

Laboratory 6: Design of a Simple Central Processing Unit

Hassan Elshikh (501263004)

COE328 - Digital Systems: Section 2

Vadim Geurkov

Nahid Gozin

Table of Contents:

Table of Contents:	2
Introduction:	3
Components:	3
Part 1:.....	3
Latch 1 & 2:.....	4
FSM:.....	6
4:16 Decoder:.....	11
7-Segment Display:.....	14
Problem 1:	19
ALU 1:.....	19
Combined ALU 1:.....	25
Problem 2:	27
ALU 2:.....	27
Combined ALU 2:.....	33
Problem 3:	36
ALU 3:.....	36
Combined ALU 3:.....	40
Conclusion:	43

Introduction:

The goal of this lab was to design and implement 3 different general purpose low level central processing units (CPUs). Each CPU had a different function depending on the problem sets assigned to the student. This goal was to be achieved using skills and components made in previous labs with some modifications, and some components provided in the lab manual. The components used to make the CPUs involved an arithmetic and logical unit (ALU), two latches, a 4:16 decoder, and a finite state machine (FSM). The FSM used Moore logic for its operation. In short, the ALU performs numerous different functions on the two 8 bit inputs provided by the two latches. The function being performed is dictated by the current state of the FSM, which is passed through the 4:16 decoder to form a 16 bit input. The results of these operations are displayed on 7-segment displays.

Components:

Part 1:

Each student uses their respective student ID for this lab. In this case, my student number is 501263004. This determines the two 8 bit values provided by the two latches as inputs to the ALU, denoted A and B respectively; Where A is the 6th and 7th digits in the student number concatenated to form one two digit number, and B is the 8th and 9th digits in the student number concatenated to form another two digit number. Therefore using this student number, $A = 30$ and $B = 04$. Converting them to binary we find that $A = 00011110_2$ and $B = 00000100_2$.

Latch 1 & 2:

The latch component is the main storage/register component of this system. The function of the latch is to take a binary number as input and pass it as an output with every cycle. This system's circuit utilizes two latches, one for each 8-bit input 'A' and 'B', and passes both of them as inputs to the ALU to perform its operations using them as arguments. In this circuit, both latches are categorized as D flip-flops, since they are controlled using a clock transition (changes state at each rising edge of clock signal).

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

entity register1 is port
(
  A          :    in std_logic_vector(7 downto 0); -- 8-bit A Input
  reset, clock: in std_logic;
  Q          :    out std_logic_vector(7 downto 0)); -- 8-bit Output
end register1;

