Course Number	COE328
Course Title	Digital Systems
Semester/Year	Fall 2024
Instructor	Dr. Arghavan Asad
TA Name	Mohammad Hossein Saliminabi

Lab No.	6
Lab Title	Design of a Simple Central Processing Unit

Section No.	18
Submission Date	December 12th, 2024
Due Date	December 12th, 2024

Student Name	Student ID	Signature
Hamzah Sahi	501221955	H.S

Table of Contents

Introduction:

The purpose of this lab was to design and implement an 8-bit central processing unit (CPU) with various components. The CPU architecture was created by combining both combinational and sequential circuits. An arithmetic logic unit (ALU), two latches, a 4:16 decoder and a finite state machine (FSM) were created and used. The ALU handled basic math and logic operations such as AND, OR, NOT. The latches were used to temporarily store memory. The 4:16 decoder helped select specific operations or memory locations. The FSM ensured that all parts of the CPU were working in the correct and specified order. By combining these components we were able to create an 8-bit CPU which then displayed the results via a 7-segment display and simulated waveforms.

Components Descriptions:

Table 1.0 - Component Descriptions

Component	Description
Latch	The latch works as an essential storage unit in the CPU and it facilitates the role of temporarily storing an 8-bit input value before passing it to the ALU component. In this system, there are two latches, each storing one of the 8-bit inputs (A and B) of the ALU.
FSM (Finite State Machine)	The FSM within this system operates as a moore machine and it is a component of the control unit. The FSM in this system is designed as an up counter with 8 states. The machine cycles through the states 0-7 when it is activated during the rising edge of the clock signal, and when data_in equals '1', before returning back to state 0. A 4-bit output representing the current state of the FSM counter signal is then passed on to the 4:16 decoder. Furthermore, the FSM also outputs a 4-bit signal representing the Student ID to the Seven Segment display as it transitions between the states.
4:16 Decoder	The 4:16 decoder takes in a 4-bit input from the FSM and generates 16 bit microcode output. This microcode corresponds with the state defined by the FSM and it determines what function the ALU will perform. This microcode allows the user to have control over the ALU's behaviour, and it coordinates the overall operation of the CPU.
ALU	The arithmetic logic unit (ALU) is the part of the CPU that handles all the math and logic work. It can perform basic arithmetic like simply adding

	and subtracting, as well as logical operations such as AND, OR, and XOR. The ALU also handles shifting bits, which can be used for certain calculations, and can compare two numbers to check if they're equal in size or whether one is bigger or not. The ALU receives its input from the CPU's registers, executes instructions it receives from the control unit and then produces an output which can be stored or used by other components.
Seven Segment Display	The seven segment display is a visual output device that can represent numerical values as well as characters. It contains seven individual segments that are arranged in a figure-eight pattern, each of which can be lit independently to display numbers from 0 to 9, or even certain letters. The segments themself are controlled by input signals, which determine which segments are illuminated to form the desired output. In this lab, the display was used to illustrate the results of the CPU's operations.

*The student ID that was utilized for this lab was 501167248. Making A = (72) and B = (48) which when converted to binary are: A = (0111 0010), B = (0100 1000).

Component BDF and Waveforms:

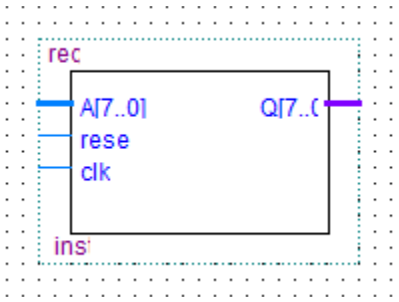
Register 1 & 2:

