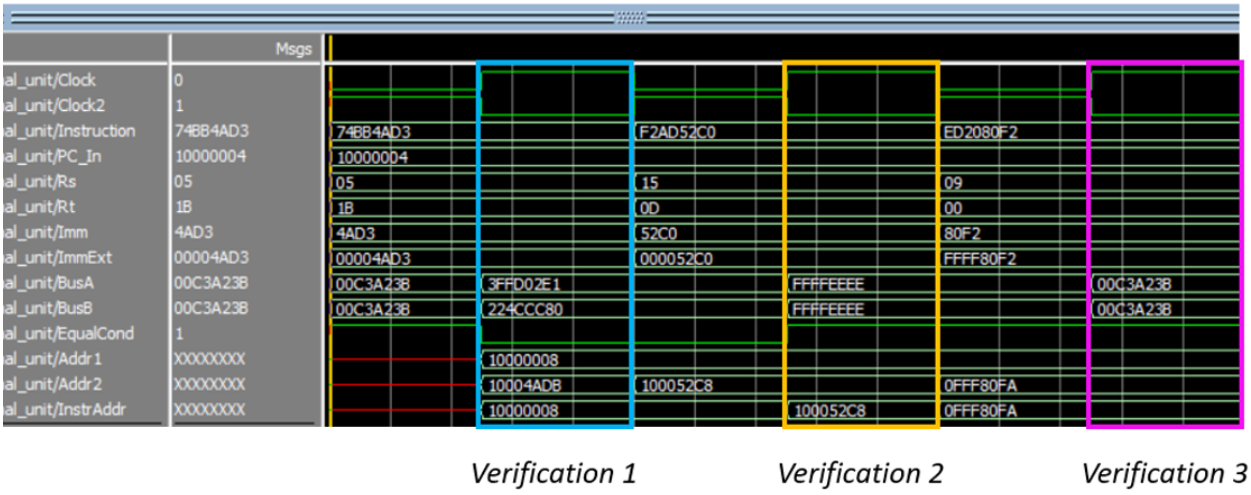

Figure 16. modelsim sim for BEQ instruction

BNE:

