16 bit microprocessor report Report

UPPSALA
UNIVERSITET

date: April 29, 2022
author: Yuzhi Chen
course: Digital Electronics Design with VHDL

Abstract

The project task is to design a 16 bit microprocessor kernel using fetch and execute states with ADD, STORE, LOAD, JUMP and JNEG functions.
The final design contains three test cases which are both simulated on software and implemented on hardware using *DE*1_*SOC* board.

Contents

1 Introduction

This project gives the possibility of building up a basic microprocessor with Program Counter (PC), Instruction Register (IR), Accumulator Register (AC), Arithmetic Logic Unit (ALU), Memory Interface (MIF). The processor is connected to a Memory which stores instructions and values and a hex-display to show the output of Accumulator register. Instructions are 16 bits with 8 bits op code and 8 bits address part pointing on data or the next operation.

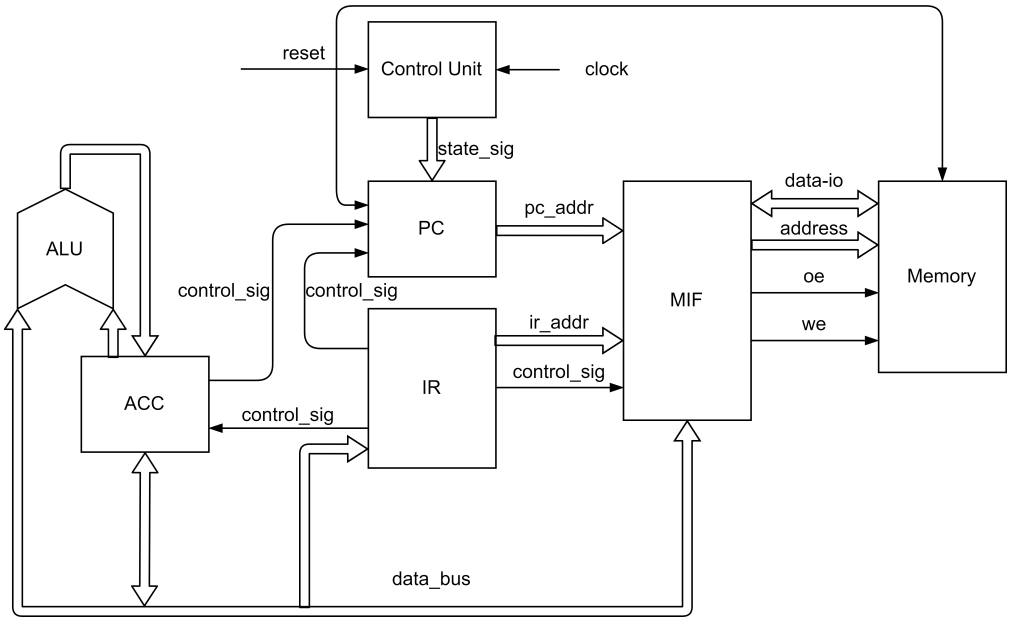

Figure 1.1: Structure of 16 bit microprocessor

The project also gives a good and valuable knowledge about design of state machine and timing issue in hardware programming.

2 Project Description

2.1 Tools Software

The project task is to design a microprocessor kernel that in the end should be tested in hardware. The Quartus project is using a CYCLONE V: 5CSEMA5F31C6.
The Intel Quartus Prime Design Software used to design for Intel FPGAs, SoCs, and complex programmable logic device (CPLD) from design entry and synthesis to optimization, verification, and simulation. ModelSim simulates behavioral, RTL, and gate-level code - delivering increased design quality and debug productivity with platform-independent compile. Single Kernel Simulator technology enables transparent mixing of VHDL and Verilog in one design. The SignalTap Logic Analyzer helps debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment. Defining custom trigger-condition logic provides greater accuracy and improves the ability to isolate problems.

2.2 Design

The design is using smaller components, these units are: Clock Frequency Divider: generate low frequency clock signal Control Unit: control the state machine Program Counter (PC): locate present address Instruction Register (IR): decode and execute Accumulator register (AC): store immediate resulte Arithmetic Logic Unit (ALU): arithmetic operations Memory Interface (MIF): access memory and data bus Hex Display: display the value of Accumulator register (AC) These units perform

simple tasks that together as a microprocessor that are doing intended operations with a sequence of instructions.

3 Theory

3.1 The design of state machine

The Control Units contains a state machine with two states fetch and execute.
When fetch state, the program counter (PC) keeps the address of the running instruction and do a instruction-fetch. After a successful fetch the program counter is increased by '1'. instruction is found at the address in Memory and go through the memory interface (MIF). The memory interface (MIF) push the value in buswire and then the value is loaded in the instruction register (IR) where the decode will happen. The fetch state can be described that way in microcode like

MIF = PC
Read memory, (Instruction from memory)
IR = memory bus
PC=PC+1

When execute state, The decoding of the instruction part in IR leads to the execution of operations on different registers and the arithmetic logic unit. New data is found in address and push into buswire so the accumulator (AC) which is the primary intermediate storage register and the arithmetic logic unit (ALU) for operations can use them . The execute state can be described that way in microcode like

Decode the op-code to find next state
MIF=IR
Start a memory read

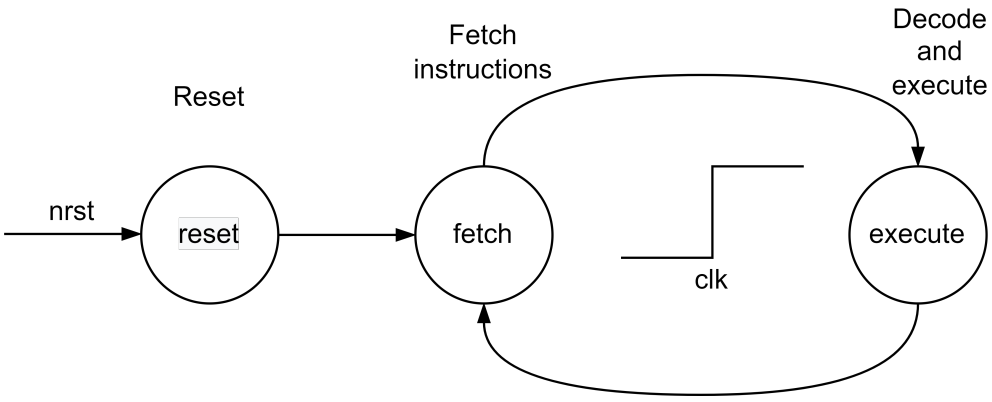

Figure 3.1: State machine design

3.2 The design of bit instructions

The memory and data parts should then be 16 bits wide. The 8 bits op-code leads to an operation when it is decoded in a control unit. The 8 bits address part is pointing on data or the next operation.

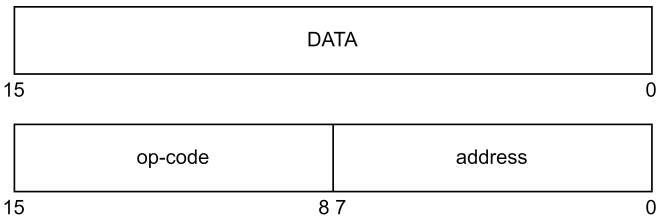

Figure 3.2: Data instruction design

The decode table of op code is as follows.

			op-code
ADD	address	$AC <= AC + contentatthememoryaddress$	00
STORE	address	$contentatthememoryaddress <= AC$	01
LOAD	address	$AC <= contentatthememoryaddress$	02
JUMP	address	$PC <= address$	03
JNEG	address	$if AC < 0 then PC <= address$	04

4 Implementation

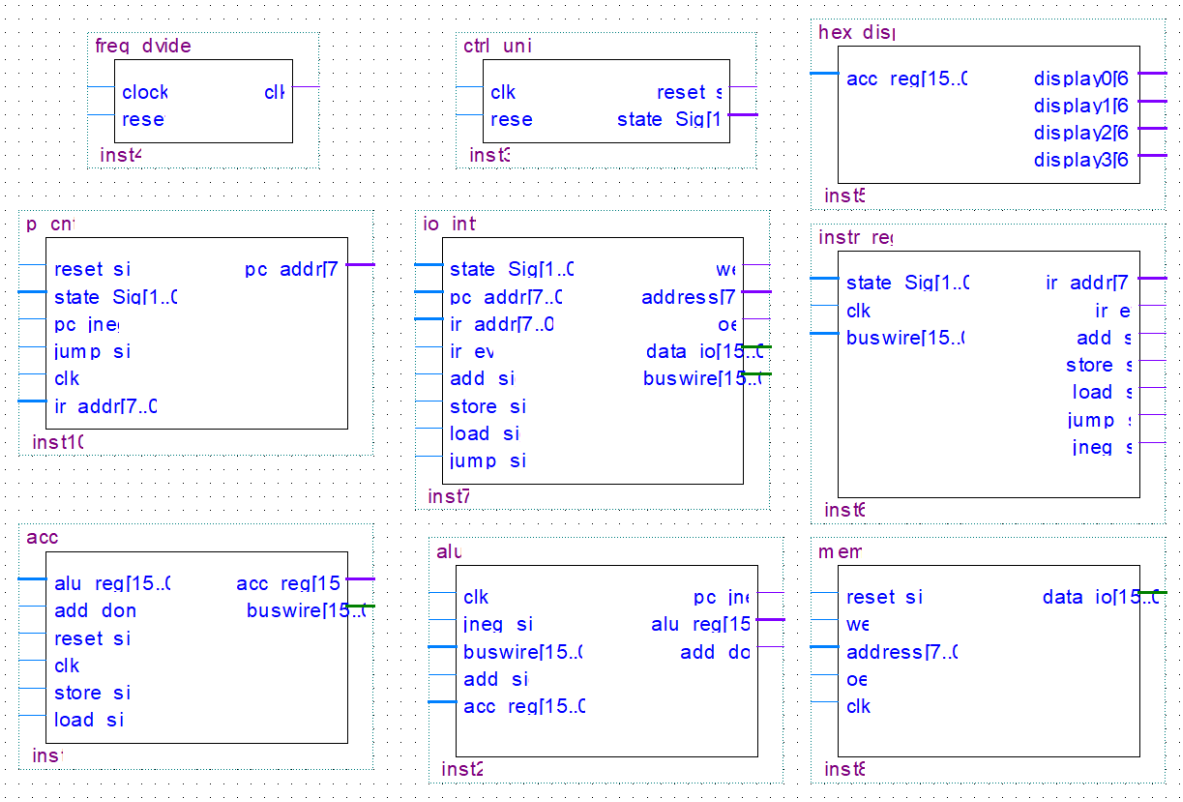

Figure 4.1: All units in this design

4.1 Control Unit

The control Unit viewed as a component has 2 inputs and 2 outputs.

