

Lab Report #2

Name: Ruochen Duan

Student ID: 1405106

The time I hand in this test is 2022/10/8, I wish to use the late token.

Explanation and Description

① 64bitADDER

I choose the behavioral for modeling type. The function of the adder is to find the sum of the two inputs-in0 and in1.

In the Lab2, I first built a 1bit full adder, and then connected 64 1bit full adders in series to form a 64-bit adder. Figure 1 is the code of the 1-bit full adder.

| Ln# | |
|-----|--|
| 1 | library ieee; |
| 2 | use ieee.std_logic_1164.all; |
| 3 | entity BIT_FULL_ADDER is |
| 4 | port (|
| 5 | A : in STD_LOGIC; |
| 6 | B : in STD_LOGIC; |
| 7 | Cin : in STD_LOGIC; |
| 8 | Sum : out STD_LOGIC; |
| 9 | Carry : out STD_LOGIC |
| 10 |); |
| 11 | end BIT_FULL_ADDER; |
| 12 | |
| 13 | architecture behv1 of BIT_FULL_ADDER is |
| 14 | begin |
| 15 | Sum <= A XOR B XOR Cin ; |
| 16 | Carry <= (A AND B) OR (Cin AND A) OR (Cin AND B) ; |
| 17 | end behv1; |
| 18 | |

Figure1

Then I built the testbench to test the function of 1-bit full adder. The testbench of mine is that(in the simulation, the signals in the simulation have the same names as the related ports.):

```
2
3  sim_proc: process
4  begin
5      A <= '0';
6      B <= '0';
7      Cin <= '0';
8      wait for 50 ns;
9
10     A <= '0';
11     B <= '0';
12     Cin <= '1';
13     wait for 50 ns;
14
15     A <= '0';
16     B <= '1';
17     Cin <= '0';
18     wait for 50 ns;
19
20     A <= '0';
21     B <= '1';
22     Cin <= '1';
23     wait for 50 ns;
24
25     A <= '1';
26     B <= '0';
27     Cin <= '0';
28     wait for 50 ns;
29
30     A <= '1';
31     B <= '0';
32     Cin <= '1';
33     wait for 50 ns;
34
35     A <= '1';
36     B <= '1';
37     Cin <= '0';
38     wait for 50 ns;
39
40     A <= '1';
41     B <= '1';
42     Cin <= '1';
```

The figure 2 is the result of the simulation.

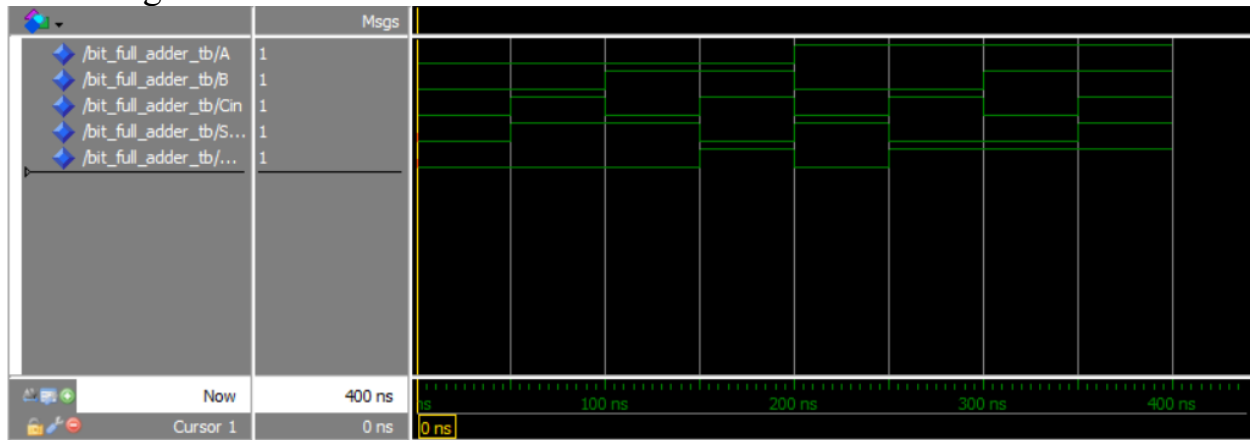


Figure 2

according to the truth table of the 1-bit full adder, the result is right.

| A _i | B _i | C _{i-1} | C _i | S _i |
|----------------|----------------|------------------|----------------|----------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

the truth table of the 1-bit full adder

Then I built the 64-bit adder by using 1-bit adder. Below is the code for the connection part.

```

signal temp:STD_LOGIC_VECTOR(64 downto 0);
begin
  temp(0) <= '0';
  ADDbuild:for i in 0 to 63 generate
  adderi:
    BIT_FULL_ADDER port map
    (
      A => in0(i),
      B => in1(i),
      Cin => temp(i),
      Carry => temp(i+1),
      Sum => output(i)
    );
  end generate ADDbuild;
end behv1;

```

Then I built the testbench to test the function of 64-bit full adder. The testbench of mine is that (in the simulation, the signals in the simulation have the same names as the related ports.):

```

stim_proc: process
begin

  in0 <= x"0000000000000000";
  in1 <= x"0000000000000001";

  wait for 100 ns;

  in0 <= x"c1111111111111111";
  in1 <= x"0eeeeeeeeeeeeeeee";

  wait for 100 ns;

  in0 <= x"1010101010101010";
  in1 <= x"0101010101010101";

  wait for 100 ns;

end process;

```

Figure 3 and 4 is the simulation result.

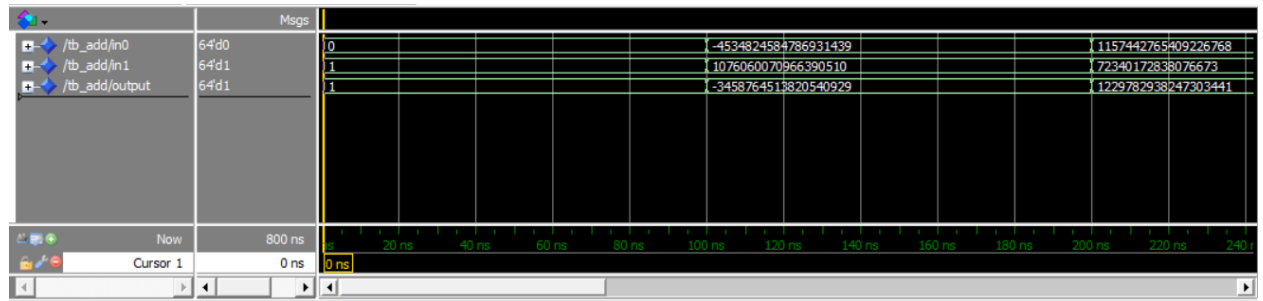


Figure 3

We can see from the simulation that, the 64 bit adder is right.

② ALU Control

4-bit ALU control input using a small control unit that has as inputs the opcode field of the instruction and a 2- bit control field, which called ALUOp. the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique.

We can find its truth table from our book.

| ALUOp | | Opcode field | | | | | | | | | | | Operation |
|--------|--------|--------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----------|
| ALUOp1 | ALUOp0 | I[31] | I[30] | I[29] | I[28] | I[27] | I[26] | I[25] | I[24] | I[23] | I[22] | I[21] | |
| 0 | 0 | X | X | X | X | X | X | X | X | X | X | X | 0010 |
| X | 1 | X | X | X | X | X | X | X | X | X | X | X | 0111 |
| 1 | X | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0010 |
| 1 | X | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0110 |
| 1 | X | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0000 |
| 1 | X | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0001 |

According to the truth table. ALUOp and opcode fields are inputs. We can control these two variable definition operations. We can limit uncertain inputs to determinable values. For example, when the operation is equal to "0111", we can use ALUOp(0) = "1" to determine, and when determining the opcode words, we can choose them as conditions to determine the operation output.

The VHDL code is that

```

7  -- To ensure proper functionality, you must implement the "don't-care" values in the funct field
8  -- for example when ALUOp = '00", Operation must be "0010" regardless of what Funct is.
9  port(
10     ALUOp      : in  STD_LOGIC_VECTOR(1 downto 0);
11     Opcode     : in  STD_LOGIC_VECTOR(10 downto 0);
12     Operation  : out STD_LOGIC_VECTOR(3 downto 0)
13  );
14  end ALUControl;
15
16  architecture behavl of ALUControl is
17  begin
18  col : process (ALUOp, Opcode)
19  begin
20  if ALUOp="00" then
21  Operation <= "0010";
22  elsif ALUOp(0)='1' then
23  Operation <= "0111";
24  elsif Opcode = "10001011000" then
25  Operation <= "0010";
26  elsif Opcode = "11001011000" then
27  Operation <= "0110";
28  elsif Opcode = "10001010000" then
29  Operation <= "0000";
30  elsif Opcode = "10101010000" then
31  Operation <= "0001";
32  else
33  Operation <="XXXX";
34  end if;
35  end process;
36  end behavl;

```

Then I built the testbench to test the function of the ALU Control. The testbench of mine is that (in the simulation, the signals in the simulation have the same names as the related ports.):

```

    ALUOp <="00";
    Opcode <= "11000000110";
wait for 50 ns;
    ALUOp <="00";
    Opcode <= "10000110110";
wait for 50 ns;
    ALUOp <="01";
    Opcode <= "11000000110";
wait for 50 ns;
    ALUOp <="11";
    Opcode <= "10000110110";
wait for 50 ns;
    ALUOp <="10";
    Opcode <= "10001011000";
wait for 50 ns;
    ALUOp <="10";
    Opcode <= "11001011000";
wait for 50 ns;
    ALUOp <="10";
    Opcode <= "10001010000";
wait for 50 ns;
    ALUOp <="10";
    Opcode <= "10101010000";
wait for 50 ns;
    ALUOp <="00";
    Opcode <= "10001010000";
wait for 50 ns;

```

Figure 4 is the simulation result:

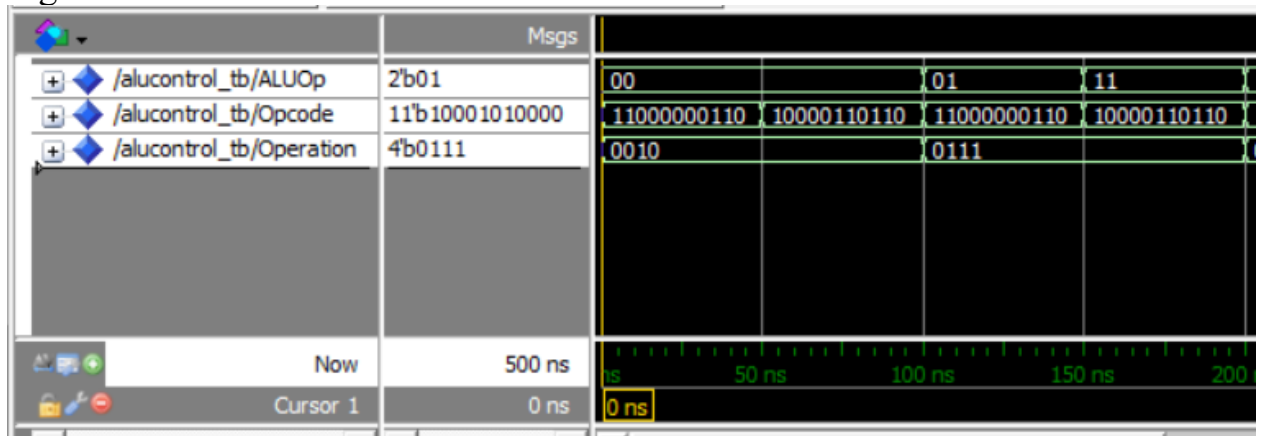


Figure 4(0ns-200ns)

We can see from the result that at $t = 150$ ns, The operation is not change, even though ALUOp(1) is changed to 1.