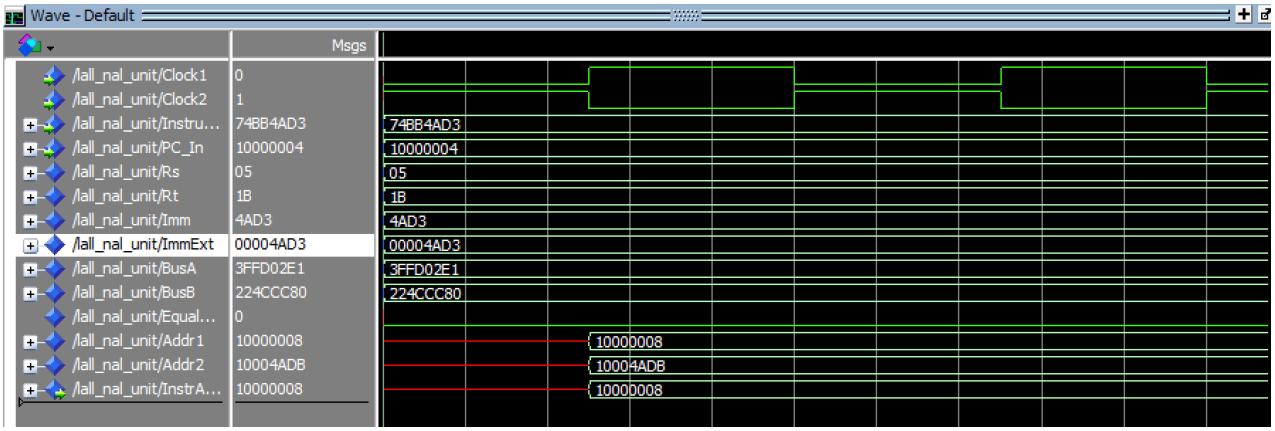

Figure 17. Verification 1

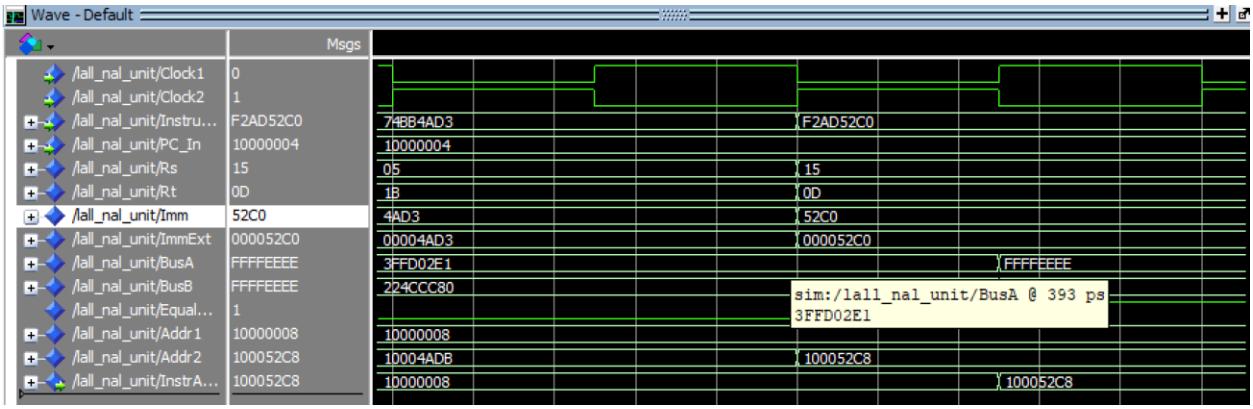

Figure 18. Verification 2

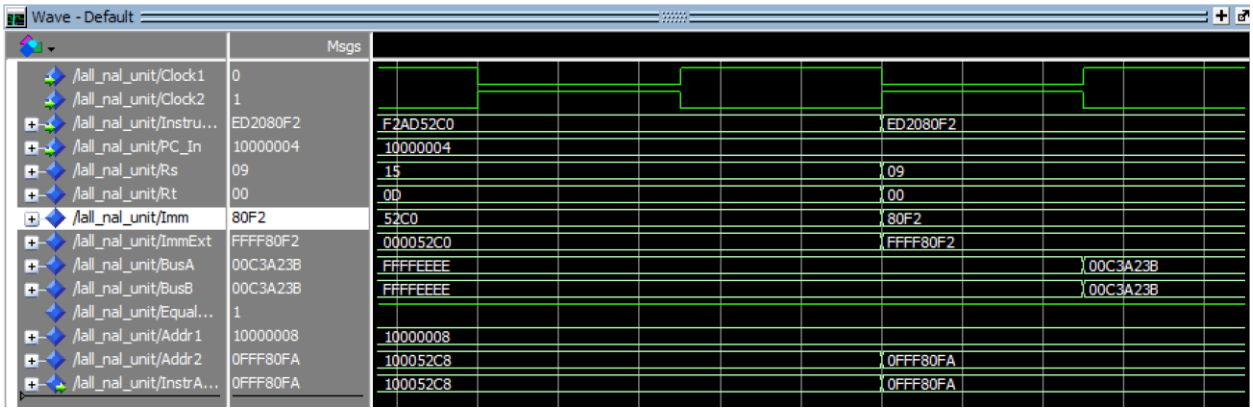

Figure 19. Verification 3

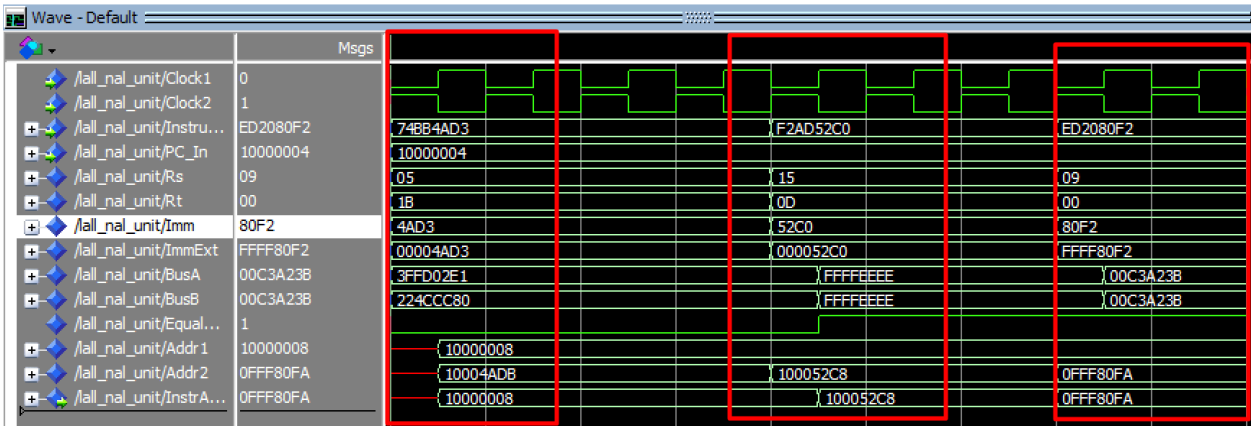

Figure 20. Total verification

Explanation analysis:

BEQ:

Three verifications will be performed to ensure the unit's accuracy. A 32-bit arbitrary instruction is

supplied into the Instruction Register (IR) for each verification, which decodes the Rs and Rt address

indices as well as the 16-bit Immediate. These indices are used as inputs to the register file, which uses them to store predefined arbitrary 32-bit data. After then, the 16-Bit Immediate is sign expanded. This information is sent to BusA and BusB, which are compared using a comparator and a condition flag is set (1 if equal, 0 if not). With a predefined arbitrary program counter value, the two possible addresses are computed using an adder and are fed to a 2:1 multiplexer which will select the final address of the next instruction based on the condition flag set earlier (i.e. select address 2 if condition flag is 1, else select address 1).