Pins	Type	bits	description
clk	Input	1	clock signal
reset	Input	1	reset button input
reset-sig	Output	1	reset signal
state-sig	Output	2	state signal

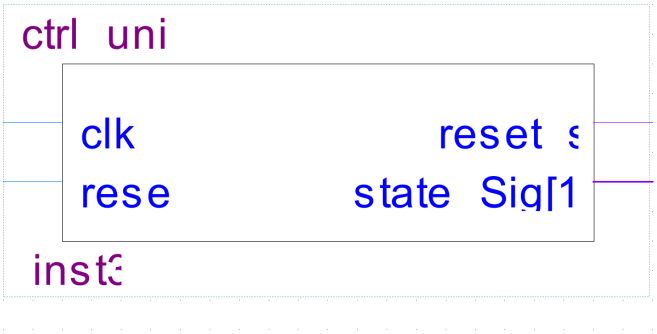

Figure 4.2: Control Unit block

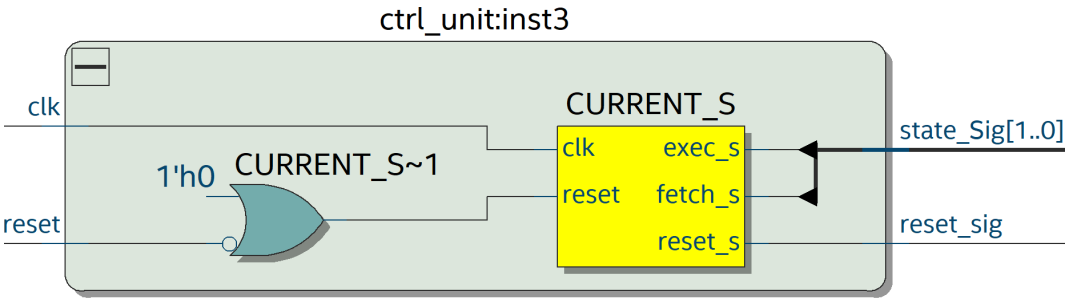

Figure 4.3: Control Unit RTL

4.2 Program Counter

The Program Counter viewed as a component has 6 inputs and 1 output.

Pins	Type	bits	description
reset-sig	Input	1	reset signal
state-sig	Input	2	state signal
pc-jneg	Input	1	do JNEG when receive signal
jump-sig	Input	1	do JUMP when receive signal
clk	Input	1	clock signal
ir-addr	Input	8	address from IR
pc-addr	Output	8	address to MIF

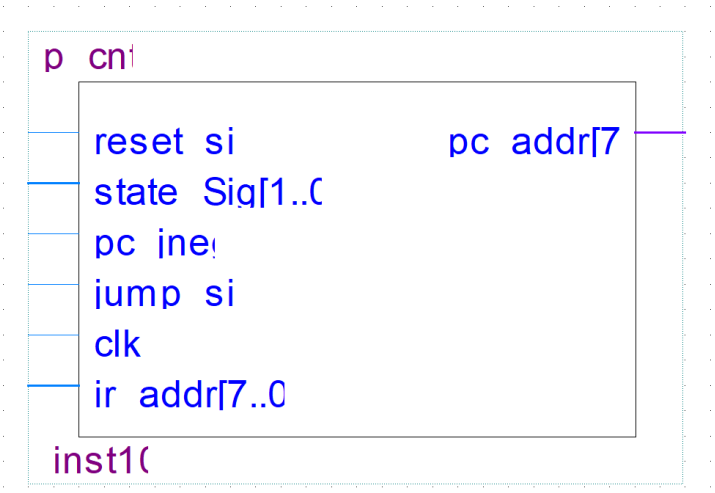

Figure 4.4: Program Counter block

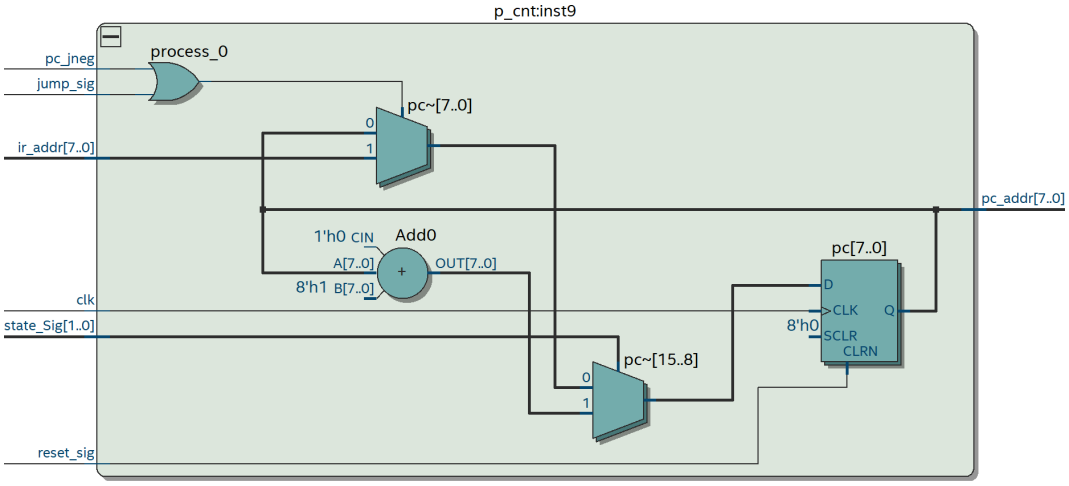

Figure 4.5: Program Counter RTL

4.3 Memory Interface

The Memory Interface viewed as a component has 8 inputs and 5 outputs.

Pins	Type	bits	description
state-sig	Input	2	state signal
pc-addr	Input	8	address from PC
ir-addr	Input	8	address from IR
ir-ev	Input	1	switch the address between IR and PC
add-sig	Input	1	push value to bus when receive signal
store-sig	Input	1	push value to Memory when receive signal
load-sig	Input	1	push value to bus when receive signal
jump-sig	Input	1	push value to bus when receive signal
we	Output	1	writing enable
address	Output	8	address to Memory
oe	Output	1	Output enable
data-io	Input/Output	16	databus between MIF and Memory
buswire	Input/Output	16	databus in microprocessor

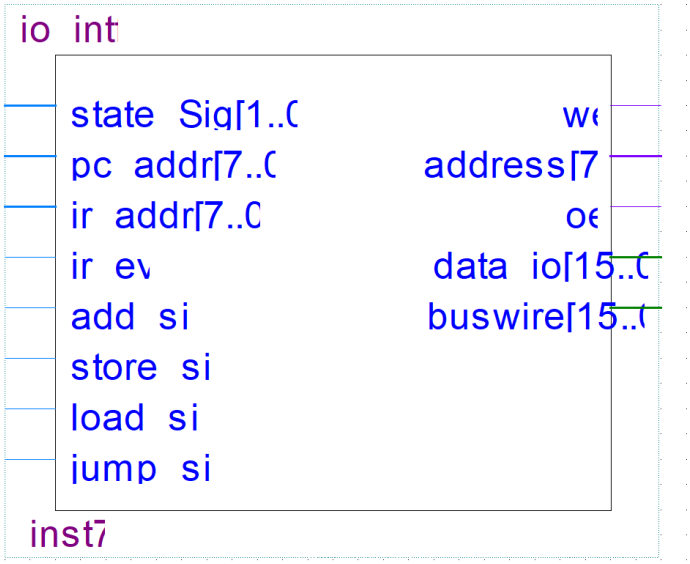

Figure 4.6: Memory Interface block

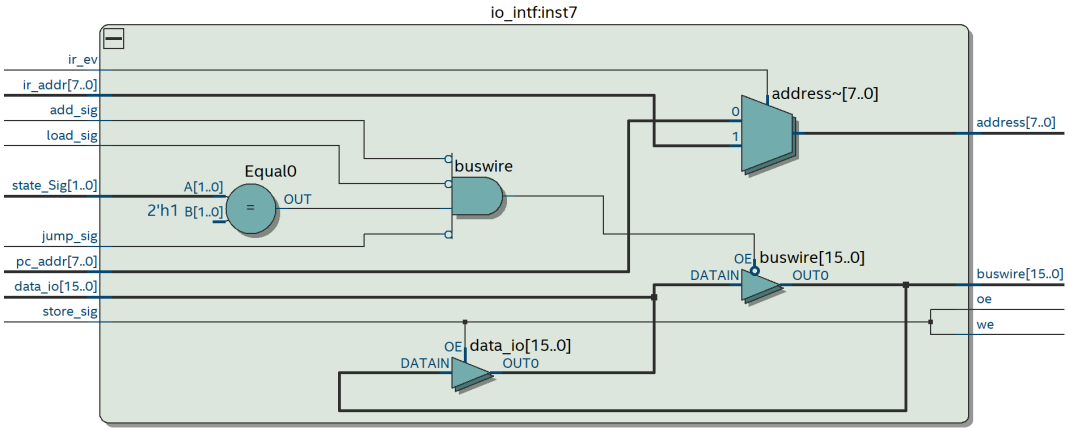

Figure 4.7: Memory Interface RTL

4.4 Instruction Register

The Instruction Register viewed as a component has 3 inputs and 7 outputs.