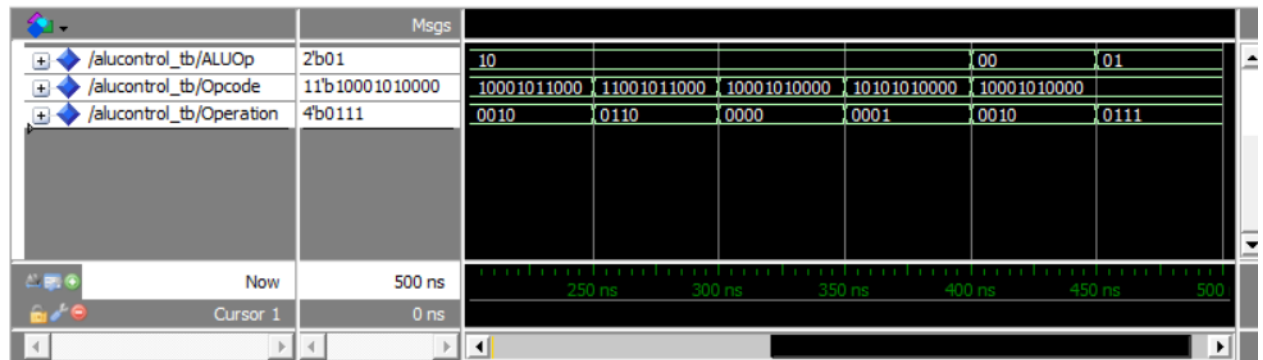


Figure 5(200ns-500ns)

The result is showing that the ALU control function is correct.

③ ALU

The LEGv8 ALU defines the six combinations of four control input and perform four functions in the lab2. The function of ALU are AND,OR,and SHUBTRACT. The ALU get two inputs form the register and get the operation code from the ALU control to do operations.

Below is the VHDL code of ALU:

```

2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  entity ALU is
5  -- Implement: AND, OR, ADD (signed), SUBTRACT (signed)
6  -- as described in Section 4.4 in the textbook.
7  -- The functionality of each instruction can be found on the 'ARM Reference Data' sheet at the
8  -- front of the textbook (or the Green Card pdf on Canvas).
9  port(
10     in0      : in    STD_LOGIC_VECTOR(63 downto 0);
11     in1      : in    STD_LOGIC_VECTOR(63 downto 0);
12     operation : in    STD_LOGIC_VECTOR(3 downto 0);
13     result   : buffer STD_LOGIC_VECTOR(63 downto 0);
14     zero     : buffer STD_LOGIC;
15     overflow : buffer STD_LOGIC
16  );
17  end ALU;
18
19  architecture behavl of ALU is
20
21
22     signal ADDresult : STD_LOGIC_VECTOR(63 downto 0);
23     component ADD is
24     port(
25         in0      : in    STD_LOGIC_VECTOR(63 downto 0);
26         in1      : in    STD_LOGIC_VECTOR(63 downto 0);
27         output   : out   STD_LOGIC_VECTOR(63 downto 0)
28     );
29     end component;
30

```



```

31 begin
32
33
34     ADDER:ADD port map(in0=>in0,in1=>in1,output=>ADDresult);
35     process(operation,in0,in1,result,ADDresult)
36     begin
37         case operation is
38             when "0000" =>
39                 result <= in0 and in1;
40                 overflow <='0';
41             when "0001" =>
42                 result <= in0 or in1;
43                 overflow <='0';
44             when "0010" =>
45
46                 result <= ADDresult;
47                 if in0(63) = '0' and in1(63) = '0' then
48                     if result(63) = '1' then
49                         overflow<='1';
50                     else overflow<= '0';
51                     end if;
52
53                 elsif in0(63) = '1' and in1(63) = '1' then
54                     if result(63) = '0' then
55                         overflow<='1';
56                     else overflow<='0';
57                     end if;
58                 end if;
59
60             when "0110" =>
61                 result <= in0-in1;
62
63                 if in0(63) = '0' and in1(63) = '1' then
64                     if result(63) = '1' then
65                         overflow<='1';
66                     else overflow<='0';
67                     end if;
68
69                 elsif in0(63) = '1' and in1(63) = '0' then
70                     if result(63) = '0' then
71                         overflow<='1';
72                     else overflow<='0';
73                     end if;
74                 end if;
75             when others =>
76                 null;
77         end case;
78         if result = "0000000000000000" then
79             zero <= '1';
80         else
81             zero <= '0';
82         end if;
83     end process;
84 end behavl;

```

Then I built the testbench to test the function of the ALU. The testbench of mine is that (in the simulation, the signals in the simulation have the same names as the related ports.):

```

44
45     operation <= "0000";
46     in0 <= x"1111_1111_1111_1111";
47     in1 <= x"0000_0000_0000_0001";
48     wait for 50 ns;
49
50     operation <= "0001";
51     in0 <= x"1111_1011_1111_1111";
52     in1 <= x"0000_0000_0000_0001";
53     wait for 50 ns;
54
55     operation <= "0010";
56     in0 <= x"1111_1111_0000_0000";
57     in1 <= x"0000_0000_0000_0001";
58     wait for 50 ns;
59
60     operation <= "0010";
61     in0 <= x"7FFF_FFFF_FFFF_FFFF";
62     in1 <= x"0000_0000_0000_0001";
63     wait for 50 ns;
64
65     operation <= "0110";
66     in0 <= x"F111_1011_1111_1111";
67     in1 <= x"0010_0000_0000_CCC1";
68     wait for 50 ns;
69
70     operation <= "0110";
71     in0 <= x"FFFF_FFFF_FFFF_FFFF";
72     in1 <= x"0000_0000_0000_0001";
73     wait for 50 ns;
74
75     operation <= "0110";
76     in0 <= x"100000000000000000";
77     in1 <= x"100000000000000000";
78     wait for 50 ns;
79
80     operation <= "0110";
81     in0 <= x"800000000000000000";
82     in1 <= x"000000000000000001";
83     wait for 50 ns;
84
85     end process;
86 end;

```

The figure 6 to figure is the simulation result

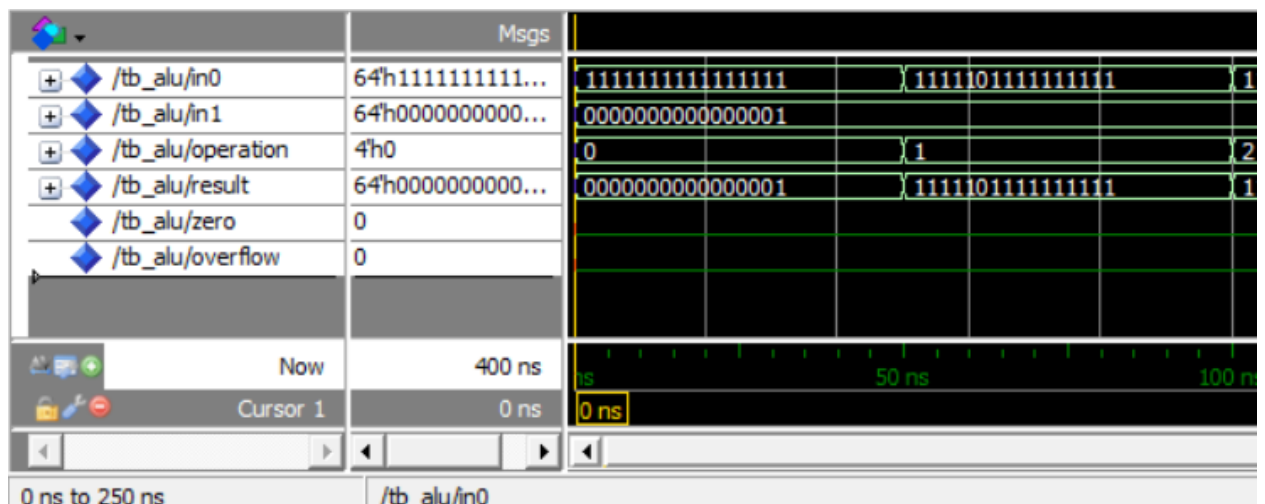


Figure 6(0ns-100ns)

We can see that in $t = 0$ ns to 50 ns, the operation is 0000, the function of ALU is AND. We can see

```
in0 = x"1111111111111111"
in1 = x"0000000000000001"
```

result = x0000000000000001, the result is right.

We can see that in t =50 ns to 100 ns, the operation is 0001, the function of ALU is OR. We can see

```
in0 = x"1111101111111111"
in1 = x"0000000000000001"
```

result = x1111101111111111, the result is right.

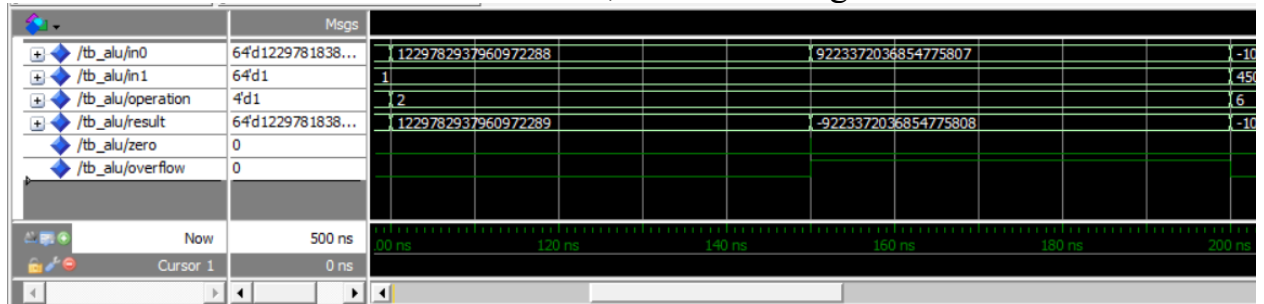


Figure 7(100ns-200ns)

We can see that in t =100 ns to 150 ns, the operation is 0010, the function of ALU is ADD. We can see

```
in0 <= x"1111_1111_0000_0000"
in1 <= x"0000_0000_0000_0001"
```

we can see from the figure that the result is right.