Table 2.0 - Register Truth Table

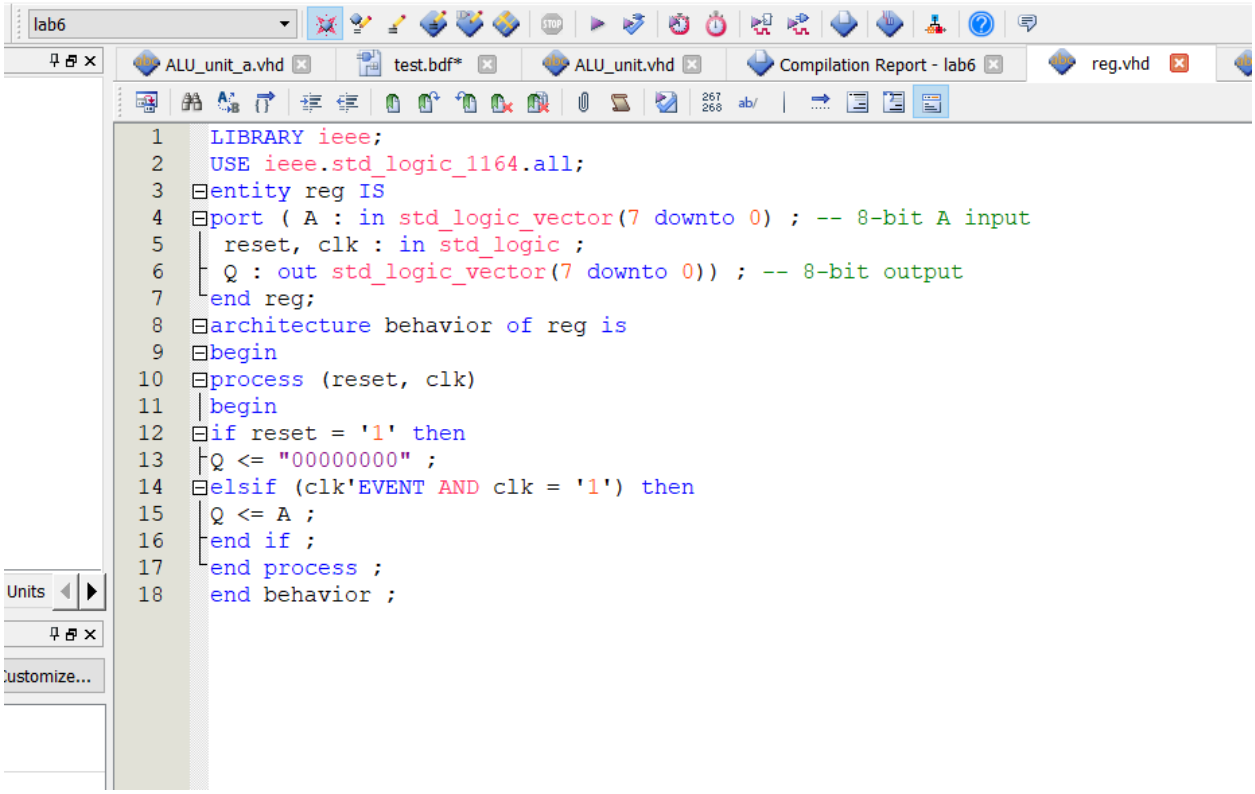
Input			Output
Reset	Clock	A[7..0]	Q(t+1)
0	0	A[7..0]	00000000
0	1	A[7..0]	00000000

1	0	A[7..0]	Q(t)
1	1	A[7..0]	A[7..0]