Pins	Type	bits	description
state-sig	Input	2	state signal
clk	Input	1	clock signal
buswire	Input	16	databus in microprocessor
ir-addr	Output	8	address to MIF
ir-ev	Output	1	switch the address between IR and PC
add-sig	Output	1	operation ADD
store-sig	Output	1	operation STORE
load-sig	Output	1	operation LOAD
jump-sig	Output	1	operation JUMP
jneg-sig	Output	1	operation JNEG

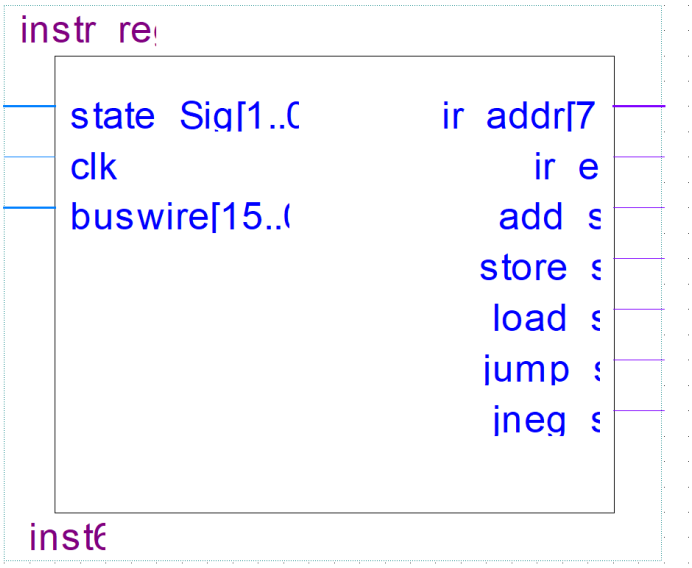

Figure 4.8: Instruction Register block

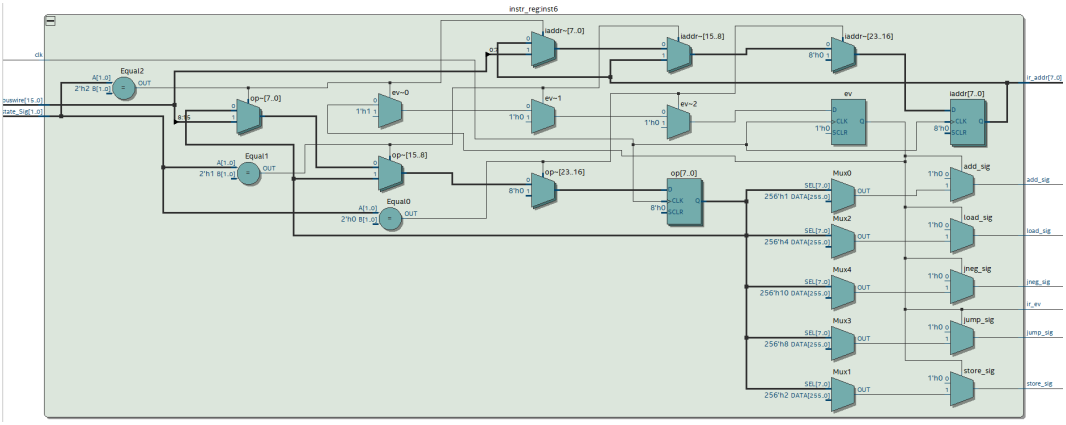

Figure 4.9: Instruction Register RTL

4.5 Accumulator register

The Accumulator viewed as a component has 6 inputs and 2 outputs.

Pins	Type	bits	description
alu-reg	Input	16	data from ALU
add-done	Input	1	get ALU value when receive signal
reset-sig	Input	1	reset ACC to 0
clk	Input	1	clock signal
store-sig	Input	1	wrtie value to bus
load-sig	Input	1	load value from bus
acc-reg	Output	16	data to ALU
buswire	Input/Output	16	databus in microprocessor

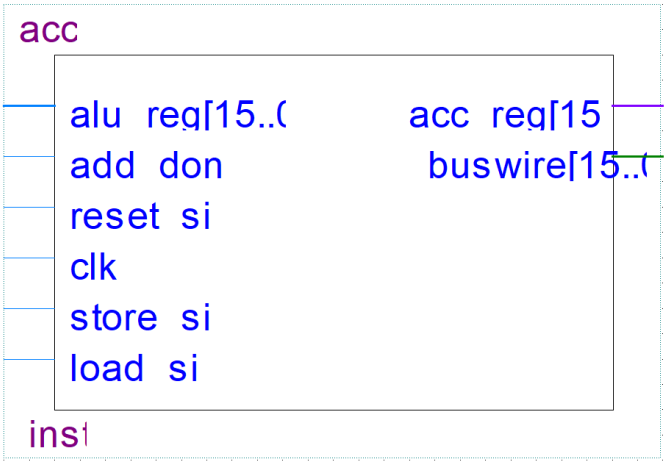

Figure 4.10: Accumulator register block

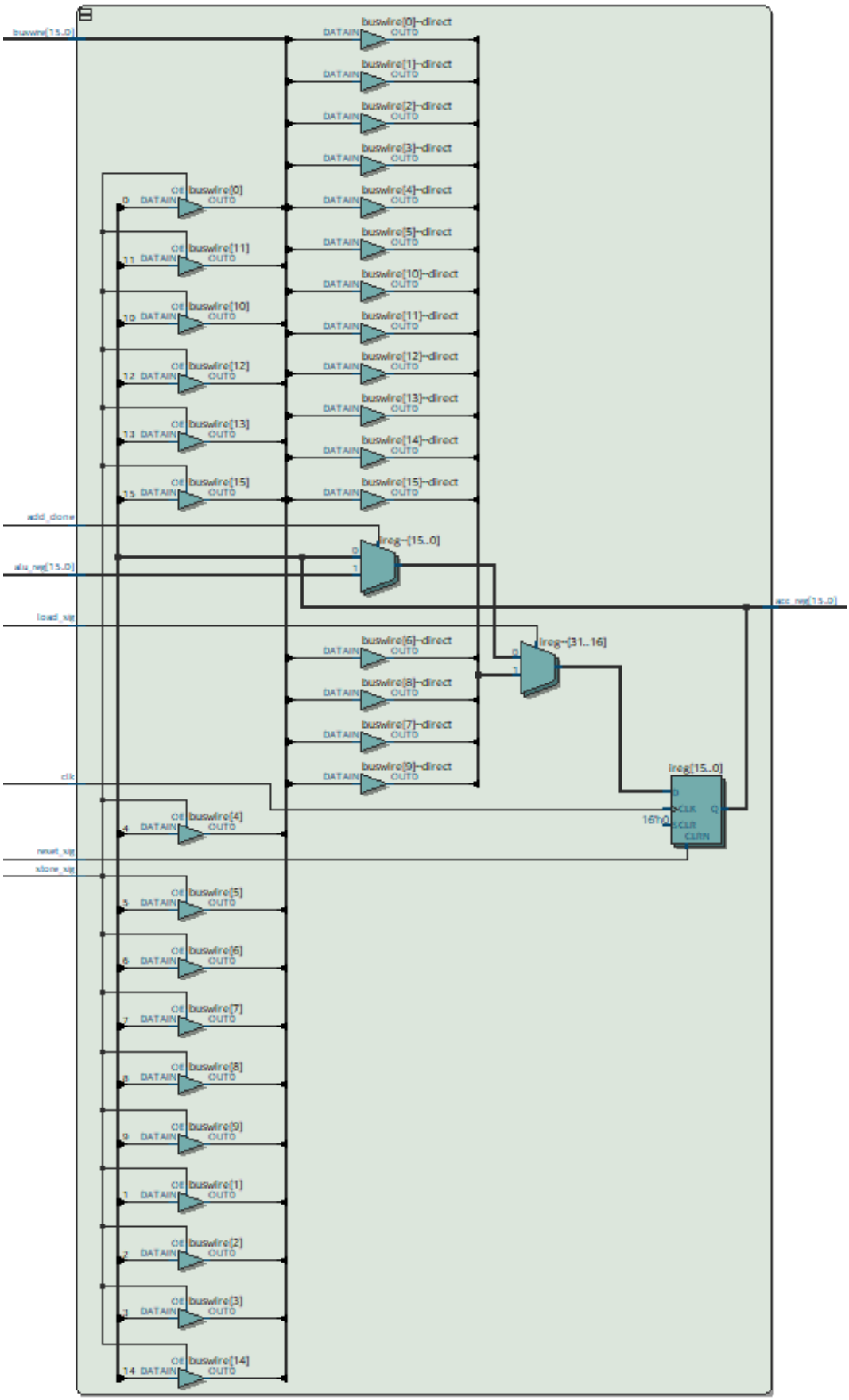

Figure 4.11: Accumulator register RTL

4.6 Arithmetic Logic Unit

The Arithmetic Logic Unit viewed as a component has 5 inputs and 3 outputs.