We can see that in t =150 ns to 200 ns, the operation is 0010, the function of ALU is ADD. We can see

```
in0 <= x"7FFF_FFFF_FFFF_FFFF"
in1 <= x"0000_0000_0000_0001"
```

So, the result is overflow. And we can see that overflow becomes high.

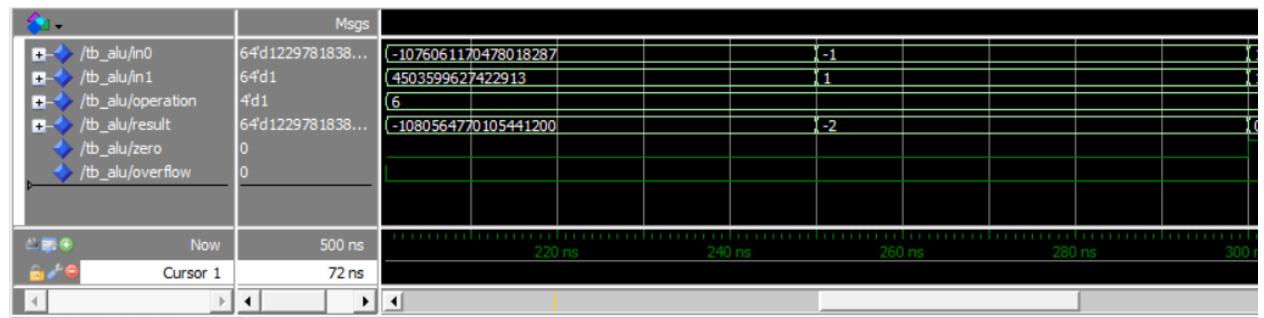


Figure 8(200ns-300ns)

We can see that in t =200 ns to 250 ns, the operation is 0110, the function of ALU is SUBSTRACT. We can see

```
in0 <= x"F111_1011_1111_1111"
```

in1 <= x"0010_0000_0000_CCC1" we can see from the figure that the

result is right.

We can see that in $t = 250$ ns to 300 ns, the operation is 0110, the function of ALU is SUBTRACT. We can see

$\text{in0} \leq \text{x} "7\text{FFF_FFFF_FFFF_FFFF}"$

$\text{in1} \leq \text{x} "0000_0000_0000_0001"$

we can see from the figure that the result is right.

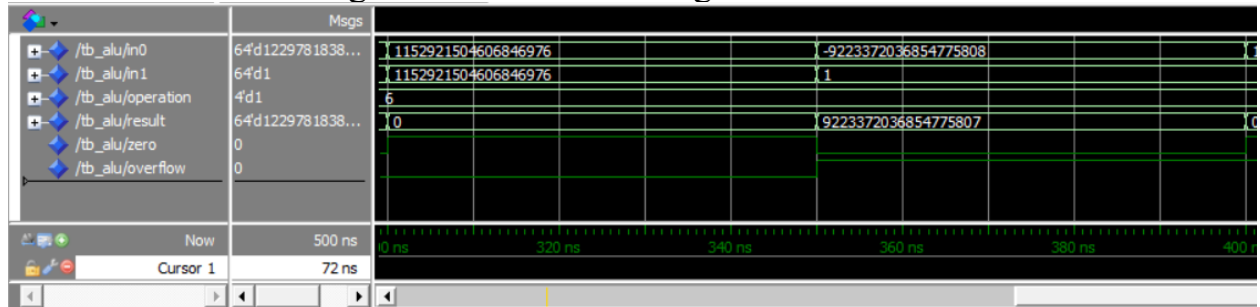


Figure 9(300ns-400ns)

We can see that in $t = 300$ ns to 350 ns, the operation is 0110, the function of ALU is SUBTRACT. We can see

$\text{in0} \leq \text{x} "1000000000000000"$.

$\text{in1} \leq \text{x} "1000000000000000"$.

The result is 0. So the output-zero becomes high.

We can see that in $t = 350$ ns to 400 ns, the operation is 0110, the function of ALU is SUBTRACT. We can see

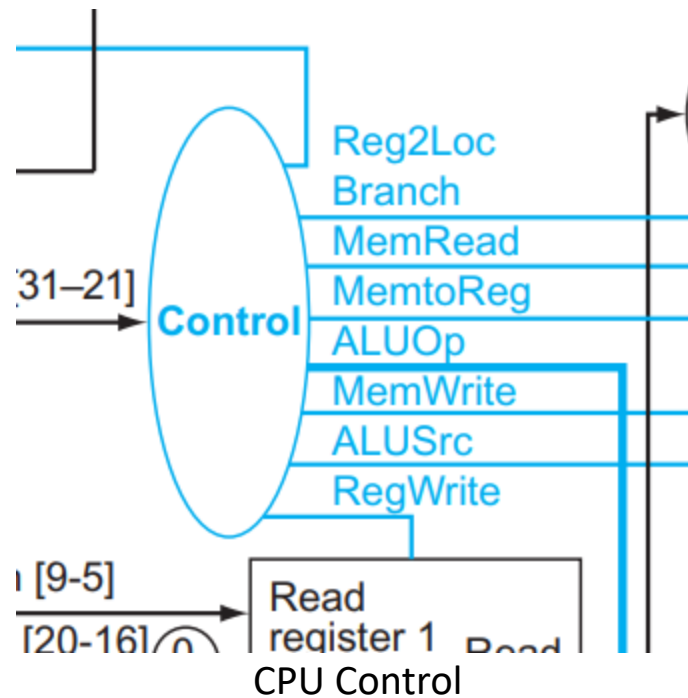
$\text{in0} \leq \text{x} "8000000000000000"$

$\text{in1} \leq \text{x} "0000000000000001"$.

So, the result is overflow. And we can see that overflow becomes high.

④ CPU Control

The input to the control unit is the 11-bit opcode field in the instruction. The output of the control unit consists of three 1-bit signals used to control the multiplexer. Three signals (RegWrite, MemRead, and MemWrite) to control reading and writing in register fields and data memory, a 1-bit signal (Branch) to determine if branching is possible, and a 2-bit control signal for the ALU (ALUOp) function should match the truth table.