Verification 1: The IR was programmed with the arbitrary 32-bit instruction 0x74BB4AD3. This instruction is decoded by the IR to produce the indices Rs = 0x05 and Rt = 0x1B, as well as the 16-Bit Immediate field = 0x4AD3. 0x00004AD3 is the signed extended ImmExt. 0x10000004 is an arbitrary 32-bit PC value. For indices Rs and Rt, the arbitrary 32-bit data (stored in the register file) is 0x3FFD02E1 and 0x224CCC80, respectively. This information is sent to Bus A and Bus B. Due to the fact that the data stored on these buses is not equal, the condition flag is set to 0. As a result, the expected output (i.e. the next instruction's address) is PC + 4 (i.e. InstrAddr = 0x10000008).

Verification 2: We used an arbitrary 32-bit instruction 0xF2AD52C0 in the IR for the second verification. This instruction is decoded by the IR to produce the indices Rs = 0x15 and Rt = 0x0D, as well as the 16-Bit Immediate field = 0x52C0. 0x000052C0 is the signed extended ImmExt. The PC value remains the same. For indices Rs and Rt, the arbitrary 32-bit data (stored in the register file) is 0xFFFFEEEE and 0xFFFFEEE, respectively. This information is sent to Bus A and Bus B. Because the data stored on these buses is identical, the condition flag is set to 1. As a result, the intended output (i.e. the next instruction's location) is PC + ImmExt + 4. As a result, the NextAddr instruction's address is 0x100052C8.

Verification 3: We inserted an arbitrary 32-bit instruction 0xED2080F2 into the IR for the third verification. This instruction is decoded by the IR to produce the indices Rs = 0x09 and Rt = 0x00, as well

as the 16-Bit Immediate field = 0x80F2. 0xFFFF80F2 is the signed extended ImmExt. The PC value remains the same. For indices Rs and Rt, the arbitrary 32-bit data (stored in the register file) is 0x00C3A23B and 0x00C3A23B, respectively. This information is sent to Bus A and Bus B. Because the data stored on these buses is identical, the condition flag is set to 1. As a result, the intended output (i.e. the next instruction's location) is PC + ImmExt + 4. As a result, the NextAddr instruction's address is 0x0FFF80FA.

BNE:

Verification 1: The IR was programmed with the arbitrary 32-bit instruction 0x74BB4AD3. This instruction is decoded by the IR to provide address indices Rs = 0x05 and Rt = 0x1B, as well as 16-Bit Immediate field = 0x4AD3. 0x00004AD3 is the symbol extended instant (ImmExt). 0x10000004 is an arbitrary 32-bit PC value. For the indices Rs and Rt, the arbitrary 32-bit data (stored in the register file) is 0x3FFD02E1 and 0x224CCC80, respectively. This information is sent to Bus A and Bus B. Due to the fact that the data stored on these buses is not identical, the condition flag is set to 1. PC + ImmExt + 4 = 0x10004ADB is the address of the following instruction.

Verification 2: We used an arbitrary 32-bit instruction 0xF2AD52C0 in the IR for the second verification. This instruction is decoded by the IR to give address indices Rs = 0x15 and Rt = 0x0D, as well as 16-Bit Immediate field = 0x52C0. 0x000052C0 is the sign extended instantaneous (ImmExt). The PC value remains the same. For address indices Rs and Rt, the arbitrary 32-bit data (stored in the register file) is 0xFFFFEEEE and 0xFFFFEEE, respectively. This information is sent to Bus A and Bus B. Because the data on these buses is identical, the condition flag is set to 0. PC + 4 = 0x10000008 is the address of the following instruction.

Verification 3: We inserted an arbitrary 32-bit instruction 0xED2080F2 into the IR for the third verification. This instruction is decoded by the IR to provide address indices Rs = 0x09 and Rt = 0x00, as well as 16-Bit Immediate field = 0x80F2. 0xFFFF80F2 is the signed extended (ImmExt). The PC value

remains the same. For address indices Rs and Rt, the arbitrary 32-bit data (stored in the register file) is 0x00C3A23B and 0x00C3A23B, respectively. This information is sent to Bus A and Bus B. Because the data on these buses is identical, the condition flag is set to 0. PC + 4 = 0x10000008 is the address of the following instruction.