Pins	Type	bits	description
clk	Input	1	clock signal
jneg-sig	Input	1	do JENG operation
buswire	Input	16	databus in microprocessor
add-sig	Input	1	do ADD operation
acc-reg	Input	16	data from ACC
pc-jneg	Input	1	tell PC do JENG operation
alu-reg	Output	16	data to ACC
add-done	Output	1	tell ACC add complete

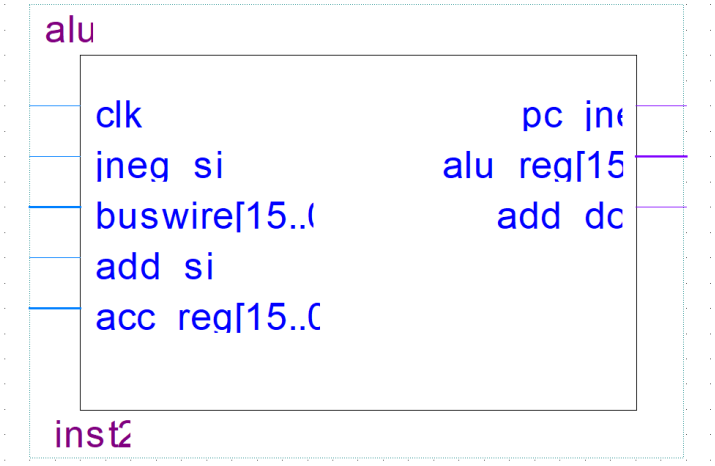

Figure 4.12: Arithmetic Logic Unit block

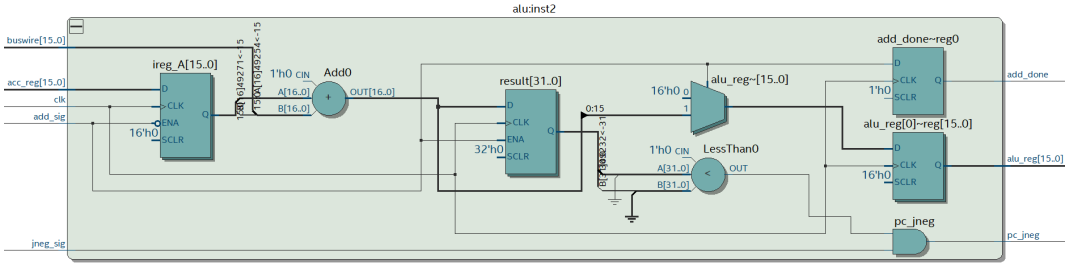

Figure 4.13: Arithmetic Logic Unit RTL

4.7 Memory

The Memory viewed as a component has 5 inputs and 1 output.

Pins	Type	bits	description
reset-sig	Input	1	reset Memory to preset value
we	Input	1	write enable
address	Input	8	address from MIF
oe	Input	1	output enable
clk	Input	1	clock signal
data-io	Input/Output	16	databus between MIF and Memory

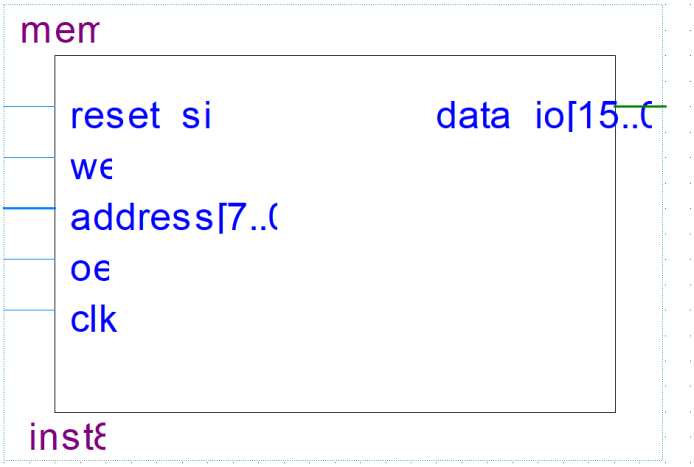

Figure 4.14: Memory block

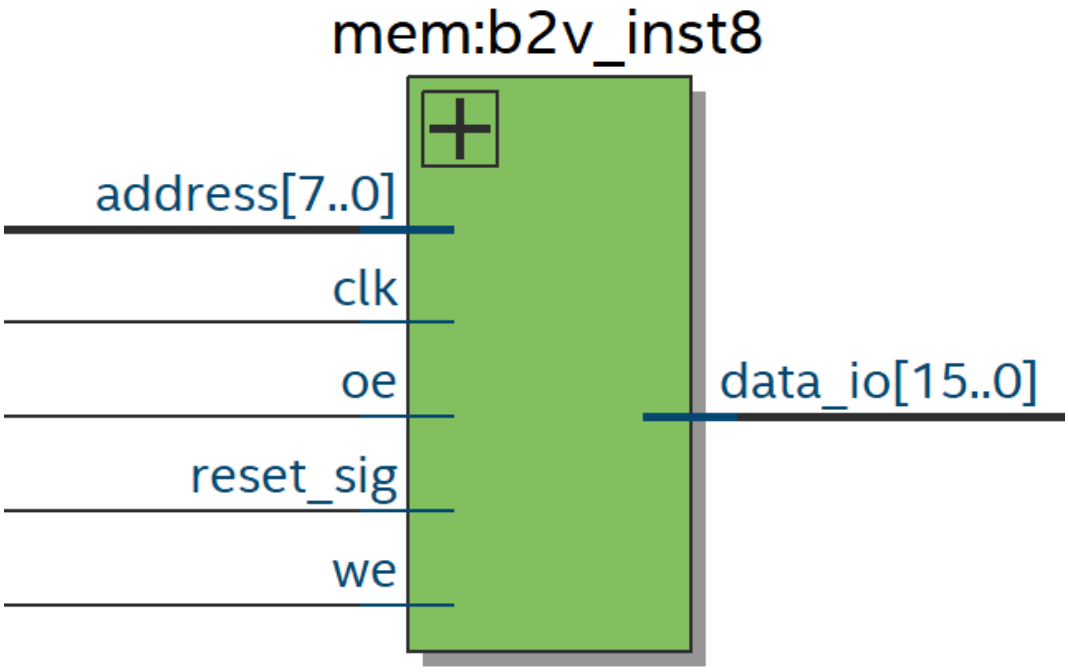

Figure 4.15: Memory RTL

4.8 Clock Frequency Divider

The Clock Frequency Divider viewed as a component has 2 inputs and 1 output.

Pins	Type	bits	description
clock	Input	1	high frequency clock signal
reset	Input	1	reset button input
clk	Input	1	low frequency clock signal

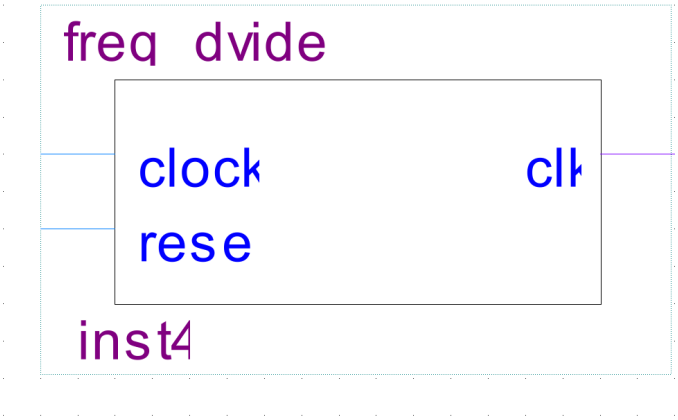

Figure 4.16: Clock Frequency Divider block

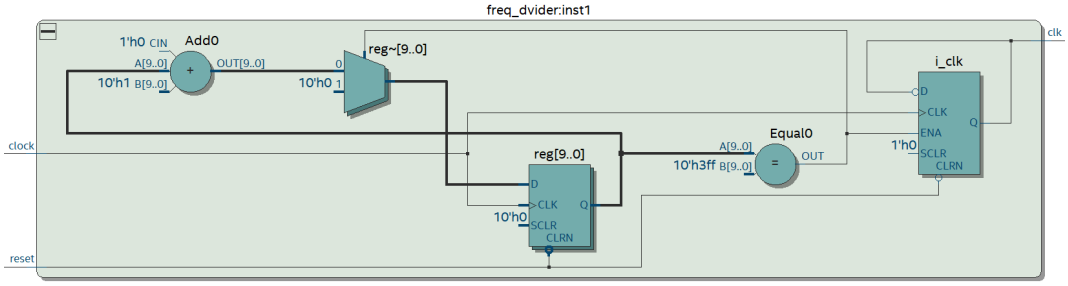

Figure 4.17: Clock Frequency Divider RTL

4.9 Hex Display

The Hex Display viewed as a component has 1 input and 4 outputs.