The truth table is

| Input or output | Signal name | R-format | LDUR | STUR | CBZ |
|-----------------|-------------|----------|------|------|-----|
| Inputs | I[31] | 1 | 1 | 1 | 1 |
| | I[30] | X | 1 | 1 | 0 |
| | I[29] | X | 1 | 1 | 1 |
| | I[28] | 0 | 1 | 1 | 1 |
| | I[27] | 1 | 1 | 1 | 0 |
| | I[26] | 0 | 0 | 0 | 1 |
| | I[25] | 1 | 0 | 0 | 0 |
| | I[24] | X | 0 | 0 | 0 |
| | I[23] | 0 | 0 | 0 | X |
| | I[22] | 0 | 1 | 0 | X |
| | I[21] | 0 | 0 | 0 | X |
| Outputs | Reg2Loc | 0 | X | 1 | 1 |
| | ALUSrc | 0 | 1 | 1 | 0 |
| | MemtoReg | 0 | 1 | X | X |
| | RegWrite | 1 | 1 | 0 | 0 |
| | MemRead | 0 | 1 | 0 | 0 |
| | MemWrite | 0 | 0 | 1 | 0 |
| | Branch | 0 | 0 | 0 | 1 |
| | ALUOp1 | 1 | 0 | 0 | 0 |
| | ALUOp0 | 0 | 0 | 0 | 1 |

```

1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  entity CPUCtrl is
5  -- Functionality should match the truth table shown in Figure 4.22 of the textbook, including the
6  -- output 'X' values.
7  -- The truth table in Figure 4.22 omits the unconditional branch instruction:
8  -- UBranch = '1'
9  -- MemWrite = RegWrite = '0'
10 -- all other outputs = 'X'
11 port (Opcode : in STD_LOGIC_VECTOR(10 downto 0);
12       Reg2Loc : out STD_LOGIC;
13       CBranch : out STD_LOGIC; --conditional
14       MemRead : out STD_LOGIC;
15       MemtoReg : out STD_LOGIC;
16       MemWrite : out STD_LOGIC;
17       ALUSrc : out STD_LOGIC;
18       RegWrite : out STD_LOGIC;
19       UBranch : out STD_LOGIC; -- This is unconditional
20       ALUOp : out STD_LOGIC_VECTOR(1 downto 0)
21 );
22 end CPUCtrl;
23
24 ARCHITECTURE Behavl OF CPUCtrl IS
25 BEGIN
26     PROCESS (opcode)
27     BEGIN
28         regWrite <= '0';
29         CASE opcode IS
30             WHEN "11111000010" => --LDUR
31                 Reg2Loc <= 'X';
32                 ALUSrc <= '1';
33                 MemtoReg <= '1';
34                 RegWrite <= '1';
35                 MemRead <= '1';
36                 MemWrite <= '0';
37                 CBranch <= '0';
38                 UBranch <= '0';
39                 ALUOp <= "00" ;
40             WHEN "11111000000" => --STUR
41                 Reg2Loc <= '1';
42                 ALUSrc <= '1';
43                 MemtoReg <= 'X';
44                 RegWrite <= '0';
45                 MemRead <= '0';
46                 MemWrite <= '1';
47                 CBranch <= '0';
48                 UBranch <= '0';
49                 ALUOp <= "00" ;
50             WHEN "10110100000" => --CBZ
51                 Reg2Loc <= '1';
52                 ALUSrc <= '0';
53                 MemtoReg <= 'X';
54                 RegWrite <= '0';
55                 MemRead <= '0';
56                 MemWrite <= '0';
57                 CBranch <= '1';
58                 UBranch <= '0';
59                 ALUOp <= "01" ;
60             WHEN "10110100001" => --CBZ
61                 Reg2Loc <= '1';
62                 ALUSrc <= '0';
63                 MemtoReg <= 'X';
64                 RegWrite <= '0';
65                 MemRead <= '0';
66                 MemWrite <= '0';
67                 CBranch <= '1';
68                 UBranch <= '0';
69                 ALUOp <= "01" ;
70             WHEN "10110100010" => --CBZ
71                 Reg2Loc <= '1';
72                 ALUSrc <= '0';
73                 MemtoReg <= 'X';
74                 RegWrite <= '0';
75                 MemRead <= '0';
76                 MemWrite <= '0';
77                 CBranch <= '1';
78                 UBranch <= '0';
79                 ALUOp <= "01" ;
80             WHEN "10110100011" => --CBZ
81                 Reg2Loc <= '1';
82                 ALUSrc <= '0';
83                 MemtoReg <= 'X';
84                 RegWrite <= '0';
85                 MemRead <= '0';
86                 MemWrite <= '0';
87                 CBranch <= '1';
88                 UBranch <= '0';
89                 ALUOp <= "01" ;
90             WHEN "10110100010" => --CBZ
91                 Reg2Loc <= '1';
92                 ALUSrc <= '0';
93                 MemtoReg <= 'X';
94                 RegWrite <= '0';
95                 MemRead <= '0';
96                 MemWrite <= '0';
97                 CBranch <= '1';
98                 UBranch <= '0';
99                 ALUOp <= "01" ;
100            WHEN "10110100011" => --CBZ
101                Reg2Loc <= '1';
102                ALUSrc <= '0';
103                MemtoReg <= 'X';
104                RegWrite <= '0';
105                MemRead <= '0';
106                MemWrite <= '0';
107                CBranch <= '1';
108                UBranch <= '0';
109                ALUOp <= "01" ;
110            WHEN "10110100010" => --CBZ
111                Reg2Loc <= '1';
112                ALUSrc <= '0';
113                MemtoReg <= 'X';
114                RegWrite <= '0';
115                MemRead <= '0';
116                MemWrite <= '0';
117                CBranch <= '1';
118                UBranch <= '0';
119                ALUOp <= "01" ;
120            WHEN "10110100011" => --CBZ
121                Reg2Loc <= '1';
122                ALUSrc <= '0';
123                MemtoReg <= 'X';
124                RegWrite <= '0';
125                MemRead <= '0';
126                MemWrite <= '0';
127                CBranch <= '1';
128                UBranch <= '0';
129                ALUOp <= "01" ;
130            WHEN "10110100010" => --CBZ
131                Reg2Loc <= '1';
132                ALUSrc <= '0';
133                MemtoReg <= 'X';
134                RegWrite <= '0';
135                MemRead <= '0';
136                MemWrite <= '0';
137                CBranch <= '1';
138                UBranch <= '0';
139                ALUOp <= "01" ;
140            WHEN "10110100011" => --CBZ
141                Reg2Loc <= '1';
142                ALUSrc <= '0';
143                MemtoReg <= 'X';
144                RegWrite <= '0';
145                MemRead <= '0';
146                MemWrite <= '0';
147                CBranch <= '1';
148                UBranch <= '0';
149                ALUOp <= "01" ;
150            WHEN "10110100010" => --CBZ
151                Reg2Loc <= '1';
152                ALUSrc <= '0';
153                MemtoReg <= 'X';
154                RegWrite <= '0';
155                MemRead <= '0';
156                MemWrite <= '0';
157                CBranch <= '1';
158                UBranch <= '0';
159                ALUOp <= "01" ;
160            WHEN "10110100011" => --CBZ
161                Reg2Loc <= '1';
162                ALUSrc <= '0';
163                MemtoReg <= 'X';
164                RegWrite <= '0';
165                MemRead <= '0';
166                MemWrite <= '0';
167                CBranch <= '1';
168                UBranch <= '0';
169                ALUOp <= "01" ;
170            WHEN "10110100010" => --CBZ
171                Reg2Loc <= '1';
172                ALUSrc <= '0';
173                MemtoReg <= 'X';
174                RegWrite <= '0';
175                MemRead <= '0';
176                MemWrite <= '0';
177                CBranch <= '1';
178                UBranch <= '0';
179                ALUOp <= "01" ;
180            WHEN "10110100011" => --CBZ
181                Reg2Loc <= '1';
182                ALUSrc <= '0';
183                MemtoReg <= 'X';
184                RegWrite <= '0';
185                MemRead <= '0';
186                MemWrite <= '0';
187                CBranch <= '1';
188                UBranch <= '0';
189                ALUOp <= "01" ;
190            WHEN "10110100010" => --CBZ
191                Reg2Loc <= '1';
192                ALUSrc <= '0';
193                MemtoReg <= 'X';
194                RegWrite <= '0';
195                MemRead <= '0';
196                MemWrite <= '0';
197                CBranch <= '1';
198                UBranch <= '0';
199                ALUOp <= "01" ;
200            WHEN "10110100011" => --CBZ
201                Reg2Loc <= '1';
202                ALUSrc <= '0';
203                MemtoReg <= 'X';
204                RegWrite <= '0';
205                MemRead <= '0';
206                MemWrite <= '0';
207                CBranch <= '1';
208                UBranch <= '0';
209                ALUOp <= "01" ;
210            WHEN "10110100010" => --CBZ
211                Reg2Loc <= '1';
212                ALUSrc <= '0';
213                MemtoReg <= 'X';
214                RegWrite <= '0';
215                MemRead <= '0';
216                MemWrite <= '0';
217                CBranch <= '1';
218                UBranch <= '0';
219                ALUOp <= "01" ;
220            WHEN "10110100011" => --CBZ
221                Reg2Loc <= '1';
222                ALUSrc <= '0';
223                MemtoReg <= 'X';
224                RegWrite <= '0';
225                MemRead <= '0';
226                MemWrite <= '0';
227                CBranch <= '1';
228                UBranch <= '0';
229                ALUOp <= "01" ;
230            WHEN "10110100010" => --CBZ
231                Reg2Loc <= '1';
232                ALUSrc <= '0';
233                MemtoReg <= 'X';
234                RegWrite <= '0';
235                MemRead <= '0';
236                MemWrite <= '0';
237                CBranch <= '1';
238                UBranch <= '0';
239                ALUOp <= "01" ;
240            WHEN "10110100011" => --CBZ
241                Reg2Loc <= '1';
242                ALUSrc <= '0';
243                MemtoReg <= 'X';
244                RegWrite <= '0';
245                MemRead <= '0';
246                MemWrite <= '0';
247                CBranch <= '1';
248                UBranch <= '0';
249                ALUOp <= "01" ;
250            WHEN "10110100010" => --CBZ
251                Reg2Loc <= '1';
252                ALUSrc <= '0';
253                MemtoReg <= 'X';
254                RegWrite <= '0';
255                MemRead <= '0';
256                MemWrite <= '0';
257                CBranch <= '1';
258                UBranch <= '0';
259                ALUOp <= "01" ;
260            WHEN "10110100011" => --CBZ
261                Reg2Loc <= '1';
262                ALUSrc <= '0';
263                MemtoReg <= 'X';
264                RegWrite <= '0';
265                MemRead <= '0';
266                MemWrite <= '0';
267                CBranch <= '1';
268                UBranch <= '0';
269                ALUOp <= "01" ;
270            WHEN "10110100010" => --CBZ
271                Reg2Loc <= '1';
272                ALUSrc <=
```

```

83
84
85     WHEN "10110100011" => --CBZ
86         Reg2Loc      <= '1';
87         ALUsrc       <= '0';
88         MemtoReg     <= 'X';
89         RegWrite     <= '0';
90         MemRead      <= '0';
91         MemWrite     <= '0';
92         CBranch      <= '1';
93         UBranch      <= '0';
94         ALUop        <= "01" ;
95
96     WHEN "10110100100" => --CBZ
97         Reg2Loc      <= '1';
98         ALUsrc       <= '0';
99         MemtoReg     <= 'X';
100        RegWrite     <= '0';
101        MemRead      <= '0';
102        MemWrite     <= '0';
103        CBranch      <= '1';
104        UBranch      <= '0';
105        ALUop        <= "01" ;
106
107     WHEN "10110100101" => --CBZ
108         Reg2Loc      <= '1';
109         ALUsrc       <= '0';
110         MemtoReg     <= 'X';
111         RegWrite     <= '0';
112         MemRead      <= '0';
113         MemWrite     <= '0';
114         CBranch      <= '1';
115         UBranch      <= '0';
116         ALUop        <= "01" ;
117
118     WHEN "10110100110" => --CBZ
119         Reg2Loc      <= '1';
120         ALUsrc       <= '0';
121         MemtoReg     <= 'X';
122         RegWrite     <= '0';
123         MemRead      <= '0';
124         MemWrite     <= '0';
125         CBranch      <= '1';
126         UBranch      <= '0';
127         ALUop        <= "01" ;
128
129     WHEN "10110100111" => --CBZ
130         Reg2Loc      <= '1';
131         ALUsrc       <= '0';
132         MemtoReg     <= 'X';
133         RegWrite     <= '0';
134         MemRead      <= '0';
135         MemWrite     <= '0';
136         CBranch      <= '1';
137         UBranch      <= '0';
138         ALUop        <= "01" ;
139
140     WHEN "10001010000" => --R-format
141         Reg2Loc      <= '0';
142         ALUsrc       <= '0';
143         MemtoReg     <= '0';
144         regWrite     <= '1';
145         MemRead      <= '0';
146         MemWrite     <= '0';
147         CBranch      <= '0';
148         UBranch      <= '0';
149         ALUop        <= "10" ;
150
151     WHEN "10001011000" => --R-format
152         Reg2Loc      <= '0';
153         ALUsrc       <= '0';
154         MemtoReg     <= '0';
155         RegWrite     <= '1';
156         MemRead      <= '0';
157         MemWrite     <= '0';
158         CBranch      <= '0';
159         UBranch      <= '0';
160         ALUop        <= "10" ;
161
162     WHEN "10101010000" => --R-format
163         Reg2Loc      <= '0';
164         ALUsrc       <= '0';
165         MemtoReg     <= '0';
166         RegWrite     <= '1';
167         MemRead      <= '0';
168         MemWrite     <= '0';
169         CBranch      <= '0';
170         UBranch      <= '0';
171         ALUop        <= "10" ;