architecture behavior of register1 is
begin
  process (reset, clock)
  begin
    if reset = '0' then
      Q <= "00000000";
    elsif (rising_edge(clock)) then
      Q <= A;
    end if;
  end process;
end behavior;
```

Figure 1: Latch/Register VHDL Code

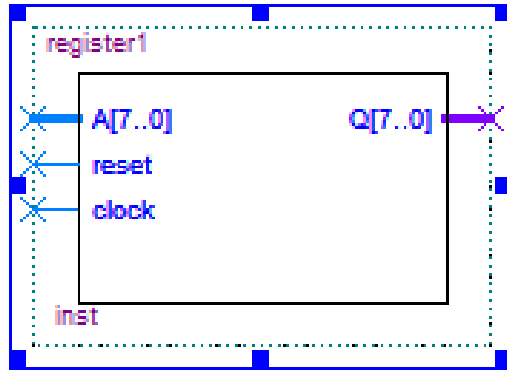


Figure 2: Latch/Register Block Symbol

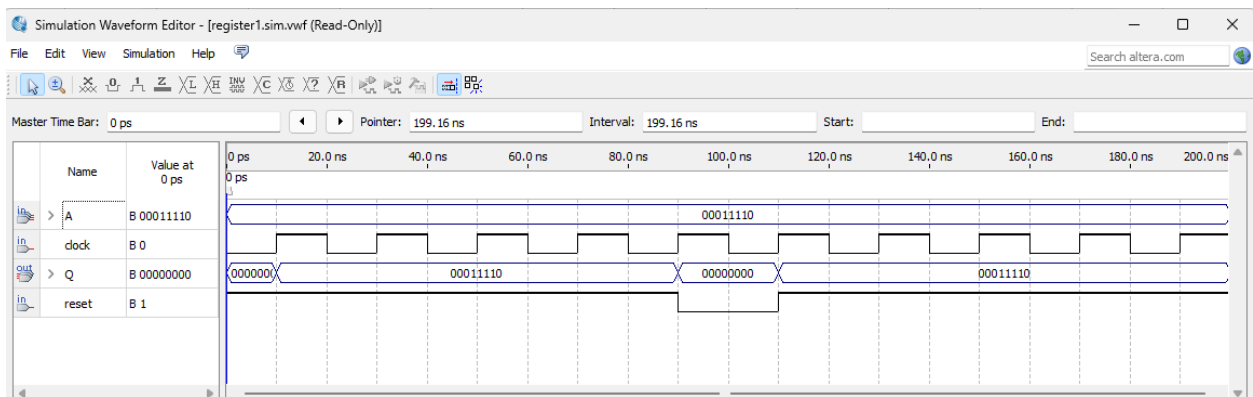


Figure 3: Latch/Register Waveform

Reset	Clock	Rising Edge	A	Q'	Q	Notes
0	X	X	X	X	00000000	Reset condition
1	0	No	10101010	11110000	11110000	No rising edge; retains Q
1	↑	Yes	10101010	11110000	10101010	Rising edge; updates to A

1	↓	No	11001100	10101010	10101010	Falling edge; retains Q
1	↑	Yes	00001111	11001100	00001111	Rising edge; updates to A

Table 1: Latch/Register Truth Table

This truth table is slightly different from the typical truth tables we use for digital logic. I customized this one to show the true function of latches, although the features displayed in this truth table are not used in the actual demonstration of the component, this truth table shows every possible use-case.

FSM:

The FSM used and implemented in this CPU is a finite state machine that uses Moore logic in its operation. It was originally created in a previous lab, but was modified to function for this CPU. It cycles through states S0 to S7. For this to occur, the data input must be 1, and the state switches with every rising edge of the clock input. The states cycle in consecutive order as follows: {S0, S1, S2, S3, S4, S5, S6, S7}. Once the current state is state 7 (S7), the next state will be state 0 (S0) once again and it will essentially loop through these 8 states indefinitely provided the parameters are met. Since the student ID contains 9 digits and the machine only has 8 states, the 2nd through 9th digits of the student ID are used and the first digit is omitted. The states are also used as outputs, and in this circuit they are outputted to the 4:16 decoder to be decoded before being passed through to the ALU to be used as input. Knowing this, we can conclude that the FSM and the decoder make up the control system for the CPU, since they, in conjunction with one another, allow the user to control what happens with the CPU and what operations are being done. Therefore the main purpose of the FSM can be reduced to storing and deciding a

pattern of outputs which are cycled through using a clock signal and a system of states, and supplying these patterns of outputs and states as outgoing data to other components.

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

entity FSM IS
    port(
        data_in, clk, reset    : in std_logic;
        student_id              : out std_logic_vector(3 downto 0);
        current_state           : out std_logic_vector(3 downto 0)
    );
end FSM;

architecture fsm of FSM is
    type state_type is (s0, s1, s2, s3, s4, s5, s6, s7);
    signal yfsm : state_type;

begin

    process (clk, reset) begin

        if reset = '0' then
            yfsm <= s0;

        elsif(rising_edge(clk)) then
            case yfsm is

                when s0 =>
                    if data_in = '1' then
                        yfsm <= s1;

                    end if;

                when s1 =>
                    if data_in = '1' then
                        yfsm <= s2;

                    end if;

                when s2 =>
                    if data_in = '1' then
                        yfsm <= s3;
```

```

        end if;

    when s3 =>
        if data_in = '1' then
            yfsm <= s4;

        end if;

    when s4 =>
        if data_in = '1' then
            yfsm <= s5;

        end if;

    when s5 =>
        if data_in = '1' then
            yfsm <= s6;

        end if;

    when s6 =>
        if data_in = '1' then
            yfsm <= s7;

        end if;

    when s7 =>
        if data_in = '1' then
            yfsm <= s0;

        end if;
    when others =>

    end case;
end if;
end process;

process (yfsm) begin

    case yfsm is -- Student Number 501263004 ~ By: Hassan Elshikh

        when s0 => current_state <= "0000";
            student_id <= "0000";-- d2 ~ 0