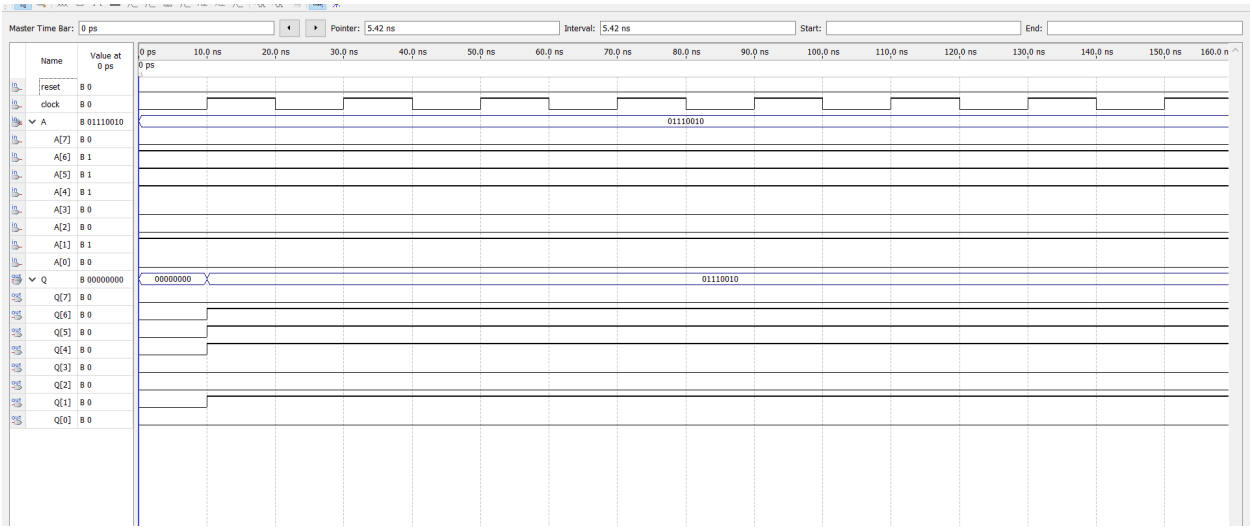
Register Block Diagram



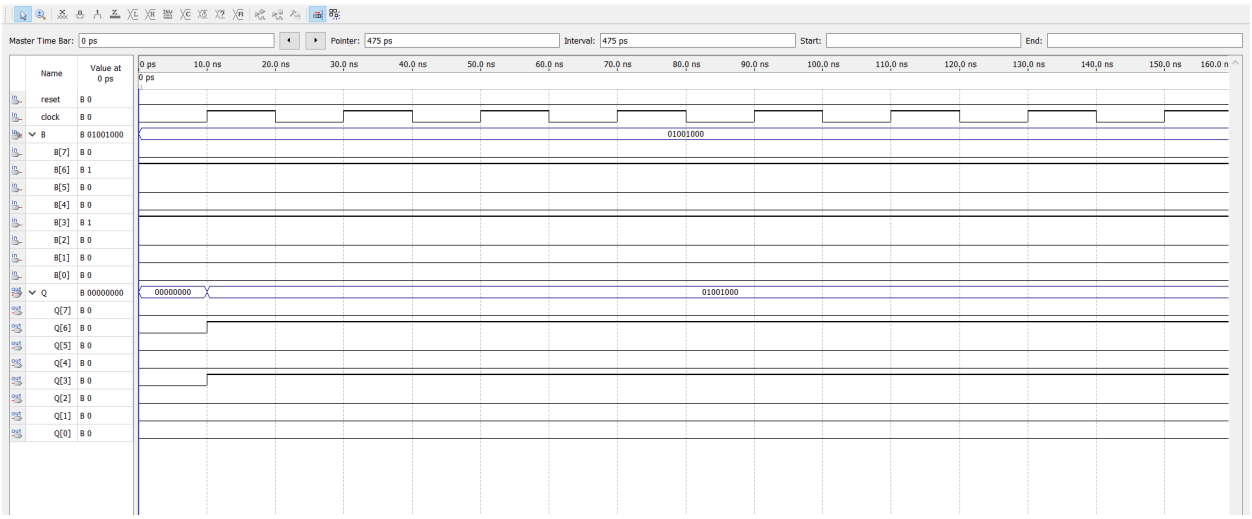
Register VHDL code



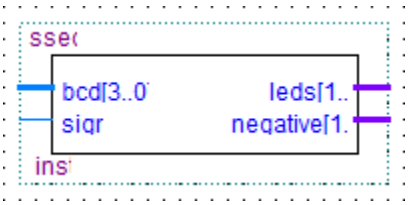
Register A waveform



Register B waveform

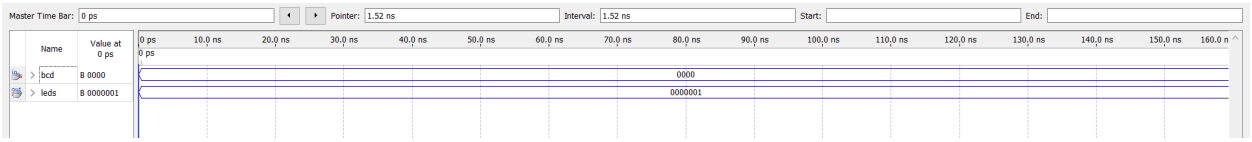


SSEG



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sseg IS
PORT (
bcd : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
leds : OUT STD_LOGIC_VECTOR(1 TO 7) ;
negative : OUT STD_LOGIC_VECTOR(1 TO 7);
sign : IN STD_LOGIC );
END sseg ;

ARCHITECTURE Behavior OF sseg IS
BEGIN
PROCESS ( bcd, sign )
BEGIN
IF (sign = '0') THEN
negative <= NOT "1111110";
ELSIF (sign = '1') THEN
negative <= NOT "0000001";
END IF;

CASE bcd IS
WHEN "0000" => leds <= NOT "1111110"; --0
WHEN "0001" => leds <= NOT "0110000"; --1
WHEN "0010" => leds <= NOT "1101101"; --2
WHEN "0011" => leds <= NOT "1111001"; --3
WHEN "0100" => leds <= NOT "0110011"; --4
WHEN "0101" => leds <= NOT "1011011"; --5
WHEN "0110" => leds <= NOT "1011111"; --6
WHEN "0111" => leds <= NOT "1110000"; --7
WHEN "1000" => leds <= NOT "1111111"; --8
WHEN "1001" => leds <= NOT "1110011"; --9
WHEN "1010" => leds <= "1110111";
WHEN "1011" => leds <= "0011111";
WHEN "1100" => leds <= "1001110";
WHEN "1101" => leds <= "0111101";
WHEN "1110" => leds <= "1001111";
```

```
WHEN "1101" => leds <= "0111101";
WHEN "1110" => leds <= "1001111";
WHEN "1111" => leds <= "1000111";
WHEN OTHERS => leds <= "-------";
END CASE ;

END PROCESS ;