```

```

166         WHEN "10101011000" => --R-format
167             Reg2Loc      <= '0';
168             ALUsrc       <= '0';
169             MemtoReg     <= '0';
170             RegWrite     <= '1';
171             MemRead      <= '0';
172             MemWrite     <= '0';
173             CBranch      <= '0';
174             UBranch      <= '0';
175             ALUop        <= "10" ;
176         WHEN "11001010000" => --R-format
177             Reg2Loc      <= '0';
178             ALUsrc       <= '0';
179             MemtoReg     <= '0';
180             RegWrite     <= '1';
181             MemRead      <= '0';
182             MemWrite     <= '0';
183             CBranch      <= '0';
184             UBranch      <= '0';
185             ALUop        <= "10" ;
186         WHEN "11001011000" => --R-format
187             Reg2Loc      <= '0';
188             ALUsrc       <= '0';
189             MemtoReg     <= '0';
190             RegWrite     <= '1';
191             MemRead      <= '0';
192             MemWrite     <= '0';
193             CBranch      <= '0';
194             UBranch      <= '0';
195             ALUop        <= "10" ;
196         WHEN "11101010000" => --R-format
197             Reg2Loc      <= '0';
198             ALUsrc       <= '0';
199             MemtoReg     <= '0';
200             RegWrite     <= '1';
201             MemRead      <= '0';
202             MemWrite     <= '0';
203             CBranch      <= '0';
204             UBranch      <= '0';
205             ALUop        <= "10" ;
206         WHEN "11101011000" => --R-format
207             Reg2Loc      <= '0';
208             ALUsrc       <= '0';
209             MemtoReg     <= '0';
210             RegWrite     <= '1';
211             MemRead      <= '0';
212             MemWrite     <= '0';
213             CBranch      <= '0';
214             UBranch      <= '0';
215             ALUop        <= "10" ;
216         WHEN "00000000000" => --R-format
217             Reg2Loc      <= '0';
218             ALUsrc       <= '0';
219             MemtoReg     <= '0';
220             RegWrite     <= '1';
221             MemRead      <= '0';
222             MemWrite     <= '0';
223             CBranch      <= '0';
224             UBranch      <= '0';
225             ALUop        <= "10" ;
226         WHEN "11010001000" =>
227             Reg2Loc      <= '0';
228             CBranch      <= '1';
229             UBranch      <= '0';
230             MemRead      <= '0';
231             MemtoReg     <= '0';
232             ALUop        <= "10";
233             MemWrite     <= '0';
234             ALUsrc       <= '1';
235             RegWrite     <= '0' ;
236         WHEN "00010100000" =>
237             Reg2Loc      <= '0';
238             CBranch      <= '1';
239             UBranch      <= '0';
240             MemRead      <= '0';
241             MemtoReg     <= '0';
242             ALUop        <= "00";
243             MemWrite     <= '0';
244             ALUsrc       <= '1';
245             RegWrite     <= '0' ;

```



```

246         WHEN OTHERS => NULL;
247         Reg2Loc    <= '0';
248         CBranch    <= '0';
249         UBranch    <= '0';
250         MemRead     <= '0';
251         MemtoReg    <= '0';
252         ALUOp       <= "00";
253         MemWrite    <= '0';
254         ALUSrc      <= '0';
255         RegWrite    <= '0';
256
257     END CASE;
258 END PROCESS;
259 END Behav1;
260

```

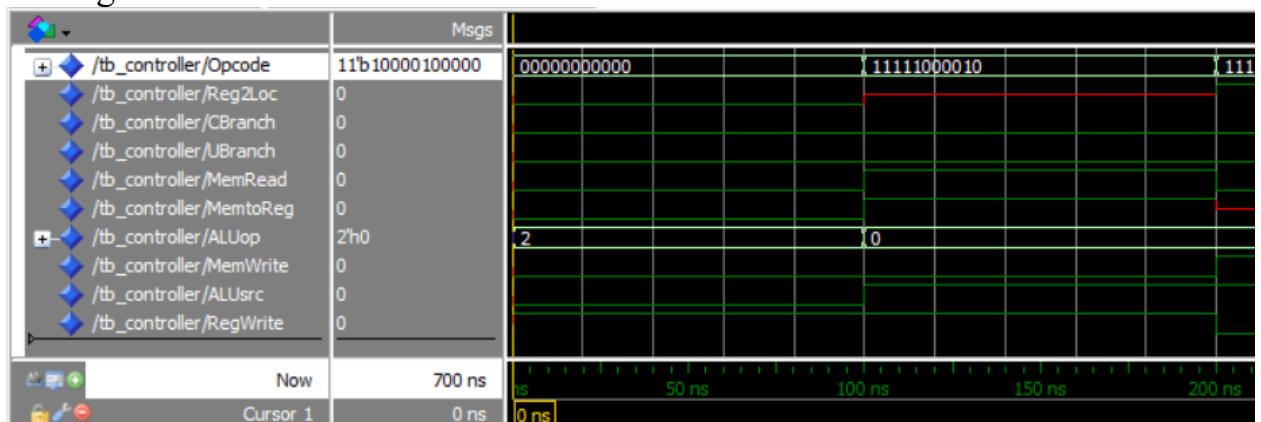
Then I built the testbench to test the function of the CPU. The testbench of mine is that (in the simulation, the signals in the simulation have the same names as the related ports.):

```

46 stim_proc : PROCESS
47 BEGIN
48
49     Opcode <= "000000000000";
50     WAIT FOR 100 ns;
51
52     Opcode <= "11111000010";
53     WAIT FOR 100 ns;
54     Opcode <= "11111000000";
55     WAIT FOR 100 ns;
56
57     Opcode <= "10110100111";
58     WAIT FOR 100 ns;
59     Opcode <= "10110100000";
60
61
62     WAIT FOR 100 ns;
63     Opcode <= "11000101000";
64     WAIT FOR 100 ns;
65     Opcode <= "11111111111";
66     WAIT FOR 100 ns;
67     Opcode <= "10000100000";
68
69 wait:
70 END PROCESS;
71 END;

```

The figure below is the simulation result



0 ns to 200 ns

During time $t = 0$ ns to $t = 100$ ns, I set

Opcode <= "000000000000"

The result is:

```

Reg2Loc   = '0'
ALUsrc    = '0'
MemtoReg  = '0'
RegWrite  = '1'
MemRead   = '0'
MemWrite  = '0'
CBranch   = '0'
UBranch   = '0'
ALUOp     = "10"

```

The result is correct.

During time $t=100$ ns to $t=200$ ns, I set

Opcode \leq "11111000010"

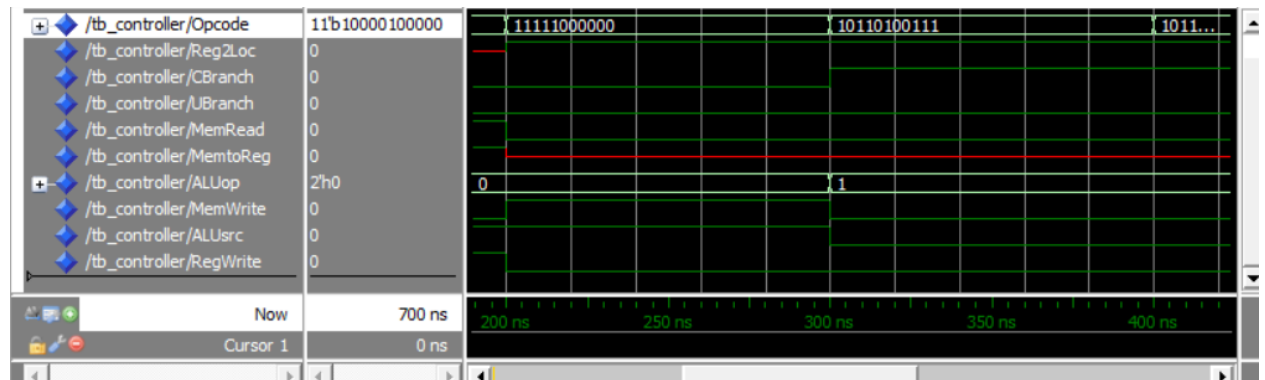
The result is:

```

Reg2Loc   = 'X'
ALUsrc    = '1'
MemtoReg  = '1'
RegWrite  = '1'
MemRead   = '1'
MemWrite  = '0'
CBranch   = '0'
UBranch   = '0'
ALUOp     = "00"

```

The result is correct.



200 ns to 400 ns

During time $t=200$ ns to $t=300$ ns, I set

Opcode \leq "11111000000"

The result is:

```

Reg2Loc    = '1'
ALUsrc     = '1'
MemtoReg   = 'X'
RegWrite   = '0'
MemRead    = '0'
MemWrite   = '1'
CBranch    = '0'
UBranch    = '0'
ALUOp      = "00"

```

The result is correct.

During time $t = 300 \text{ ns}$ to $t = 400 \text{ ns}$, I set

```
Opcode <= "10110100111"
```

The result is:

```

Reg2Loc = '1'
ALUsrc   = '0'
MemtoReg = 'X'
RegWrite = '0'
MemRead  = '0'
MemWrite = '0'
CBranch  = '1'
UBranch  = '0'
ALUOp    = "01"

```

The result is correct.

⑤ Data Memory testbench

The par is to write the testbench of data memory. The function is to read from or write to the data memory provided. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory.

Below is the testbench I built (in the simulation, the signals in the simulation have the same names as the related ports.): The cycle of the clock is set to 50 ns.

```

);
sim_proc: process
begin

    MemWrite <= '0';
    MemRead <= '1';
    Address <= x"0000000000000000";
    WriteData <= x"1111222200000000";

    WAIT FOR 50 ns;

    MemWrite <= '1';
    MemRead <= '0';
    WriteData <= x"1111222200000001";
    Address <= x"000000000000000A";
    WAIT FOR 50 ns;

    MemWrite <= '0';
    MemRead <= '1';
    Address <= x"0000000000000050";
    WriteData <= x"1111222200000000";

    wait;
END PROCESS;
Clock <= not Clock after 25 ns;
END.