```



```

when s1 => current_state <= "0001";
           student_id <= "0001"; -- d3 ~ 1

when s2 => current_state <= "0010";
           student_id <= "0010"; -- d4 ~ 2

when s3 => current_state <= "0011";
           student_id <= "0110"; -- d5 ~ 6

when s4 => current_state <= "0100";
           student_id <= "0011"; -- d6 ~ 3

when s5 => current_state <= "0101";
           student_id <= "0000"; -- d7 ~ 0

when s6 => current_state <= "0110";
           student_id <= "0000"; -- d8 ~ 0

when s7 => current_state <= "0111";
           student_id <= "0100"; -- d9 ~ 4

end case;

end process;

end fsm;

```

Figure 4: FSM VHDL Code

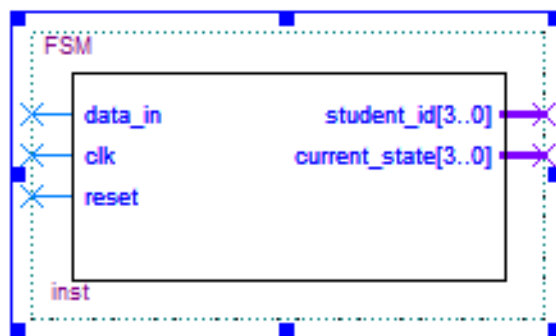


Figure 5: FSM Block Symbol

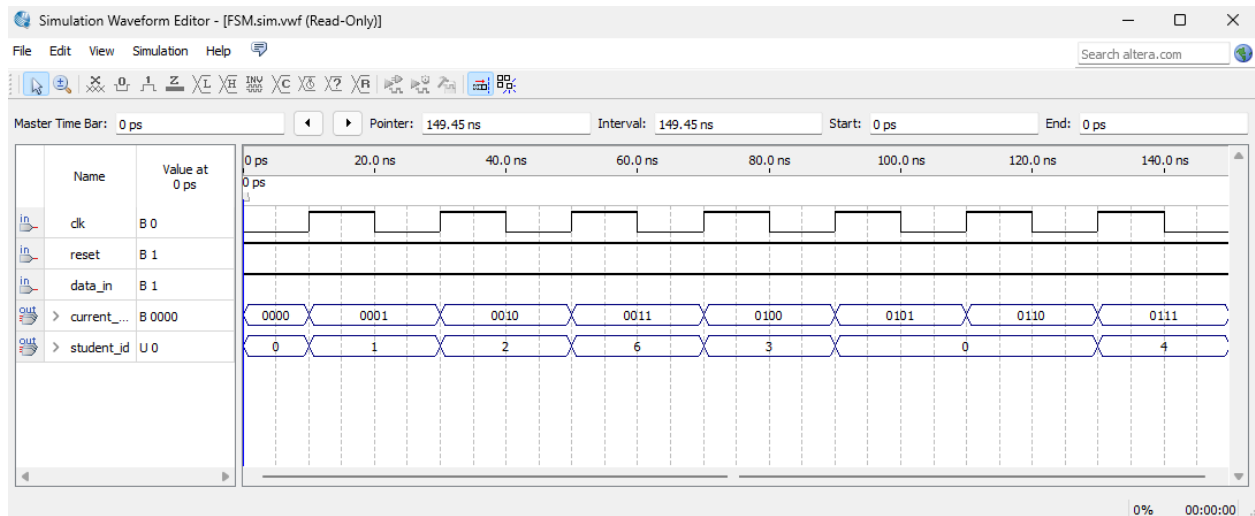


Figure 6: FSM Waveform

Present State	Next State		Output: Student ID	
	Data In = 0	Data In = 1	Binary	Decimal
0000	0000	0001	0000	0
0001	0001	0010	0001	1
0010	0010	0011	0010	2
0011	0011	0100	0110	6
0100	0100	0101	0011	3
0101	0101	0110	0000	0
0110	0110	0111	0000	0
0111	0111	0000	0100	4