END Behavior ;
```



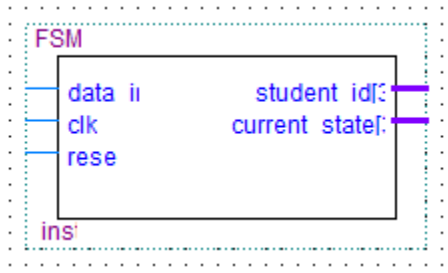
FSM:

Table 3.0 - FSM State Table

Current state	Next State		output
	data_in = 0	data_in = 1	
s0	s0	s1	d2 = 0000
s1	s1	s2	d3 = 0001
s2	s2	s3	d4 = 0010
s3	s3	s4	d5 = 0100
s4	s4	s5	d6 = 0011
s5	s5	s6	d7 = 1001
s6	s6	s7	d8 = 0010

s7	s7	s0	d9 = 1001

FSM Block Diagram

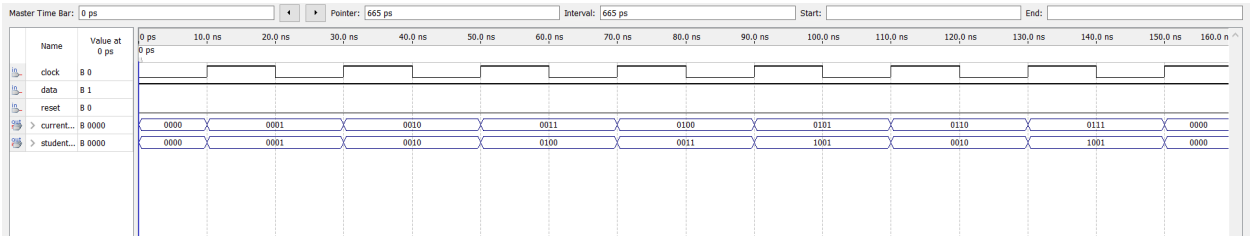


FSM VHDL Code

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4   entity FSM IS
5   port (data_in, clk, reset : in std_logic ;
6   student_id : out std_logic_vector(3 downto 0);
7   current_state : out std_logic_vector(3 downto 0));
8
9   end FSM ;
10
11  architecture fsm of FSM is
12  type state_type is (s0, s1, s2, s3, s4, s5, s6, s7);
13  signal yfsm : state_type ;
14  begin
15  process ( clk, reset )
16  begin
17  if reset = '1' then
18  yfsm <= s0 ;
19  elsif (clk'EVENT AND clk = '1') then
20  case yfsm is
21  when s0 =>
22  if data_in = '1' then
23  yfsm <= s1 ;
24  end if ;
25  when s1 =>
26  if data_in = '1' then
27  yfsm <= s2 ;
28  end if ;
29  when s2 =>
30  if data_in = '1' then
31  yfsm <= s3 ;
32  end if ;
33  when s3 =>
34  if data_in = '1' then
35  yfsm <= s4 ;
36  end if ;
37  when s4 =>
```

```
37    when s4 =>
38  if data_in = '1' then
39    yfsm <= s5 ;
40  end if ;
41    when s5 =>
42  if data_in = '1' then
43    yfsm <= s6 ;
44  end if ;
45    when s6 =>
46  if data_in = '1' then
47    yfsm <= s7 ;
48  end if ;
49    when s7 =>
50  if data_in = '1' then
51    yfsm <= s0 ;
52  end if ;
53  end case ;
54  end if ;
55  end process ;
56  process ( yfsm )
57    begin
58  case yfsm is
59    when s0 => current_state <= "0000" ;
60    student_id <= "0000" ; -- d2 0
61    when s1 => current_state <= "0001" ;
62    student_id <= "0001" ; -- d3 1
63    when s2 => current_state <= "0010" ;
64    student_id <= "0010" ; -- d4 2
65    when s3 => current_state <= "0011" ;
66    student_id <= "0100" ; -- d5 4
67    when s4 => current_state <= "0100" ;
68    student_id <= "0011" ; -- d6 3
69    when s5 => current_state <= "0101" ;
70    student_id <= "1001" ; -- d7 9
71    when s6 => current_state <= "0110" ;
72    student_id <= "0010" ; -- d8 2
```

```
72    student_id <= "0010" ; -- d8 2
73    when s7 => current_state <= "0111" ;
74    student_id <= "1001" ; -- d9 6 St. ID
75
76      end case ; -- d1 d2 d3 d4 d5 d6 d7 d8 d9
77   end process ; -- 5 0 1 2 4 3 9 2 6
78   end fsm ; -- states: s0 s1 s2 s3 s4 s5 s6 s7
```