Pins	Type	bits	description
acc-reg	Input	16	data from ACC
display0-3	Output	1	decoded hex signal

Figure 4.18: Hex Display block

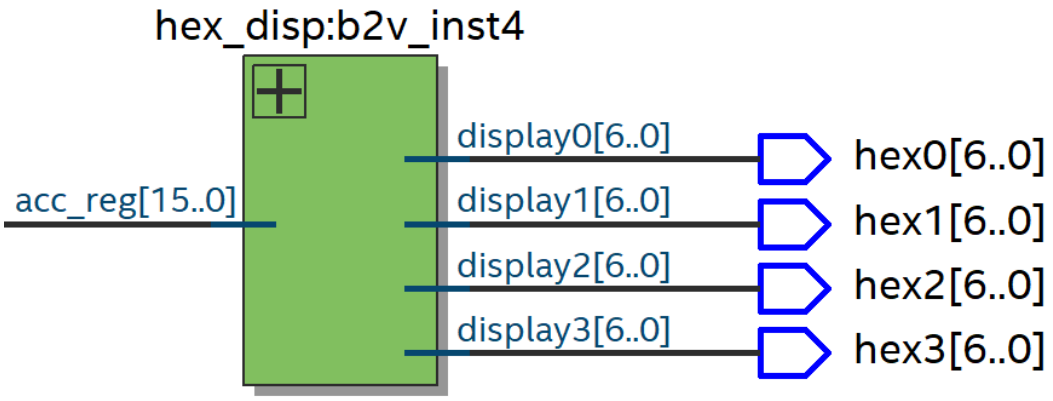

Figure 4.19: Hex Display RTL

4.10 Complete Design

The complete design as shown in the block diagram.

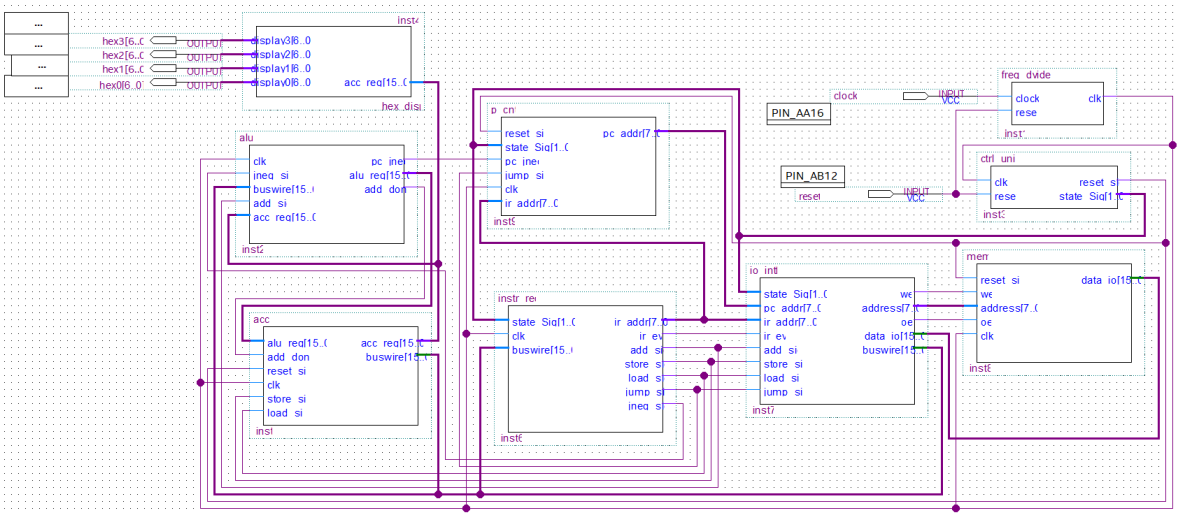

Figure 4.20: The complete design

5 Tests Result

5.1 A=B+C

Figure 5.1 shows the code for 'A=B+C' process.

1. mem-array(0) >= "0000001011111110"; –02FE:LOAD 254
   Load the value in mem-array(254), ACC should contains a value 3 now.

2. mem-array(1) >= "0000000011111101"; –00FD:ADD 253
   Add the value in mem-array(253), ACC should contains a value 7 now.

3. mem-array(2) >= "0000000111111111"; –01FF:STORE 255
   Store the value in mem-array(255), mem-array(255) should contains a value 7 now.

4. mem-array(3) >= ”0000001011111111”; –02FF:LOAD 255
   Load the value in mem-array(255), ACC should contains a value 7 now. It is used to check if value is loaded in mem-array(255).

5. mem-array(4) >= ”0000001100000100”; –0304:JUMP 04
   PC should jump to 04, which means this instruction will execute infinitely. ACC should hold the value 7.

```
mem_array(0) <= "0000001011111110";   --02FE:LOAD 254
mem_array(1) <= "0000000011111101";   --00FD:ADD 253
mem_array(2) <= "0000000111111111";   --01FF:STORE 255
mem_array(3) <= "0000001011111111";   --02FF:LOAD 255
mem_array(4) <= "0000001100000100";   --0304:JUMP 04
mem_array(253) <= "0000000000000100";   --0004
mem_array(254) <= "0000000000000011";   --0003
```

Figure 5.1: A=B+C Code

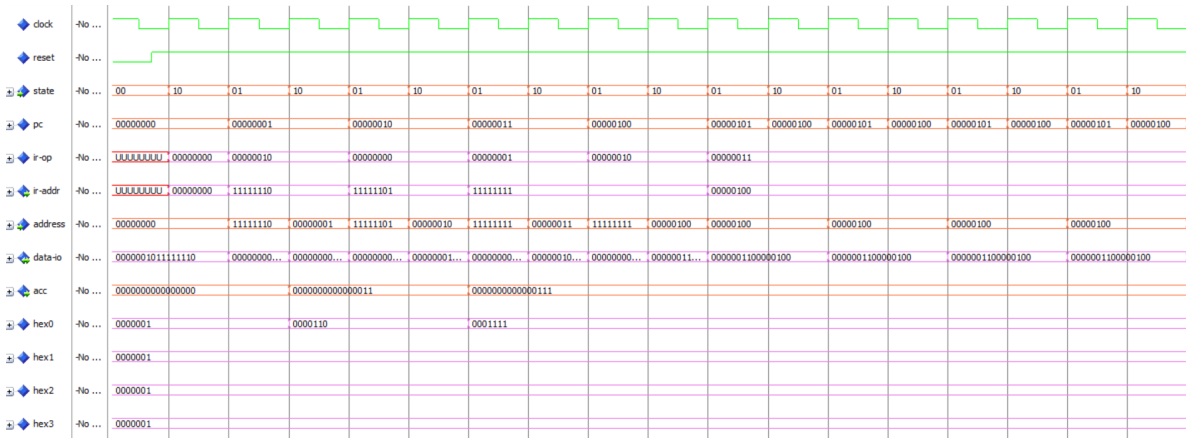

Figure 5.2: RTL simulation of the test case

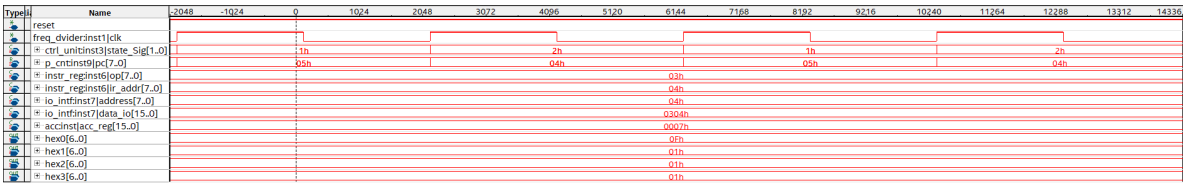

Figure 5.3: SignalTap of the test case

5.2 If AC<0 then PC <= address

Figure 5.4 shows the code for ’If AC<0 then PC <= address’ process.

1. mem-array(0) >= ”0000001011111110”; –02FE:LOAD 254
   Load the value in mem-array(252), ACC should contains a value 3 now.

2. mem-array(1) >= ”0000000011111101”; –00FD:ADD 253
   Add the value in mem-array(253), ACC should contains a value -1 now.

3. mem-array(2) >= ”0000010000000100”; –0404:JNEG 04
   Rise a JNEG process. In this case, PC should jump to 04.

4. mem-array(3) >= ”0000001011111110”; –02FE:LOAD 254
   In this case this instruction should be skiped and ACC should contains a value -1.