Table 2: FSM Truth Table

Observing the truth table above we can see that when the data in value is a '1', that the current state will always change to the next state. Similarly, when the data in value is a '0', the current state remains unchanged. Another thing to note is that the final state changes to the first state when the data in value is '1'; This shows how the FSM repeats states when it reaches its

final state. The student ID column displays my student number and the correlations between each digit and its corresponding state. We can see that the truth table and the waveform align in the values, so we can conclude that the waveform and implementation of the FSM are correct.

4:16 Decoder:

The 4:16 decoder is provided with a 4-bit input, which in this case is the current state of the FSM, and provides a unique 16-bit output for each respective input value given. This is why we stated earlier that the FSM and decoder make up the control unit of the system. For this system, I chose to implement the 4:16 decoder using a schematic involving five 2:4 decoders.

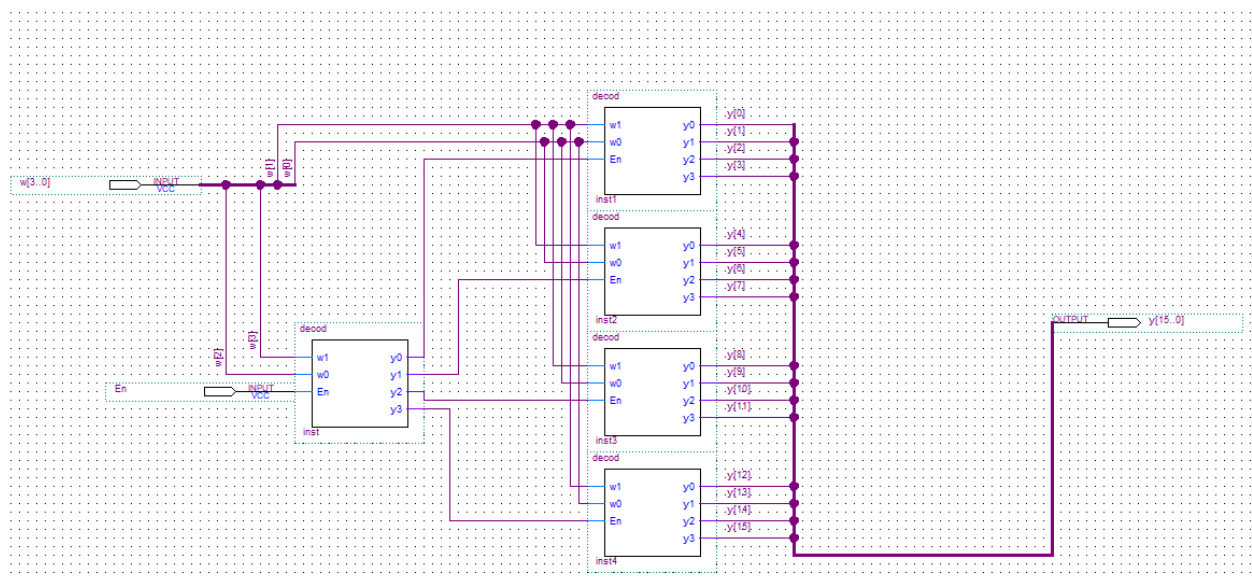