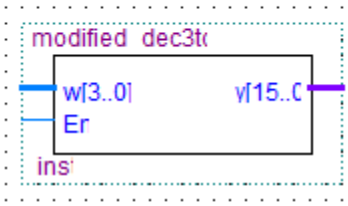


4:16 Decoder

Table 4.0 - 4:16 Decoder truth table

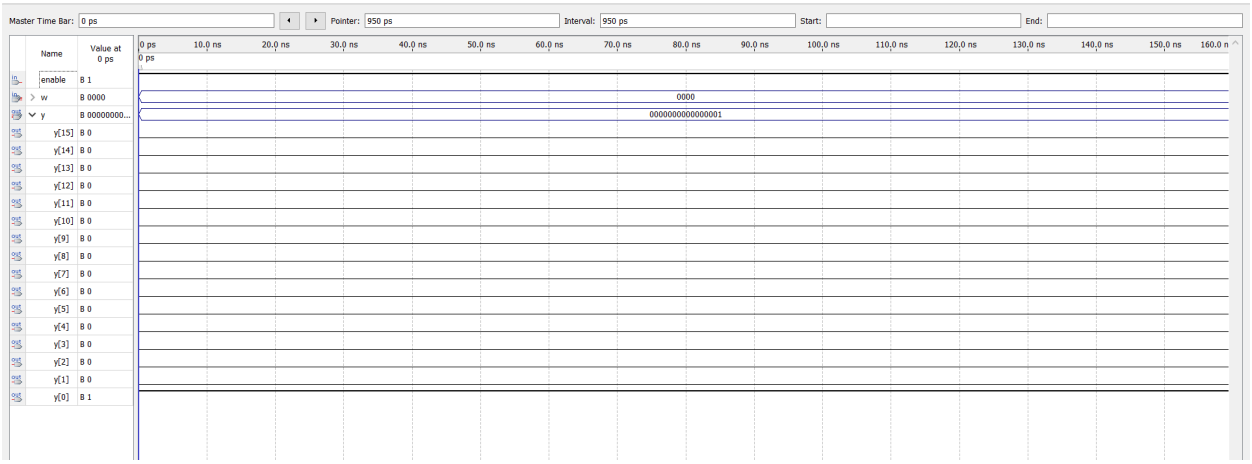
En	w_3	w_2	w_1	w_0	output
1	0	0	0	0	opcode[0]
1	0	0	0	1	opcode[1]
1	0	0	1	0	opcode[2]
1	0	0	1	1	opcode[3]
1	0	1	0	0	opcode[4]
1	0	1	0	1	opcode[5]
1	0	1	1	0	opcode[6]
1	0	1	1	1	opcode[7]
1	1	0	0	0	opcode[8]
1	1	0	0	1	opcode[9]
1	1	0	1	0	opcode[10]
1	1	0	1	1	opcode[11]
1	1	1	0	0	opcode[12]
1	1	1	0	1	opcode[13]
1	1	1	1	0	opcode[14]
1	1	1	1	1	opcode[15]

4:16 Decoder Block Diagram



4:16 Decoder VHDL Code

```
1  LIBRARY ieee ;
2  USE ieee.STD_LOGIC_1164.all ;
3  ENTITY modified_dec3to8 IS
4  PORT ( w : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5  En : IN STD_LOGIC ;
6  y : OUT STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
7  END modified_dec3to8 ;
8  ARCHITECTURE Behavior OF modified_dec3to8 IS
9  SIGNAL Enw : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
10 BEGIN
11 Enw <=
12  En & w ;
13 WITH Enw SELECT
14 y <=
15 "0000000000000001" WHEN "10000",
16 "0000000000000010" WHEN "10001",
17 "0000000000000100" WHEN "10010",
18 "0000000000001000" WHEN "10011",
19 "0000000000010000" WHEN "10100",
20 "0000000000100000" WHEN "10101",
21 "0000000001000000" WHEN "10110",
22 "0000000010000000" WHEN "10111",
23 "0000000000000000" WHEN OTHERS ;
24 END Behavior ;
```



Problem 1:

ALU_1 For Problem Set 1:

ALU_1 VHDL code

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.all;

entity ALU_unit_a is -- ALU unit includes Reg. 3
  port (
    clk, res : in std_logic;
    Reg1, Reg2 : in std_logic_vector(7 downto 0); -- 8-bit inputs A & B from Reg. 1 & Reg. 2
    opcode : in std_logic_vector(15 downto 0); -- 16-bit opcode from Decoder
    Result : buffer std_logic_vector(7 downto 0);  -- 8-bit Result
    neg : out std_logic;
    res1: out std_logic_vector (3 downto 0);
    res2: out std_logic_vector (3 downto 0)

  );
end ALU_unit_a;