5. mem-array(4) >= ”0000001100000100”; –0304:JUMP 04
   PC should jump to 04, which means this instruction will execute infinitely. ACC should hold the value -1.

```
mem_array(0) <= "0000001011111110";   --02FE:LOAD 254
mem_array(1) <= "0000000111111101";   --00FD:ADD 253
mem_array(2) <= "0000010000000100";   --0404:JNEG 04
mem_array(3) <= "0000001011111110";   --02FE:LOAD 254
mem_array(4) <= "0000001100000100";   --0304:JUMP 04
mem_array(253) <= "1111111111111011";   -- -4
mem_array(254) <= "0000000000000011";   -- 3
```

Figure 5.4: If AC<0 then PC <= address Code

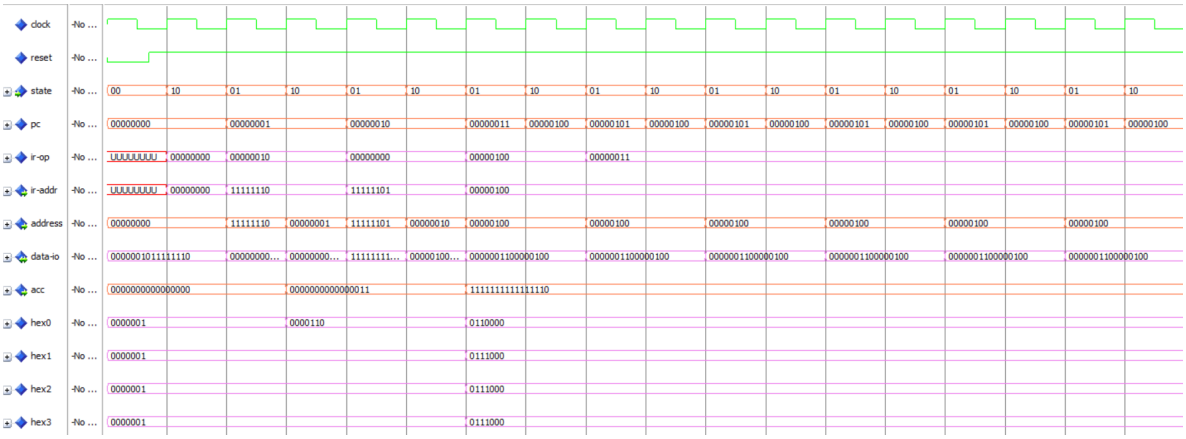

Figure 5.5: RTL simulation of the test case

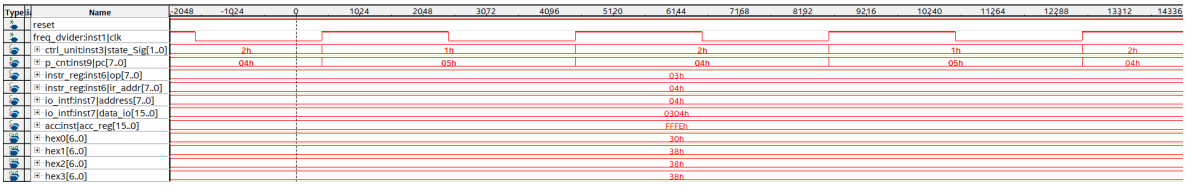

Figure 5.6: SignalTap of the test case

5.3 If A>=0 then B=C

Figure 5.4 shows the code for ’If A>=0 then B=C’ process.

1. mem-array(0) >= ”0000001011111110”; –02FE:LOAD 254
   Load the value in mem-array(254), ACC should contains a value 3 now.

2. mem-array(1) >= ”0000000111111101”; –00FD:ADD 253
   Add the value in mem-array(253), ACC should contains a value 7 now.

3. mem-array(2) >= ”0000010000000100”; –0404:JNEG 04
   Rise a JNEG process. In this case, PC will not jump.

4. mem-array(3) >= ”0000001011111110”; –02FE:LOAD 254
   Load the value in mem-array(254), ACC should contains a value 3 now.

5. mem-array(4) >= ”0000001100000100”; –0304:JUMP 04
   PC should jump to 04, which means this instruction will execute infinitely. ACC should hold the value 3.

```
mem_array(0) <= "0000001011111110";   --02FE:LOAD 254
mem_array(1) <= "0000000011111101";   --00FD:ADD 253
mem_array(2) <= "0000010000000100";   --0404:JNEG 04
mem_array(3) <= "0000001011111110";   --02FE:LOAD 254
mem_array(4) <= "0000001100000100";   --0304:JUMP 04
mem_array(253) <= "0000000000000100";   -- 4
mem_array(254) <= "0000000000000011";   -- 3
```

Figure 5.7: If A>=0 then B=C Code

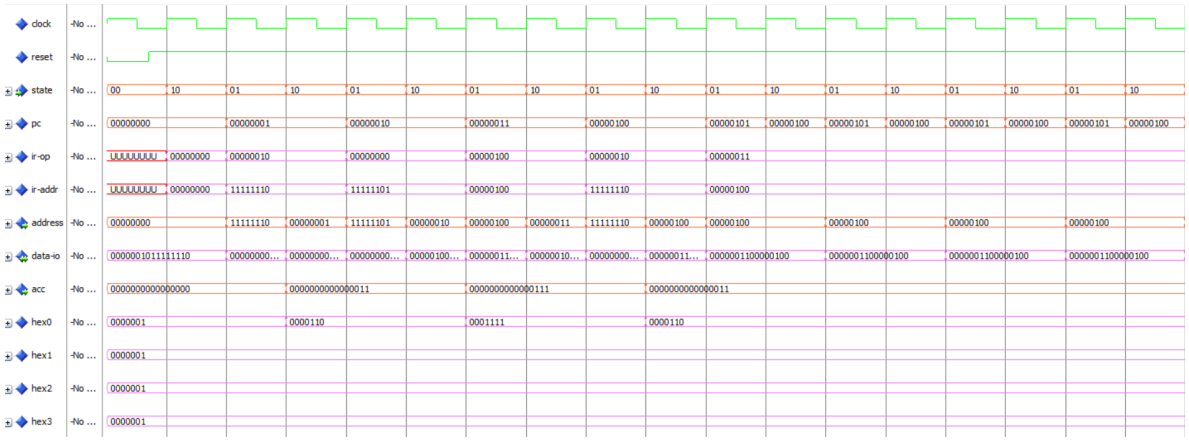

Figure 5.8: RTL simulation of the test case

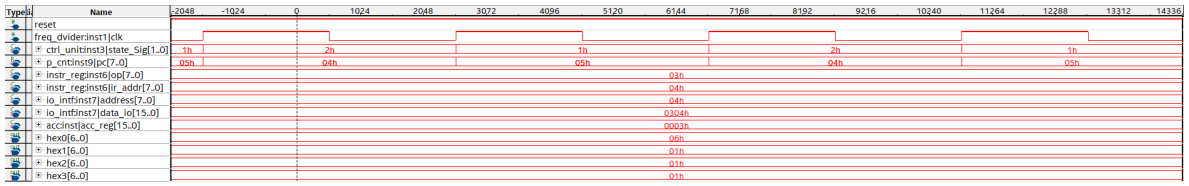

Figure 5.9: SignalTap of the test case

6 Conclusions and evaluation

By analyzing the RTL waveform and signaltap waveform, it can be seen that the microprocessor has obtained the correct result at the RTL simulation level and SignalTap simulation. The project eventually shows correct value and is able to pass all three test case on hardware.
From figure5.3 figure5.6 figure5.9, the SignalTap waveform cannot show the first few clock cycle of the wave. It can only show the final wave in the infinite loop.
Through many experiments and analysis, I found the problem might be that the setting of SignalTap trigger is not correct.

7 Appendix A. Source Code

7.1 Control Unit

```
1  LIBRARY ieee;
2  USE ieee. std_logic_1164 .ALL;
3
4  ENTITY ctrl_unit IS
5      PORT(
6                      clk : IN  std_logic ;
7                      reset : IN  std_logic ;
8                       reset_sig : OUT std_logic;
9                      state_Sig : OUT std_logic_vector(1 DOWNTO 0)  --state signal
10     );
11 END ctrl_unit;
12
13 ARCHITECTURE ctrl_unit_architecture OF ctrl_unit IS
14 TYPE state_type IS ( reset_s , fetch_s , exec_s );   --define state type
15 SIGNAL CURRENT_S,NEXT_S: state_type;
16 SIGNAL iev: std_logic ;
17 BEGIN
18
19     PROCESS(reset, clk)
20     BEGIN
21         IF  reset  = '0'  THEN
22             CURRENT_S <= reset_s;  --reset the state machine
23         ELSIF  rising_edge (clk )  THEN
24             CURRENT_S <= NEXT_S;
25         END IF;
26     END PROCESS;
27
28     PROCESS(CURRENT_S)
29     BEGIN
30          reset_sig  <= '0';
31         CASE CURRENT_S IS
32         WHEN reset_s =>  --reset state
33             NEXT_S <= fetch_s;
34              state_Sig  <= "00";
35              reset_sig  <= '1';
36
37         WHEN fetch_s =>  --fetch state
38             NEXT_S <= exec_s;
39              state_Sig  <= "10";
40
41         WHEN exec_s =>  --execute state
42             NEXT_S <= fetch_s;
43              state_Sig  <= "01";
44
45         WHEN OTHERS =>
46             state_Sig  <= "XX";
47         END CASE;
48     END PROCESS;
49
50 END  ctrl_unit_architecture ;
```

7.2 Program Counter

```
1  LIBRARY ieee;
2  USE ieee. std_logic_1164 .ALL;
```

```
3  USE ieee. std_logic_unsigned .ALL;
4
5  ENTITY p_cnt IS
6      PORT(
7           reset_sig : IN  std_logic ;
8           state_Sig : IN  std_logic_vector (1 DOWNTO 0);
9           pc_jneg: IN  std_logic ;
10          jump_sig: IN  std_logic ;
11          clk: IN  std_logic ;
12          ir_addr : IN  std_logic_vector (7 DOWNTO 0);
13          pc_addr: OUT std_logic_vector(7 DOWNTO 0)
14          );
15 END p_cnt;
16
17 ARCHITECTURE p_cnt_architecture OF p_cnt IS
18 SIGNAL pc: std_logic_vector (7 DOWNTO 0);
19 BEGIN
20
21     PROCESS(clk,reset_sig)
22     BEGIN
23         IF  reset_sig ='1' THEN  --reset program counter
24             pc <= "00000000";
25         ELSIF rising_edge (clk) THEN
26             IF  state_Sig (1) = '1' THEN  --pc add 1 when exec_s
27                 pc <= pc + '1';
28             ELSIF jump_sig='1' OR pc_jneg='1' THEN  --pc jump
29                 pc <= ir_addr;
30             END IF;
31         END IF;
32     END PROCESS;
33     pc_addr <= pc;  --give address to MIF
34
35 END p_cnt_architecture;
```