Figure 7: 4:16 Decoder Implemented using five 2:4 Decoders

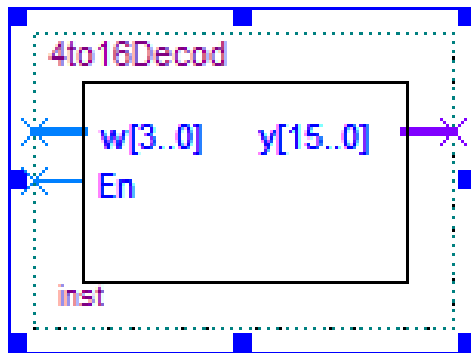


Figure 8: 4:16 Decoder Block Symbol

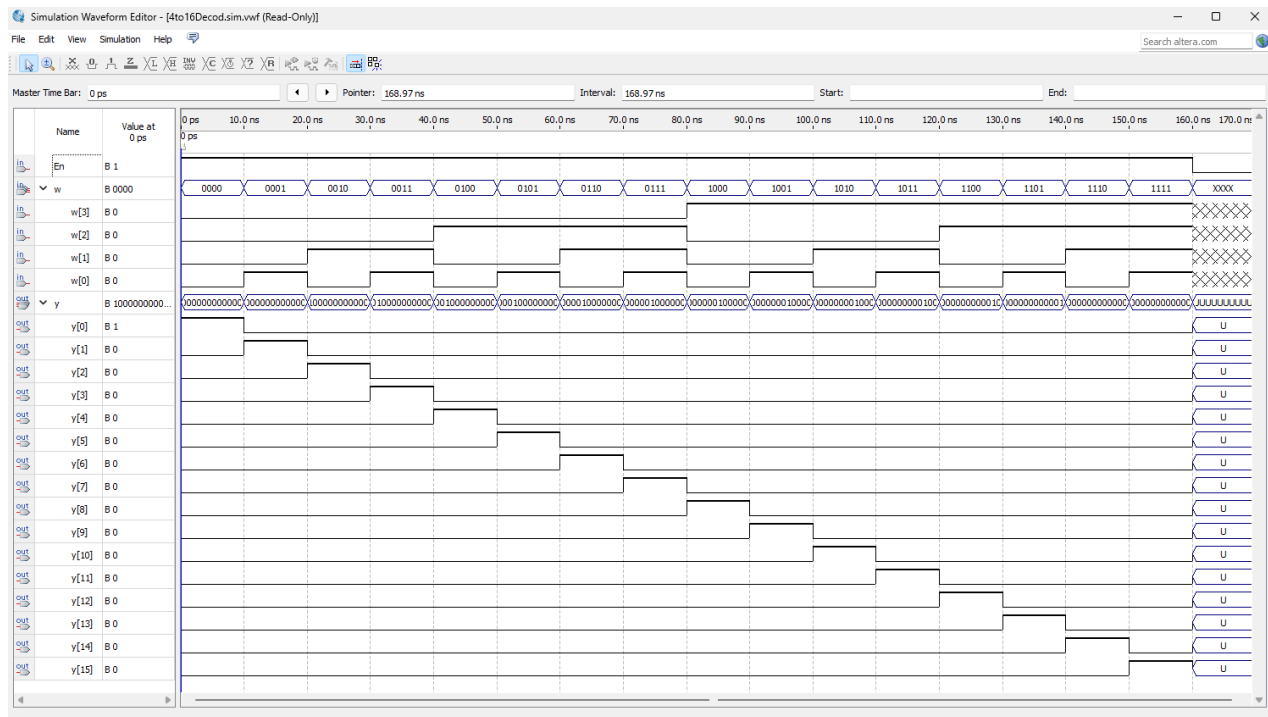


Figure 9: 4:16 Decoder Waveform

Note: As seen above, there is a region ranging from 160.0 ns and 170.0 ns where the output ‘y’ is showing ‘U’. This means that the output is unknown. This is because during this region, the enable input signal “En” is set to ‘0’. This means that the input signals denoted ‘w’ are insignificant and do not have meaning. We can see that I also set those values to “xxxx” meaning that they are variable inputs at this region. I did this to demonstrate that the output of

the decoder is meaningless when the input values are insignificant. In other words, the output is not meaningful so long that the enable signal “En” is set to ‘0’.

[illegible]

Table 3: 4:16 Decoder Truth Table

Once again, observing the truth table above along with the waveform of the 4:16 Decoder, we can see that the set of inputs and outputs correspond well across the two. Therefore we can conclude that the way the 4:16 decoder was implemented is correct.