architecture calculation of ALU_unit_a is
begin
  process (clk, res)
  begin
    if res = '1' then
      Result <= "00000000";
    elsif (clk'EVENT AND clk = '1') then
      case opcode is
        when "0000000000000001" =>
          Result <= Reg1 + Reg2;-- Function 1 addition Reg1 and Reg2
        when "0000000000000010" =>
          Result <= Reg1 - Reg2;-- Function 2 subtraction Reg1 and Reg2
        when "0000000000000100" =>
          Result <= not Reg1;-- Function 3 inverse Reg1
        when "0000000000001000" =>
          Result <= not (Reg1 and Reg2);-- Function 4 Boolean NAND Reg1 and Reg2
        when "0000000000010000" =>
          Result <= not (Reg1 or Reg2);-- Function 5 Boolean NOR Reg1 and Reg2
        when "0000000000100000" =>
        when "0000000000100000" =>
          Result <= Reg1 and Reg2;-- Function 6 Boolean AND Reg1 and Reg2
        when "0000000001000000" =>
          Result <= Reg1 xor Reg2;-- Function 7 Boolean XOR Reg1 and Reg2
        when "0000000010000000" =>
          Result <= Reg1 or Reg2;-- Function 8 Boolean OR Reg1 and Reg2
        when others =>
          Result <= "00000000";-- Don't care, do nothing
      end case;
      if Result(7) = '1' then
           neg <= '1';
        else
           neg <= '0';
      end if;
    end if;
  end process;
   res1 <= result(7 downto 4);
   res2 <= result(3 downto 0);
end calculation;
```

ALU_1 Block Diagram

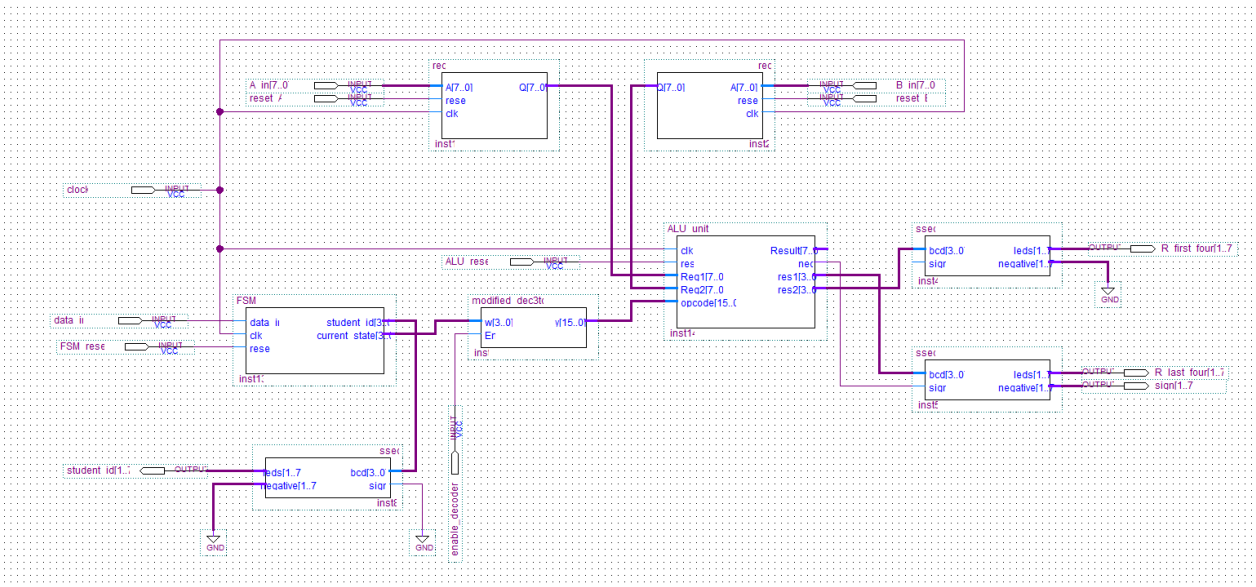
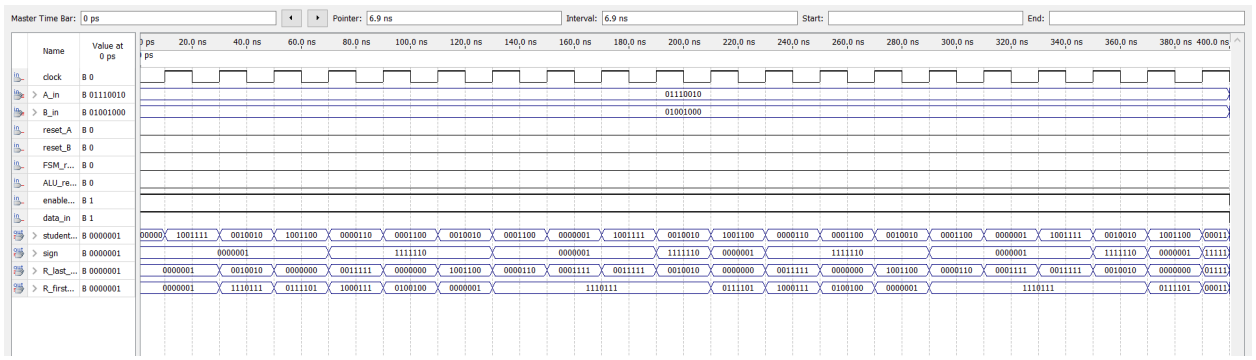


Table 5.0 - Microcode Table

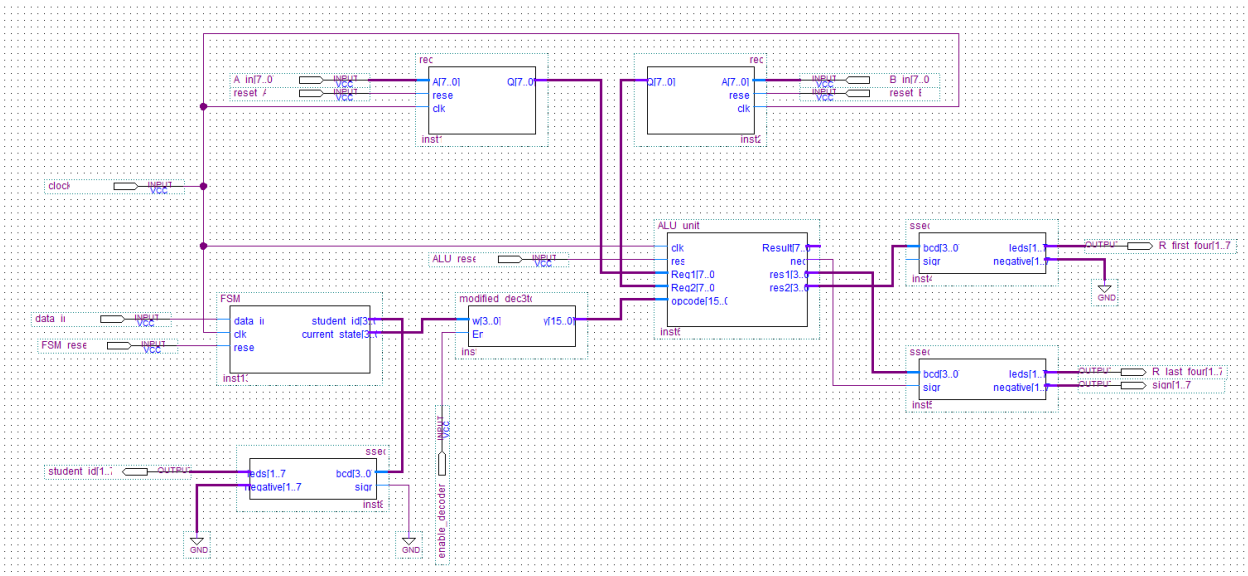
Function #	Microcode	Function
1	0000000000000001	*sum*(***A***, ***B***)
2	0000000000000010	*dif*(***A***, ***B***)
3	0000000000000100	$\overline{\boldsymbol{A}}$
4	0000000000001000	$\overline{\boldsymbol{A} \cdot \boldsymbol{B}}$
5	0000000000010000	$\overline{\boldsymbol{A} + \boldsymbol{B}}$
6	0000000000100000	$\boldsymbol{A} \cdot \boldsymbol{B}$
7	0000000001000000	$\boldsymbol{A} \oplus \boldsymbol{B}$
8	0000000010000000	$\boldsymbol{A} + \boldsymbol{B}$



The purpose of the ALU is to perform a set of arithmetic and logical operations based on an 8-bit opcode input. It takes two 8-bit inputs, A and B, and produces an 8-bit output result. The functions implemented in this ALU are written above in table 5.0 and include operations such as addition, subtraction, logical NOT on the input A, and more, for a total of 8 functions. The inputs to the ALU includes the clock (clk) which keeps all the components synchronized, the reset signal (res) that clears the output result when activated, and the 8-bit operands A and B. The opcode input selects the specific operation that is done on the inputs A and B. The outputs of the ALu include an 8-bit result that will show the output of the operations. It also includes a negative (neg) that will indicate whether or not the result is negative. The results from the ALU are split into two 4-bit outputs (res 1 and res 2) which are each connected to a 4-bit seven segment display that when put together illustrate the 8-bit results.

Problem 2

ALU_2 For Problem Set 2



b)

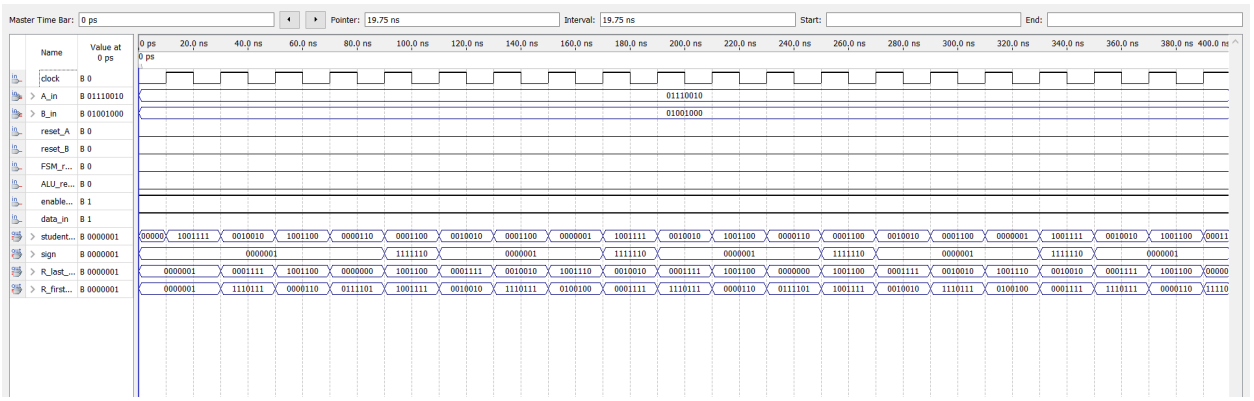
Function #	Operation / Function
1	Swap the lower and upper 4 bits of **A**