7.3 Memory Interface

```
1  LIBRARY ieee;
2  USE ieee. std_logic_1164 .ALL;
3  USE ieee. std_logic_unsigned .ALL;
4
5  ENTITY io_intf IS
6      PORT(
7                      state_Sig : IN  std_logic_vector (1 DOWNTO 0);
8                      pc_addr: IN  std_logic_vector (7 DOWNTO 0);  --address from pc
9                      ir_addr : IN  std_logic_vector (7 DOWNTO 0);  --address from ir
10                     ir_ev : IN  std_logic ;
11                     add_sig: IN  std_logic ;
12                     store_sig : IN  std_logic ;
13                     load_sig : IN  std_logic ;
14                     jump_sig: IN  std_logic ;
15                     we: OUT std_logic;  --write enable
16                     address: OUT std_logic_vector(7 DOWNTO 0);
17                     oe: OUT std_logic;  --output enable
18                     data_io: INOUT std_logic_vector(15 DOWNTO 0);
19                     buswire: INOUT std_logic_vector(15 DOWNTO 0)
20         );
```

```
21 END io_intf;
22
23 ARCHITECTURE io_intf_architecture OF io_intf IS
24 BEGIN
25
26     buswire <= (OTHERS=>'Z') WHEN state_Sig="01" AND jump_sig='0' AND load_sig='0' AND
       add_sig='0' ELSE data_io;
27     --output to MEMORY
28     address <= ir_addr WHEN ir_ev='1' ELSE pc_addr;
29     data_io <= buswire WHEN store_sig='1' ELSE (OTHERS=>'Z');
30     PROCESS(store_sig)
31     BEGIN
32         IF store_sig ='1' THEN --output to MEMORY
33             we <= '1';
34             oe <= '1';
35         ELSE
36             we <= '0';
37             oe <= '0';
38         END IF;
39     END PROCESS;
40
41 END io_intf_architecture ;
```

7.4 Instruction Register

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.std_logic_unsigned.ALL;
4
5  ENTITY instr_reg IS
6      PORT(
7                      state_Sig : IN std_logic_vector (1 DOWNTO 0);
8                      clk : IN std_logic ;
9                      buswire: IN std_logic_vector (15 DOWNTO 0);
10                     ir_addr : OUT std_logic_vector(7 DOWNTO 0);
11                     ir_ev : OUT std_logic;
12                     add_sig : OUT std_logic;
13                     store_sig : OUT std_logic;
14                     load_sig : OUT std_logic;
15                     jump_sig: OUT std_logic;
16                     jneg_sig : OUT std_logic
17         );
18 END instr_reg;
19
20 ARCHITECTURE instr_reg_architecture OF instr_reg IS
21 SIGNAL op: std_logic_vector(7 DOWNTO 0);
22 SIGNAL iaddr: std_logic_vector (7 DOWNTO 0);
23 SIGNAL ev: std_logic;
24 SIGNAL decode_done: std_logic := '0';
25 BEGIN
26
27     --output to MIF
28     ir_addr <= iaddr;
29     ir_ev <= decode_done;
30
31     PROCESS(clk)
```

```
32      BEGIN
33          IF  rising_edge (clk) THEN
34              IF  state_Sig  = "00" THEN  --reset_s
35                  ev <= '0';
36                  iaddr <= (OTHERS=>'0');
37                  op <= (OTHERS=>'0');
38              ELSIF state_Sig  = "01" THEN  --fetch_s
39                  ev <= '0';
40              ELSIF state_Sig  = "10" THEN  --execute_s
41                  ev <= '1';
42                  iaddr <= buswire(7 DOWNTO 0);  --get address
43                  op <= buswire(15 DOWNTO 8);  --get op code
44              END IF;
45          END IF;
46      END PROCESS;
47
48      PROCESS(ev)  --decode process
49      BEGIN
50          add_sig <= '0';  --reset output signal
51          store_sig  <= '0';
52          load_sig  <= '0';
53          jump_sig <= '0';
54          jneg_sig  <= '0';
55          decode_done <= '0';
56          IF ev='1' THEN
57              decode_done <= '1';
58              CASE op IS
59                  WHEN "00000000" =>  --ADD
60                      add_sig  <= '1';
61                  WHEN "00000001" =>  --STORE
62                       store_sig  <= '1';
63                  WHEN "00000010" =>  --LOAD
64                       load_sig  <= '1';
65                  WHEN "00000011" =>  --JUMP
66                      jump_sig <= '1';
67                  WHEN "00000100" =>  --JNEG
68                       jneg_sig  <= '1';
69                  WHEN OTHERS =>
70                      NULL;
71              END CASE;
72          END IF;
73      END PROCESS;
74  END  instr_reg_architecture ;
```

7.5 Accumulator register

```
1   LIBRARY ieee;
2   USE ieee. std_logic_1164 .ALL;
3   USE ieee. std_logic_unsigned .ALL;
4
5   ENTITY acc IS
6       PORT(
7                       alu_reg : IN  std_logic_vector (15 DOWNTO 0);
8                       add_done: IN  std_logic ;
9                       reset_sig : IN  std_logic ;
10                      clk : IN  std_logic ;
```

```
                    store_sig : IN  std_logic ;
                    load_sig : IN  std_logic ;
                    acc_reg: OUT std_logic_vector(15 DOWNTO 0);
                    buswire: INOUT std_logic_vector(15 DOWNTO 0)
        );
END acc;

ARCHITECTURE acc_architecture OF acc IS
SIGNAL ireg: std_logic_vector (15 DOWNTO 0);
BEGIN

    acc_reg <= ireg;  --output to ALU
    buswire <= ireg WHEN store_sig='1' ELSE (OTHERS=>'Z');  --write to buswire

    PROCESS(clk,reset_sig)
    BEGIN
        IF reset_sig ='1' THEN
            ireg <= "0000000000000000";
        ELSIF rising_edge(clk) THEN
            IF load_sig ='1' THEN
                ireg <= buswire;  --get content at memory address
            ELSIF add_done='1' THEN
                ireg <= alu_reg;  --refresh when ADD complete
            END IF;
        END IF;
    END PROCESS;

END acc_architecture;
```

7.6 Arithmetic Logic Unit

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY alu IS
    PORT(
                    clk: IN  std_logic ;
                    jneg_sig : IN  std_logic ;
                    buswire: IN  std_logic_vector (15 DOWNTO 0);
                    add_sig: IN  std_logic ;
                    pc_jneg: OUT std_logic;
                    acc_reg: IN  std_logic_vector (15 DOWNTO 0);
                    alu_reg: OUT std_logic_vector(15 DOWNTO 0);
                    add_done: OUT std_logic
        );
END alu;

ARCHITECTURE alu_architecture OF alu IS
SIGNAL ireg_A: std_logic_vector (15 DOWNTO 0);  --from AC
BEGIN

    pc_jneg <= '1' WHEN jneg_sig='1' AND conv_integer(alu_reg) < 0 ELSE '0';
    PROCESS(clk)

    BEGIN
```

```
26          IF  rising_edge (clk)  THEN
27              IF  add_sig='1' THEN  --calculation
28                  alu_reg  <= acc_reg + buswire;
29                  add_done <= '1';
30              ELSE
31                  ireg_A  <= acc_reg;  --input from AC
32                  alu_reg  <= (OTHERS=>'0');
33                  add_done <= '0';
34              END IF;
35          END IF;
36      END PROCESS;
37
38  END alu_architecture;
```

7.7 Memory

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3
4   PACKAGE MEM256 IS
5       COMPONENT mem IS
6           PORT(
7                           reset_sig : IN  std_logic ;
8                           we: IN  std_logic ;   --write enable
9                           address: IN  std_logic_vector (7 DOWNTO 0);
10                          oe: IN  std_logic ;   --output enable
11                          clk: IN  std_logic ;
12                          data_io: INOUT std_logic_vector(15 DOWNTO 0)
13                          );
14      END COMPONENT;
15  END PACKAGE MEM256;
16
17
18  LIBRARY ieee;
19  USE ieee.std_logic_1164.ALL;
20  USE ieee.std_logic_unsigned.ALL;
21
22  ENTITY mem IS
23      PORT(
24                          reset_sig : IN  std_logic ;
25                          we: IN  std_logic ;   --write enable
26                          address: IN  std_logic_vector (7 DOWNTO 0);
27                          oe: IN  std_logic ;   --output enable
28                          clk: IN  std_logic ;
29                          data_io: INOUT std_logic_vector(15 DOWNTO 0)
30          );
31  END mem;
32
33  ARCHITECTURE mem_architecture OF mem IS
34  TYPE MEMORY IS ARRAY(255 DOWNTO 0) OF std_logic_vector(15 DOWNTO 0);
35  SIGNAL mem_array: MEMORY:=(OTHERS => (OTHERS => '0'));
36  SIGNAL data_in:  std_logic_vector (15 DOWNTO 0);
37  SIGNAL data_out:  std_logic_vector (15 DOWNTO 0);
38  BEGIN
39
40      data_in  <= data_io;
```