```

Figure 10 to below shows the result of the simulation:

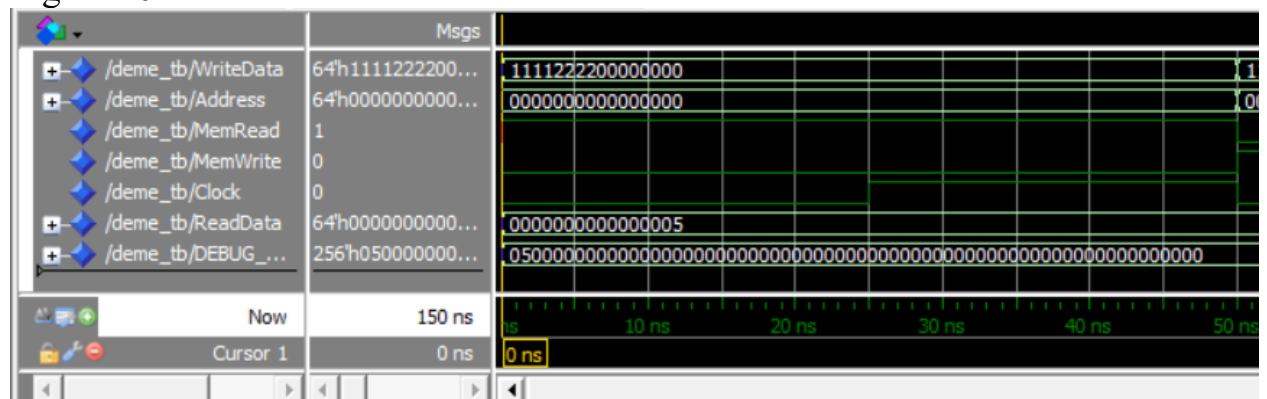


Figure 10 (0ns-50ns)

During $t = 0$ ns to $t = 50$ ns, I try to read the value stored in the `dmemBytes(0)`, So, I set :

```

MemWrite <= '0';
MemRead <= '1'

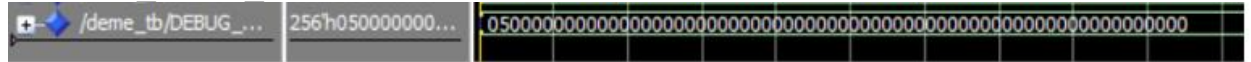
```

Address <= x"0000000000000000"

WriteData <= x"1111222200000000"

And the result:

DEBUG MEM CONTENTS is



And in the initialize of the dmemBytes, I set dmemBytes(0) to "00000101"

```
if(first) then
-- Example: MEM(0x0) = 0x0000000000000001 (Hex)
-- 1(decimal)
dmemBytes(7) <= "00000000";
dmemBytes(6) <= "00000000";
dmemBytes(5) <= "00000000";
dmemBytes(4) <= "00000000";
dmemBytes(3) <= "00000000";
dmemBytes(2) <= "00000000";
dmemBytes(1) <= "00000000";
dmemBytes(0) <= "00000101"; --least significant has the lowest address

dmemBytes(15) <= "00000000";
```

And in the dmem.vhd, we can learn that

```
-- Connect the signals that will be used for testing. Complete it to (64*4 1 downto 0)
DEBUG_MEM_CONTENTS <=
dmemBytes( 0) & dmemBytes( 1) & dmemBytes( 2) & dmemBytes( 3) & --DMEM(0)
dmemBytes( 4) & dmemBytes( 5) & dmemBytes( 6) & dmemBytes( 7) & --DMEM(4)
dmemBytes( 8) & dmemBytes( 9) & dmemBytes(10) & dmemBytes(11) & --DMEM(8)
dmemBytes(12) & dmemBytes(13) & dmemBytes(14) & dmemBytes(15) & --DMEM(12)
dmemBytes(16) & dmemBytes(17) & dmemBytes(18) & dmemBytes(19) & --DMEM(16)
dmemBytes(20) & dmemBytes(21) & dmemBytes(22) & dmemBytes(23) & --DMEM(20)
dmemBytes(24) & dmemBytes(25) & dmemBytes(26) & dmemBytes(27) & --DMEM(24)
dmemBytes(28) & dmemBytes(29) & dmemBytes(30) & dmemBytes(31); --DMEM(28)
```

dmemBytes(0) is the most significant byte of the DEBUG_MEM_CONTENTS. So, the result- DEBUG_MEM_CONTENTS gotten from the simulation is right.

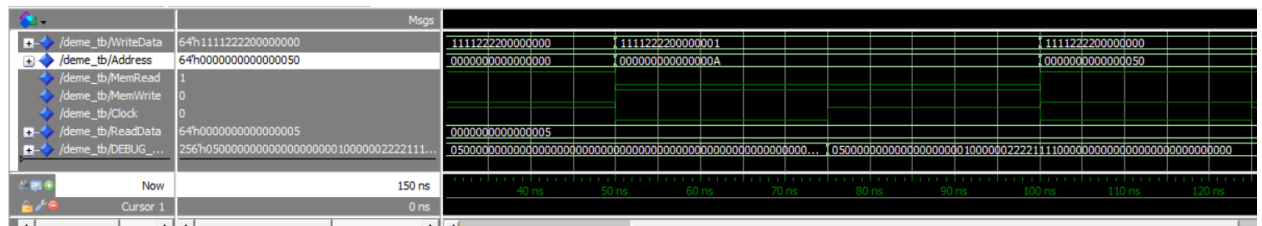


Figure 11 (50ns-100ns)

During t = 50 ns to t = 100 ns, I try to store the value stored data in the dmemBytes(10), So, I set:

```
MemWrite <= '1'
MemRead <= '0'
```

```
WriteData <= x"11112222000000001"
Address <= x"0000000000000000A"
```

And we can see the simulation result- DEBUG MEM CONTENTS

```
-- Connect the signals that will be used for testing. Complete it to (64*4 1 downto 0)
DEBUG_MEM_CONTENTS <=
  dmemBytes( 0) & dmemBytes( 1) & dmemBytes( 2) & dmemBytes( 3) & --DMEM(0)
  dmemBytes( 4) & dmemBytes( 5) & dmemBytes( 6) & dmemBytes( 7) & --DMEM(4)
  dmemBytes( 8) & dmemBytes( 9) & dmemBytes(10) & dmemBytes(11) & --DMEM(8)
  dmemBytes(12) & dmemBytes(13) & dmemBytes(14) & dmemBytes(15) & --DMEM(12)
  dmemBytes(16) & dmemBytes(17) & dmemBytes(18) & dmemBytes(19) & --DMEM(16)
  dmemBytes(20) & dmemBytes(21) & dmemBytes(22) & dmemBytes(23) & --DMEM(20)
  dmemBytes(24) & dmemBytes(25) & dmemBytes(26) & dmemBytes(27) & --DMEM(24)
  dmemBytes(28) & dmemBytes(29) & dmemBytes(30) & dmemBytes(31) & --DMEM(28)
end behavioral;
```

```
/deme_tb/uut/dmemBytes
0: 05 00 00 00 00 00 00 00
8: 00 00 01 00 00 00 22 22
16: 11 11 00 00 00 00 00 00
24: 00 00 00 00 00 00 00 00
32: 00 00 00 00 00 00 00 00
40: 00 01 00 00 00 00 00 00
48: 00 00 00 00 00 00 00 00
56: 00 00 00 00 00 00 00 00
64: XX
```

The data have been written in the dmemBytes from dmemBytes(10).

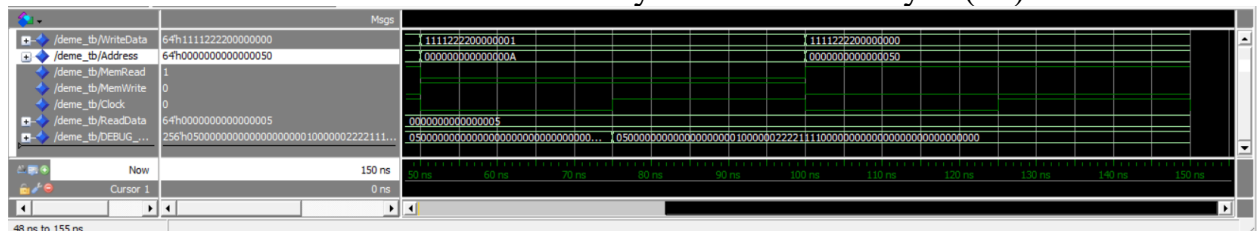


Figure 12 (100ns-150ns)

During t = 100 ns to t = 150 ns, I try print the error message, So, I set:

```
MemWrite <= '0'
MemRead <= '1'
Address <= x"00000000000000050"
WriteData <= x"11112222000000000"
```

And result is:

```
# ** Error: Invalid DMEM addr. Attempted to read 4-bytes starting at address 80 but only 64 bytes are available
# Time: 100 ns Iteration: 1 Instance: /deme_tb/uut
# ** Error: Invalid DMEM addr. Attempted to read 4-bytes starting at address 80 but only 64 bytes are available
# Time: 125 ns Iteration: 0 Instance: /deme_tb/uut
# ** Error: Invalid DMEM addr. Attempted to read 4-bytes starting at address 80 but only 64 bytes are available
# Time: 150 ns Iteration: 0 Instance: /deme_tb/uut
add wave position breakpoint sim:/deme_tb/
```

⑥ Instruction Memory

In this part, we need to build our Instruction Memory which is a byte addressable, little-endian, read-only memory. And the Instruction Memory should be 128 bytes. The Instruction Memory can only be read. So, it only has one input

which shows the memory to be read and one output that output the data gotten from memory.

The VHDL code is that:

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.NUMERIC_STD.ALL;
4  entity IMEM is
5  -- The instruction memory is a byte addressable, little-endian, read-only memory
6  -- Reads occur continuously
7  -- HINT: Use the provided dmem.vhd as a starting point
8  generic(NUM_BYTES : integer := 128);
9  -- NUM_BYTES is the number of bytes in the memory (small to save computation resources)
10 port(
11     Address : in  STD_LOGIC_VECTOR(63 downto 0); -- Address to read from
12     ReadData : out STD_LOGIC_VECTOR(31 downto 0)
13 );
14 end IMEM;
15
16 architecture behavl of IMEM is
17     type ByteArray is array (0 to NUM_BYTES) of STD_LOGIC_VECTOR(7 downto 0);
18     signal ImemBytes:ByteArray;
19 begin
20     process(Address)
21         variable addr:integer;
22         variable first:boolean := true;
23     begin
24         if(first) then
25
26             ImemBytes(7)  <= "00000000";
27             ImemBytes(6)  <= "00000000";
28             ImemBytes(5)  <= "00000000";
29             ImemBytes(4)  <= "11111111";
30             ImemBytes(3)  <= "00000000";
31             ImemBytes(2)  <= "00000000";
32             ImemBytes(1)  <= "00000000";
33             ImemBytes(0)  <= "00000001"; --least significant has the lowest address
34
35             ImemBytes(15) <= "00000000";
36             ImemBytes(14) <= "00000000";
37             ImemBytes(13) <= "00000000";
38             ImemBytes(12) <= "00000000";
```

The initialization of intermediate memory is not displayed.