2	Produce the result of ORing **A** and **B**
3	Decrement **B** by 5
4	Invert all bits of **A**
5	Invert the bit-significance order of **A**
6	Find the greater value of **A** and **B** and produce the results (Max(**A**,**B**))
7	Produce the difference between **A** and **B**
8	Produce the result of XNORing **A** and **B**

Microcode Table

Function #	Microcode	Function
1	0000000000000001	Swap lower and upper 4 bits of ***A***
2	0000000000000010	***A*** + ***B***
3	0000000000000100	Decrement ***B*** by 5
4	0000000000001000	$\overline{\boldsymbol{A}}$
5	0000000000010000	Invert bit-significance order of ***A***
6	0000000000100000	Output the greater value of ***A*** and ***B*** as the result
7	0000000001000000	*dif*(***A***, ***B***)
8	0000000010000000	$\overline{\boldsymbol{A} \oplus \boldsymbol{B}}$

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;
4  USE ieee.numeric_std.all;
5
6  entity ALU_unit_b is -- ALU unit includes Reg. 3
7    port (
8      clk, res : in std_logic;
9      Reg1, Reg2 : in std_logic_vector(7 downto 0); -- 8-bit inputs A & B from Reg. 1 & Reg. 2
10     opcode : in std_logic_vector(15 downto 0); -- 16-bit opcode from Decoder
11     Result : buffer std_logic_vector(7 downto 0);  -- 8-bit Result
12     neg : out std_logic;
13     res1: out std_logic_vector (3 downto 0);
14     res2: out std_logic_vector (3 downto 0)
15   );
16 end ALU_unit_b;
17
18 architecture calculation of ALU_unit_b is
19 begin
20   process (clk, res)
21   begin
22     if res = '1' then
23       Result <= "00000000";
24     elsif (clk'EVENT AND clk = '1') then
25       case opcode is
26         when "0000000000000001" =>
27           -- Function 1 Swap the lower and upper 4 bits of Reg1
28           Result <= Reg1(3 downto 0) & Reg1(7 downto 4);
29         when "0000000000000010" =>
30           -- Function 2 Or of Reg1 and Reg2
31           Result <= Reg1 or Reg2;
32         when "0000000000000100" =>
33           -- Function 3 Decrement Reg2 by 5
34           Result <= std_logic_vector(signed(Reg2) - 5);
35         when "0000000000001000" =>
36           -- Function 4 Invert bits of Reg1
37           Result <= not Reg1;
```

```
        Result <= not Reg1;
      when "0000000000010000" =>
        -- Function 5 Invert the bit-significance order of Reg1
        Result <= Reg1(0) & Reg1(1) & Reg1(2) & Reg1(3) & Reg1(4) & Reg1(5) & Reg1(6) & Reg1(7);
      when "0000000000100000" =>
        -- Function 6 greater value of Reg1 and Reg2 (Max(Reg1, Reg2))
        if unsigned(Reg1) > unsigned(Reg2) then
          Result <= Reg1;
        else
          Result <= Reg2;
        end if;
      when "0000000001000000" =>
        -- Function 7 difference between Reg1 and Reg2
        Result <= std_logic_vector(signed(Reg1) - signed(Reg2));
      when "0000000010000000" =>
        -- Function 8 XNOR of Reg1 and Reg2
        Result <= not (Reg1 xor Reg2);
      when others =>
        -- Default case: Do nothing
        Result <= "00000000";
    end case;

    -- Check if the result is negative
    if Result(7) = '1' then
      neg <= '1';
    else
      neg <= '0';
    end if;
  end if;
end process;

res1 <= Result(7 downto 4);
res2 <= Result(3 downto 0);
end calculation;
```