```
41      data_io <= data_out WHEN oe='0' ELSE (OTHERS => 'Z');
42      --internal memory structure
43      data_out <= mem_array(conv_integer(address)) WHEN we='0' ELSE (OTHERS => 'Z');
44
45      PROCESS(clk,we,reset_sig) --writing process
46      BEGIN
47          IF reset_sig = '1' THEN
48              --preset value
49              mem_array(0) <= "0000001011111110"; --02FE:LOAD 254
50              mem_array(1) <= "0000010000000100"; --0403:JNEG 04
51              mem_array(2) <= "0000001011111101"; --0304:LOAD 253
52              mem_array(3) <= "0000001100000011"; --0303:JUMP 03
53              mem_array(253) <= "0000000000000100"; --0004
54              mem_array(254) <= "0000000000000011"; --0003
55          ELSIF rising_edge(clk) THEN
56              IF we='1' THEN
57                  mem_array(conv_integer(address)) <= data_in;
58              END IF;
59          END IF;
60      END PROCESS;
61
62  END mem_architecture;
63
64
65  LIBRARY ieee;
66  USE ieee.std_logic_1164.ALL;
67  USE work.MEM256.ALL;
68
69  ENTITY my_mem256 IS
70      PORT(
71                      reset_sig : IN std_logic ;
72                      we: IN std_logic ;  --write enable
73                      address: IN std_logic_vector (7 DOWNTO 0);
74                      oe: IN std_logic ;  --output enable
75                      clk: IN std_logic ;
76                      data_in: IN std_logic_vector (15 DOWNTO 0);
77                      data_out: OUT std_logic_vector(15 DOWNTO 0)
78          );
79  END my_mem256;
80
81  ARCHITECTURE my_mem256_architecture OF my_mem256 IS
82  SIGNAL bidirectional_buffer : std_logic_vector (15 downto 0);
83  BEGIN
84
85      data_out <= bidirectional_buffer ;
86      bidirectional_buffer <= data_in WHEN oe='1' ELSE (OTHERS => 'Z');
87      todo: mem port map (reset_sig,we,address,oe,clk, bidirectional_buffer );
88
89  END my_mem256_architecture;
```

7.8 Clock Frequency Divider

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.std_logic_unsigned.ALL;
4
```

```
5  ENTITY freq_dvider IS
6      PORT(
7                clock : IN  std_logic ;
8                reset : IN  std_logic ;
9                clk : OUT std_logic
10           );
11 END freq_dvider;
12
13 ARCHITECTURE freq_dvider_architecture OF freq_dvider IS
14 SIGNAL reg: INTEGER RANGE 0 TO 24999999;
15 SIGNAL i_clk:  std_logic ;
16 BEGIN
17
18     PROCESS(reset,clock)
19     BEGIN
20         IF  reset='0' THEN
21             reg <= 0;
22             i_clk <= '0';
23         ELSIF rising_edge(clock) THEN
24             IF reg=24999999 THEN
25                 i_clk <= not i_clk;
26                 reg <= 0;
27             ELSE
28                 reg <= reg + 1;
29             END IF;
30         END IF;
31     END PROCESS;
32
33     clk <= i_clk;
34
35 END  freq_dvider_architecture ;
```

7.9 Hex Display

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.std_logic_unsigned.ALL;
4
5  ENTITY hex_disp IS
6      PORT(
7                    acc_reg : IN  std_logic_vector (15 DOWNTO 0);
8                    display0 : OUT std_logic_vector(6 DOWNTO 0);
9                    display1 : OUT std_logic_vector(6 DOWNTO 0);
10                   display2 : OUT std_logic_vector(6 DOWNTO 0);
11                   display3 : OUT std_logic_vector(6 DOWNTO 0)
12           );
13 END hex_disp;
14
15 ARCHITECTURE hex_disp_architecture OF hex_disp IS
16 SIGNAL ori0:  std_logic_vector (3 DOWNTO 0);
17 SIGNAL ori1:  std_logic_vector (3 DOWNTO 0);
18 SIGNAL ori2:  std_logic_vector (3 DOWNTO 0);
19 SIGNAL ori3:  std_logic_vector (3 DOWNTO 0);
20 BEGIN
21
22     ori0 <= acc_reg(3 DOWNTO 0);
```

```
23      ori1 <= acc_reg(7 DOWNTO 4);
24      ori2 <= acc_reg(11 DOWNTO 8);
25      ori3 <= acc_reg(15 DOWNTO 12);
26      PROCESS(ori0,ori1,ori2, ori3 )
27      BEGIN
28          CASE ori0 IS
29              WHEN "0000" => display0 <= "0000001"; --0:01
30              WHEN "0001" => display0 <= "1001111"; --1:4F
31              WHEN "0010" => display0 <= "0010010"; --2:12
32              WHEN "0011" => display0 <= "0000110"; --3:06
33              WHEN "0100" => display0 <= "1001100"; --4:4C
34              WHEN "0101" => display0 <= "0100100"; --5:24
35              WHEN "0110" => display0 <= "0100000"; --6:20
36              WHEN "0111" => display0 <= "0001111"; --7:0F
37              WHEN "1000" => display0 <= "0000000"; --8:00
38              WHEN "1001" => display0 <= "0000100"; --9:04
39              WHEN "1010" => display0 <= "0001000"; --A:08
40              WHEN "1011" => display0 <= "1100000"; --B:60
41              WHEN "1100" => display0 <= "0110001"; --C:31
42              WHEN "1101" => display0 <= "1000010"; --D:42
43              WHEN "1110" => display0 <= "0110000"; --E:30
44              WHEN "1111" => display0 <= "0111000"; --F:38
45              WHEN OTHERS => display0 <= "0000000";
46          END CASE;
47
48          CASE ori1 IS
49              WHEN "0000" => display1 <= "0000001"; --0:01
50              WHEN "0001" => display1 <= "1001111"; --1:4F
51              WHEN "0010" => display1 <= "0010010"; --2:12
52              WHEN "0011" => display1 <= "0000110"; --3:06
53              WHEN "0100" => display1 <= "1001100"; --4:4C
54              WHEN "0101" => display1 <= "0100100"; --5:24
55              WHEN "0110" => display1 <= "0100000"; --6:20
56              WHEN "0111" => display1 <= "0001111"; --7:0F
57              WHEN "1000" => display1 <= "0000000"; --8:00
58              WHEN "1001" => display1 <= "0000100"; --9:04
59              WHEN "1010" => display1 <= "0001000"; --A:08
60              WHEN "1011" => display1 <= "1100000"; --B:60
61              WHEN "1100" => display1 <= "0110001"; --C:31
62              WHEN "1101" => display1 <= "1000010"; --D:42
63              WHEN "1110" => display1 <= "0110000"; --E:30
64              WHEN "1111" => display1 <= "0111000"; --F:38
65              WHEN OTHERS => display1 <= "0000000";
66          END CASE;
67
68          CASE ori2 IS
69              WHEN "0000" => display2 <= "0000001"; --0:01
70              WHEN "0001" => display2 <= "1001111"; --1:4F
71              WHEN "0010" => display2 <= "0010010"; --2:12
72              WHEN "0011" => display2 <= "0000110"; --3:06
73              WHEN "0100" => display2 <= "1001100"; --4:4C
74              WHEN "0101" => display2 <= "0100100"; --5:24
75              WHEN "0110" => display2 <= "0100000"; --6:20
76              WHEN "0111" => display2 <= "0001111"; --7:0F
77              WHEN "1000" => display2 <= "0000000"; --8:00
78              WHEN "1001" => display2 <= "0000100"; --9:04
```

```
79              WHEN "1010" => display2 <= "0001000"; --A:08
80              WHEN "1011" => display2 <= "1100000"; --B:60
81              WHEN "1100" => display2 <= "0110001"; --C:31
82              WHEN "1101" => display2 <= "1000010"; --D:42
83              WHEN "1110" => display2 <= "0110000"; --E:30
84              WHEN "1111" => display2 <= "0111000"; --F:38
85              WHEN OTHERS => display2 <= "0000000";
86          END CASE;
87
88          CASE ori3 IS
89              WHEN "0000" => display3 <= "0000001"; --0:01
90              WHEN "0001" => display3 <= "1001111"; --1:4F
91              WHEN "0010" => display3 <= "0010010"; --2:12
92              WHEN "0011" => display3 <= "0000110"; --3:06
93              WHEN "0100" => display3 <= "1001100"; --4:4C
94              WHEN "0101" => display3 <= "0100100"; --5:24
95              WHEN "0110" => display3 <= "0100000"; --6:20
96              WHEN "0111" => display3 <= "0001111"; --7:0F
97              WHEN "1000" => display3 <= "0000000"; --8:00
98              WHEN "1001" => display3 <= "0000100"; --9:04
99              WHEN "1010" => display3 <= "0001000"; --A:08
100             WHEN "1011" => display3 <= "1100000"; --B:60
101             WHEN "1100" => display3 <= "0110001"; --C:31
102             WHEN "1101" => display3 <= "1000010"; --D:42
103             WHEN "1110" => display3 <= "0110000"; --E:30
104             WHEN "1111" => display3 <= "0111000"; --F:38
105             WHEN OTHERS => display3 <= "0000000";
106         END CASE;
107     END PROCESS;
108
109 END hex_disp_architecture;
```