7-Segment Display:

The 7-Segment display is a component used to take the numeric outputs of other components in binary form, and output them on the board's display for users to observe. This is done using a display that has seven different line segments that form a figure eight. The idea being that any number, along with the first 6 letters of the alphabet can be displayed using only these seven line segments. Therefore we can display any hexadecimal number on these displays. Multiple 7-segment displays were used in this project; two seven segments used for the outputs of the ALU components, as well as another seven segment display used for the student ID output of the FSM. Two seven segments were required for the outputs of the ALU since the outputs of the ALU are 8-bits, while hexadecimal only encapsulates 4-bits per digit. There are 2 iterations of the 7-segment display component, as there was a unique requirement for problem 3. I will demonstrate both iterations below.

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

ENTITY sseg IS
PORT (
    neg          : IN STD_LOGIC;
    bcd          : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    ResultLED, NegativeLED : OUT STD_LOGIC_VECTOR(0 TO 6));
END sseg ;

ARCHITECTURE Behavior OF sseg IS
```

```

BEGIN
PROCESS (bcd, neg) BEGIN

    NegativeLED <= not ("000000" & neg);

CASE bcd IS --abcdefg
WHEN "0000" => ResultLED <= NOT ("1111110");
WHEN "0001" => ResultLED <= NOT ("0110000");
WHEN "0010" => ResultLED <= NOT ("1101101");
WHEN "0011" => ResultLED <= NOT ("1111001");
WHEN "0100" => ResultLED <= NOT ("0110011");
WHEN "0101" => ResultLED <= NOT ("1011011");
WHEN "0110" => ResultLED <= NOT ("1011111");
WHEN "0111" => ResultLED <= NOT ("1110000");
WHEN "1000" => ResultLED <= NOT ("1111111");
WHEN "1001" => ResultLED <= NOT ("1110011");
WHEN "1010" => ResultLED <= ("0001000");
WHEN "1011" => ResultLED <= ("1100000");
WHEN "1100" => ResultLED <= ("0110001");
WHEN "1101" => ResultLED <= ("1000010");
WHEN "1110" => ResultLED <= ("0110000");
WHEN "1111" => ResultLED <= ("0111000");
WHEN OTHERS => ResultLED <= ("-----");
END CASE;
END PROCESS;
END Behavior;

```

Figure 10: 7-Segment Display (First Iteration) VHDL Code

```

LIBRARY ieee ;

USE ieee.std_logic_1164.all ;

ENTITY sseg3 IS

PORT (

```

```

bcd          : IN STD_LOGIC_VECTOR(3 DOWNTO 0);

ResultLED : OUT STD_LOGIC_VECTOR(0 TO 6));

END sseg3 ;

ARCHITECTURE Behavior OF sseg3 IS

BEGIN

PROCESS (bcd) BEGIN

CASE bcd IS --abcdefg

WHEN "0000" => ResultLED <= ("1101010"); -- n

WHEN "1111" => ResultLED <= ("1000100"); -- y

WHEN OTHERS => ResultLED <= ("-----");

END CASE;

END PROCESS;