The ALU_2 component is used to do more difficult arithmetic and logical operations. Based on an 8-bit opcode, it performs operations including bit inversion, addition, switching the lower and upper four bits of an input, and comparing which of two values is bigger. For display on seven-segment displays, the component splits the 8-bit result it creates from two 8-bit operands (A and B) into two 4-bit outputs (Res1 and Res2). A negative flag (neg) is one of the other outputs that shows whether the result is negative or not. Other inputs include a reset signal (res) to clear outputs when needed and a clock signal (clk) for synchronization.

Problem 3

a) For each opcode submitted to the ALU, display 'y' if the **student_id** signal value is odd and 'n' otherwise

I didn't have access to quartus for this.

Table 5.0 - Microcode Table

Function #	Microcode	Function
1	0000000000000001	$sum(\boldsymbol{A}, \boldsymbol{B})$
2	0000000000000010	$dif(\boldsymbol{A}, \boldsymbol{B})$
3	0000000000000100	$\overline{\boldsymbol{A}}$
4	0000000000001000	$\overline{\boldsymbol{A} \cdot \boldsymbol{B}}$
5	0000000000010000	$\overline{\boldsymbol{A} + \boldsymbol{B}}$
6	0000000000100000	$\boldsymbol{A} \cdot \boldsymbol{B}$
7	0000000001000000	$\boldsymbol{A} \oplus \boldsymbol{B}$
8	0000000010000000	$\boldsymbol{A} + \boldsymbol{B}$

Conclusion

This lab helped deepen our understanding of how to build and implement a basic 8-bit CPU using arithmetic logic units (ALUs), latches, a 4:16 decoder, and a finite state machine (FSM). The ALU did the calculations, the FSM handled control flow, and the seven-segment display produced the visual outputs, every part was essential to the CPU's operation. Even though I didn't have access to quartus for problem 3 which had three data inputs, the lab still showed how combinational and sequential logic may be combined to produce a CPU.