```
164     ImemBytes(117) <= "00000000";
165     ImemBytes(116) <= "00000000";
166     ImemBytes(115) <= "00000000";
167     ImemBytes(114) <= "00000000";
168     ImemBytes(113) <= "00000000";
169     first := false;
170     end if;
171
172     addr:=to_integer(unsigned(Address));
173     if (addr+7 < NUM_BYTES) then
174         ReadData <= ImemBytes(addr+3) & ImemBytes(addr+2) &
175             ImemBytes(addr+1) & ImemBytes(addr+0);
176     else report "Invalid DMEM addr. Attempted to read 4-bytes starting at address " &
177         integer'image(addr) & " but only " & integer'image(NUM_BYTES) & " bytes are available"
178         severity error;
179     end if;
180
181 end process;
182
183 end behavl;
184
185
```

Then I built the testbench to test the Instruction Memory. The testbench of mine is that (in the simulation, the signals in the simulation have the same names as the related ports.):


```
stimu_proc: process
begin
    Address <= x"0000_0000_0000_0004";
    wait for 50 ns;
    Address <= x"0000_0000_0000_0020";
    wait for 50 ns;
    Address <= x"0000_0000_0000_0008";
    wait for 50 ns;
    Address <= x"0000_0000_1000_0040";
    wait for 50 ns;
    wait;
end process;
end;
```

Figure 13 is the simulation result:

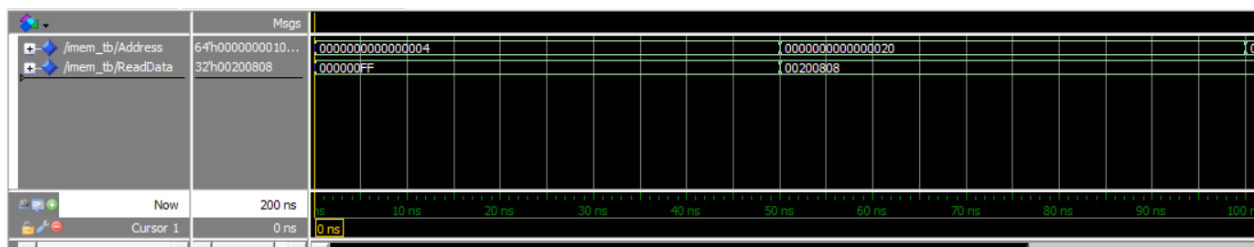


Figure 13 (0ns-100ns)

During $t = 0$ ns to $t = 50$ ns, I try to read the data stored data in the `ImemBytes(4)`, So, I set: `Address <= x"0000_0000_0000_0004"`;
And the `ReadData = x000000FF`. According to the memory list:


```

ImemBytes(36) <= "00000000";
ImemBytes(35) <= "00000000";
ImemBytes(34) <= "00100000";
ImemBytes(33) <= "00001000";
ImemBytes(32) <= "00001000";

```

During $t = 150 \text{ ns}$ to $t = 200 \text{ ns}$, I try print the error message, So, I set:

Address $\leq x"0000_0000_1000_0040"$;

The result is

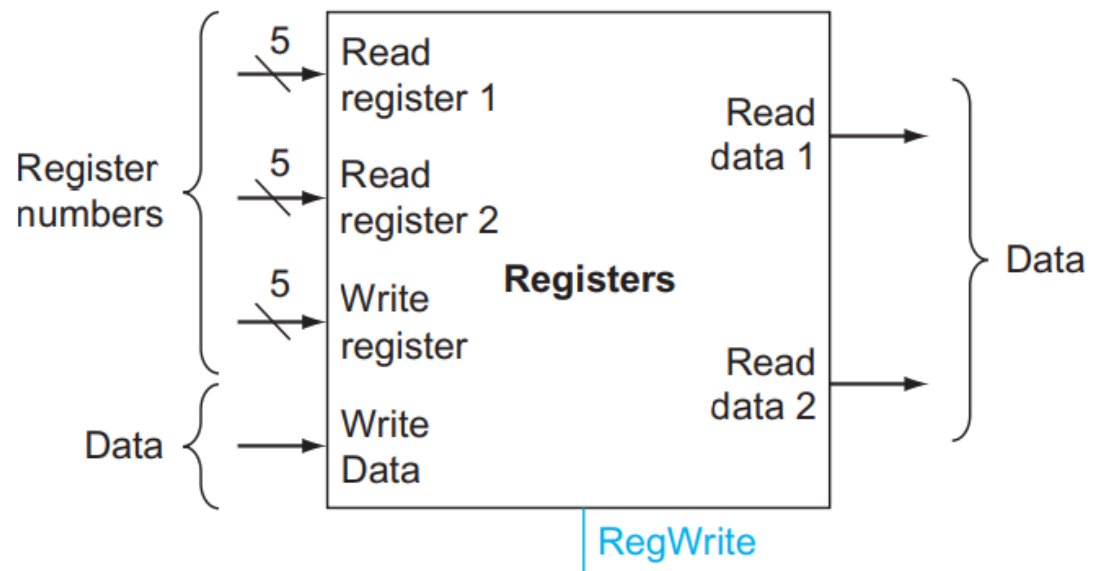
```

sim 2> run
** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
Time: 0 ns Iteration: 0 Instance: /imem_tb/uut
** Error: Invalid DMEM addr. Attempted to read 4-bytes starting at address 268435520 but only 128 bytes are available
Time: 150 ns Iteration: 1 Instance: /imem_tb/uut

```

⑦ Registers

In this part of lab, we need to build 32 registers. And each of these registers has 64-bit. We need to read data from the register or load data to the register.



According to the figure in the book. When $\text{RegWrite} = 1$, we want to load WD to the register according to WR. WR, RR1 and RR2 show we that which register would be written or be loaded data. And RD1 and RD2 as the

output to output the data we read from Number.RD1 register and Number.RD1 register.

So, the VHDL code is that:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use IEEE.numeric_std.ALL;
4  entity registers is
5  -- This component is described in the textbook, starting on section 4.3
6  -- The indices of each of the registers can be found on the LEGv8 Green Card
7  -- Keep in mind that register 31 (XZR) has a constant value of 0 and cannot be overwritten
8  -- This should only write on the negative edge of Clock when RegWrite is asserted.
9  -- Reads should be purely combinatorial, i.e. they don't depend on Clock
10 -- HINT: Use the provided dmem.vhd as a starting point
11 generic(TotalByte : integer := 256);
12
13 port(RR1      : in  STD_LOGIC_VECTOR (4 downto 0);
14      RR2      : in  STD_LOGIC_VECTOR (4 downto 0);
15      WR       : in  STD_LOGIC_VECTOR (4 downto 0);
16      WD       : in  STD_LOGIC_VECTOR (63 downto 0);
17      RegWrite  : in  STD_LOGIC;
18      Clock    : in  STD_LOGIC;
19      RD1      : out STD_LOGIC_VECTOR (63 downto 0);
20      RD2      : out STD_LOGIC_VECTOR (63 downto 0);
21
22      --Probe ports used for testing.
23      -- Notice the width of the port means that you are
24      -- reading only part of the register file.
25      -- This is only for debugging
26      -- You are debugging a subset of registers here
27      -- Temp registers: $X9 & $X10 & X11 & X12
28      -- 4 refers to number of registers you are debugging
29      DEBUG_TMP_REGS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0);
30      -- Saved Registers X19 & $X20 & X21 & X22
31      DEBUG_SAVED_REGS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0)
32 );
33 end registers;
34
35 architecture behav1 of registers is
36     type ByteArray is array (0 to TotalByte) of STD_LOGIC_VECTOR(7 downto 0);
37     signal dmemBytes : ByteArray;
38
39     begin
40         process(Clock,RegWrite,WD,WR,RR2,RR1)
41             variable addr:integer;
42             variable addr1 : integer;
43             variable addr2 : integer;
44             variable first:boolean := true;
45         begin
46             ....

```

```

44         if(first) then
45
46
47             dmemBytes(7)  <= "00000000";
48             dmemBytes(6)  <= "00000000";
49             dmemBytes(5)  <= "00000000";
50             dmemBytes(4)  <= "00000000";
51             dmemBytes(3)  <= "00000000";
52             dmemBytes(2)  <= "00000000";
53             dmemBytes(1)  <= "00000000";
54             dmemBytes(0)  <= "00000001"; --X0
55
56             dmemBytes(15) <= "00000000";
57             dmemBytes(14) <= "00000000";
58             dmemBytes(13) <= "00000000";
59             dmemBytes(12) <= "00000000";
60             dmemBytes(11) <= "00000000";
61             dmemBytes(10) <= "00000000";
62             dmemBytes(9)  <= "00000000";
63             dmemBytes(8)  <= "00000000"; --X1
64
65
66
67             dmemBytes(23) <= "00000000";
68             dmemBytes(22) <= "00000000";
69             dmemBytes(21) <= "00000000";
70             dmemBytes(20) <= "00000000";
71             dmemBytes(19) <= "00000000";
72             dmemBytes(18) <= "00000000";
73             dmemBytes(17) <= "00000000";
74             dmemBytes(16) <= "00000000"; --X2
75
76

```

registers.vhd x register_tb.vhd x

The initialization of other register memory is not displayed.

```

348
349
350     dmemBytes(247) <= "00000000";
351     dmemBytes(246) <= "00000000";
352     dmemBytes(245) <= "00000000";
353     dmemBytes(244) <= "00000000";
354     dmemBytes(243) <= "00000000";
355     dmemBytes(242) <= "00000000";
356     dmemBytes(241) <= "00000000";
357     dmemBytes(240) <= "00000000"; --X30
358
359
360     dmemBytes(255) <= "00000000";
361     dmemBytes(254) <= "00000000";
362     dmemBytes(253) <= "00000000";
363     dmemBytes(252) <= "00000000";
364     dmemBytes(251) <= "00000000";
365     dmemBytes(250) <= "00000000";
366     dmemBytes(249) <= "00000000";
367     dmemBytes(248) <= "00000000"; --X31
368
369     first := false;
370
371     end if;

```

```

Ln#
375 if Clock = '1' and Clock'event and RegWrite='1' then
376   addr:=to_integer(unsigned(WR)) * 8;
377   if(addr<248) then
378     dmemBytes(addr+7) <= WD(63 downto 56);
379     dmemBytes(addr+6) <= WD(55 downto 48);
380     dmemBytes(addr+5) <= WD(47 downto 40);
381     dmemBytes(addr+4) <= WD(39 downto 32);
382     dmemBytes(addr+3) <= WD(31 downto 24);
383     dmemBytes(addr+2) <= WD(23 downto 16);
384     dmemBytes(addr+1) <= WD(15 downto 8);
385     dmemBytes(addr) <= WD(7 downto 0);
386   elsif (addr=248)
387     then report "X31 cannot be written" severity error;
388     else report "Invalid Register address" severity error;
389     end if;
390   elsif RegWrite='0' then
391     addr1 := to_integer(unsigned(RR1)) * 8;
392     addr2:=to_integer(unsigned(RR2)) * 8;
393     if addr1<248 and addr2<248 then
394       RD1 <= dmemBytes(addr1) & dmemBytes(addr1+1)
395       & dmemBytes(addr1+2) & dmemBytes(addr1+3)
396       & dmemBytes(addr1+4) & dmemBytes(addr1+5)
397       & dmemBytes(addr1+6) & dmemBytes(addr1+7);
398
399
400
401       RD2 <= dmemBytes(addr2) & dmemBytes(addr2+1)
402       & dmemBytes(addr2+2) & dmemBytes(addr2+3)
403       & dmemBytes(addr2+4) & dmemBytes(addr2+5)
404       & dmemBytes(addr2+6) & dmemBytes(addr2+7);
405     elsif addr1 = 248 or addr2 =248 then
406       report "X31 cannot be written" severity error;
407
408
409
410
411
412
413
414 dmemBytes(154) & dmemBytes(155) &
415 dmemBytes(156) & dmemBytes(157) &
416 dmemBytes(158) & dmemBytes(159) &
417 dmemBytes(160) & dmemBytes(161) &
418 dmemBytes(162) & dmemBytes(163) &
419 dmemBytes(164) & dmemBytes(165) &
420 dmemBytes(166) & dmemBytes(167) &
421 dmemBytes(168) & dmemBytes(169) &
422 dmemBytes(170) & dmemBytes(171) &
423 dmemBytes(172) & dmemBytes(173) &
424 dmemBytes(174) & dmemBytes(175) &
425 dmemBytes(176) & dmemBytes(177) &
426 dmemBytes(178) & dmemBytes(179) &
427 dmemBytes(180) & dmemBytes(181) &
428 dmemBytes(182) & dmemBytes(183);
429 DEBUG_TMP_REGS <= dmemBytes(72) & dmemBytes(73) &
430 dmemBytes(74) & dmemBytes(75) &
431 dmemBytes(76) & dmemBytes(77) &
432 dmemBytes(78) & dmemBytes(79) &
433 dmemBytes(80) & dmemBytes(81) &
434 dmemBytes(82) & dmemBytes(83) &
435 dmemBytes(84) & dmemBytes(85) &
436 dmemBytes(86) & dmemBytes(87) &
437 dmemBytes(88) & dmemBytes(89) &
438 dmemBytes(90) & dmemBytes(91) &
439 dmemBytes(92) & dmemBytes(93) &
440 dmemBytes(94) & dmemBytes(95) &
441 dmemBytes(96) & dmemBytes(97) &
442 dmemBytes(98) & dmemBytes(99) &
443 dmemBytes(100) & dmemBytes(101) &
444 dmemBytes(102) & dmemBytes(103);
445 end behavior;

```

Then I built the testbench to test the registers. The testbench of mine is that (in the simulation, the signals in the simulation have the same names as the related ports.):

```

50
51 );
52
53 sim_proc: process
54 begin
55     RegWrite <= '0';
56     RR1  <= "01001"; --X9
57     RR2  <= "01010"; --X10
58     wait FOR 50 ns;
59
60     RegWrite <= '1';
61     WR<="10011";
62     WD<=X"FFFFFFFFFFFFFFFF";
63     wait FOR 50 ns;
64
65     RegWrite <= '1';
66     WR<="11111";
67     WD<=X"FFFFFFFFFFFFFFFF";
68     wait;
69
70
71 END PROCESS;
72 Clock <= not Clock after 25 ns;
73 END;
74

```

And figure14 to16 are the simulation result:

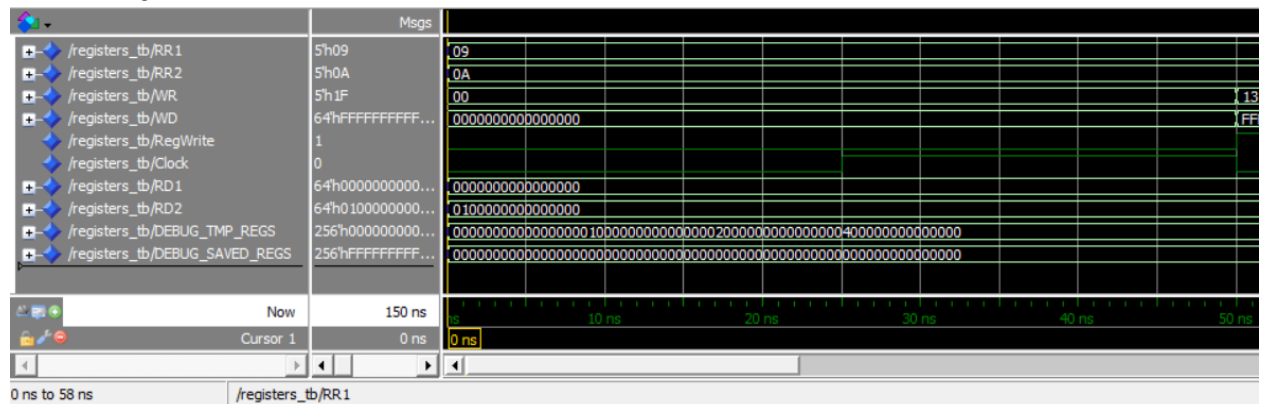


Figure 15 (0ns-50ns)

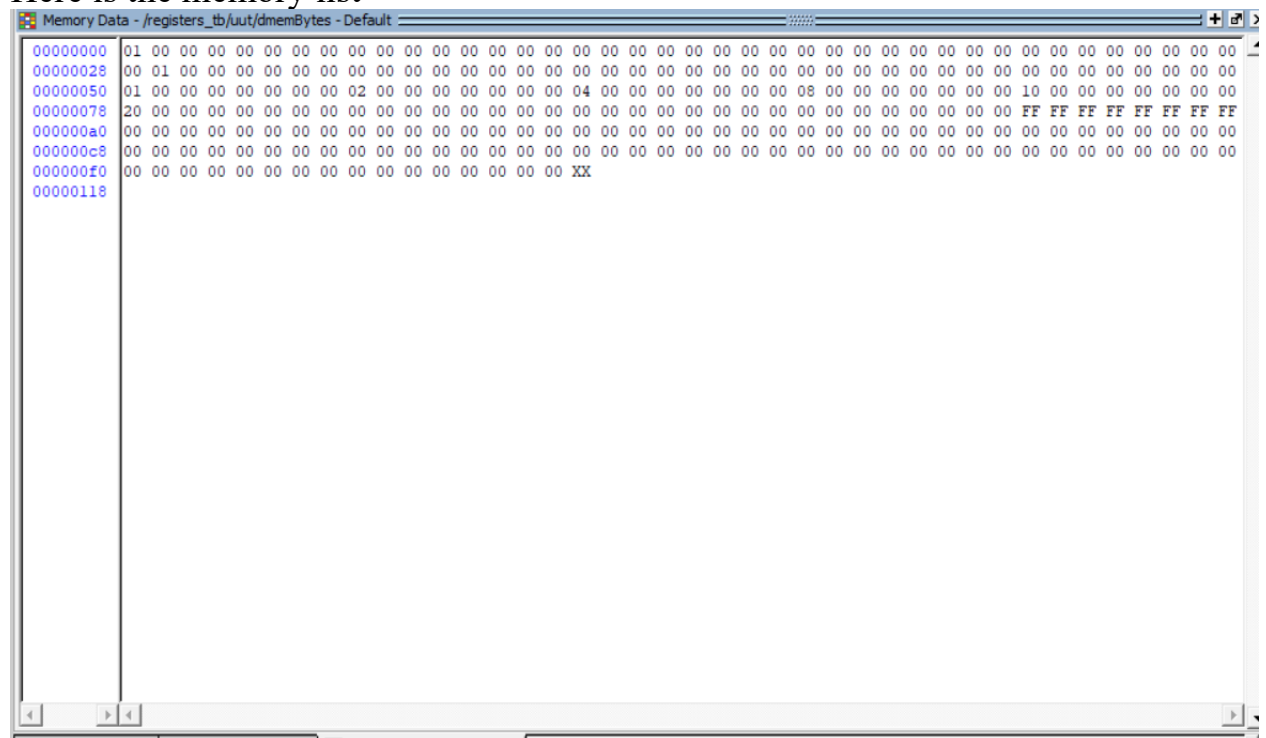
During $t = 0$ ns to $t = 50$ ns, I try to read the data stored in the X9 and X10
So, I set:

```

RegWrite <= '0'
RR1  <= "01001"; --X9
RR2  <= "01010"; --X10

```

Here is the memory list



So, the result is correct.

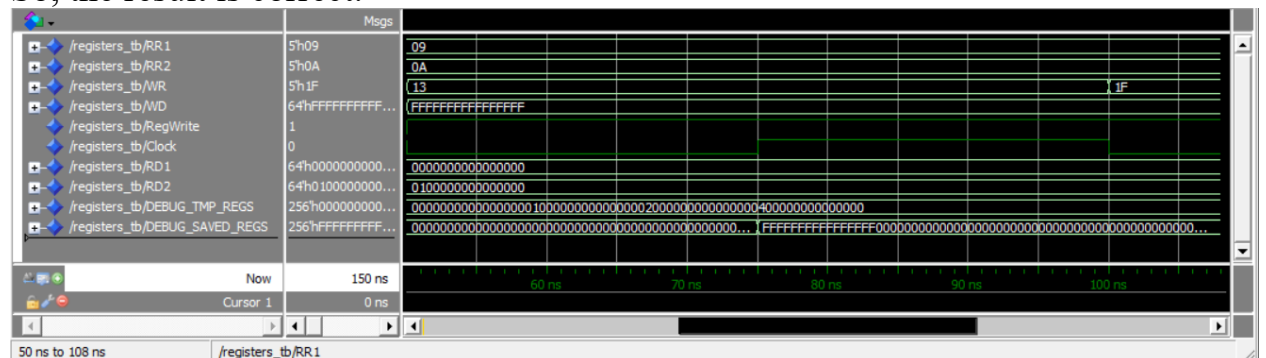


Figure 15 (50ns-100ns)

During $t = 50 \text{ ns}$ to $t = 100 \text{ ns}$, I try to write the data in in the X20. So, I set:

```
RegWrite <= '1'
WR <= "10011"
WD <= X"FFFFFFFFFFFFFFFF"
```

According to the memory list, the result is correct.

During $t = 100 \text{ ns}$ to $t = 150 \text{ ns}$, I try print the error message, So, I set:

```
RegWrite <= '1'
WR <= "11111"
WD <= X"FFFFFFFFFFFFFFFF"
```

And this is the result:

```
# Loading work.registers(benavi) #1
add wave -position insertpoint sim:/registers_tb/*
VSIM 17> run
# ** Error: X31 cannot be written
#   Time: 125 ns  Iteration: 0  Instance: /registers_tb/uut
VSIM 18>
```