END Behavior;

```

Figure 11: 7-Segment Display (Second Iteration) VHDL Code

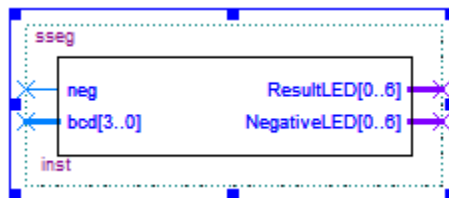


Figure 12: 7-Segment Display (First Iteration) Block Symbol

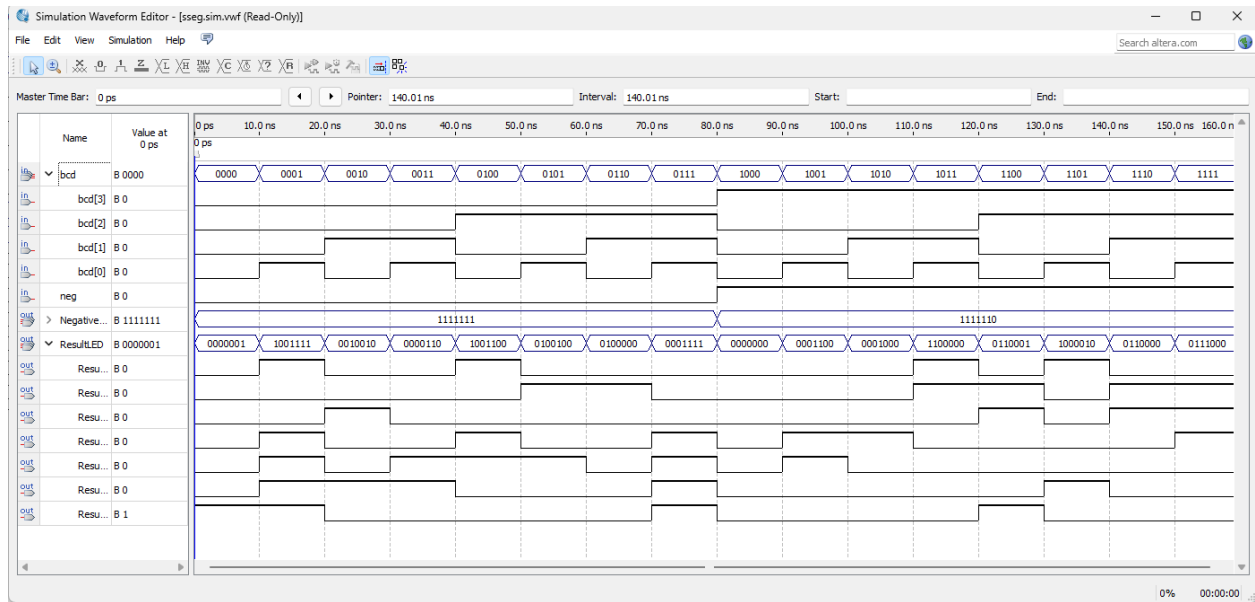


Figure 13: 7-Segment Display (First Iteration) Waveform

Note: The second half of the waveform, the “neg” input is set to ‘1’. This changes the NegativeLED Output from “1111111” to “1111110”. This is because the final bit which changes from ‘1’ to ‘0’ is the bit that dictates whether the 7th segment of the display is on or off. When NegativeLED is at “1111111”, every segment is off; whereas when it is set to “1111110”, every segment is off except for the final segment, which will form a negative sign. This helps us display negative numbers on the 7-segment displays.

BCD (input)	ResultLED (output)
0000	0000001
0001	0110000
0010	0010010
0011	0000110
0100	1001100
0101	1011011

0110	1011111
0111	1110000
1000	1111111
1001	1110011
1010	0001000
1011	1100000
1100	0110001
1101	1000010
1110	0110000
1111	0111000

Table 4: 7-Segment Display (First Iteration) Numbers

neg	NegativeLED
0	1111111
1	1111110

Table 5: 7-Segment Display (First Iteration) Negative Sign

Note: The first and second iterations of the 7-Segment display work fundamentally identically, however for different purposes. The first iteration is used to display numerical and alphabetical values with the capability of displaying negative signs, whereas the second iteration simply displays the letters ‘y’ and ‘n’ depending on the function of the ALU. This will be expanded upon further at a later section in the report.’

Problem 1:

ALU 1:

The purpose of the Arithmetic Logic Unit (ALU) component is to perform operations on which are dictated by the states outputted from the FSM through the decoder. In this lab, there are 8 of the following operations to be implemented in ALU 1:

Function #	Function
1	$sum(A, B)$
2	$dif(A, B)$
3	\bar{A}
4	$\overline{A \cdot B}$
5	$\overline{A + B}$
6	$A \cdot B$
7	$A \oplus B$
8	$A + B$

Table 6: ALU 1 Core Operations

The operations above were implemented by adding the necessary code to the outline provided in the lab manual. The first ALU has a total of 5 inputs. These inputs include: Clock, A, B, Reset, and Opcode.

The clock is the input which is shared between all of the clock regulated components, such as the latches/registers, FSM and ALU. It is the input which cycles through any sequence oriented component. The clock input is a signal which alternates between '0' and '1', which is

controlled by the users using push-buttons on the board once the project is implemented on the boards.

The A and B inputs are the values being passed as arguments for the operations from the latches/registers. Each value is 8-bits respectively and come from identical latch/register components.

The reset is the input which controls if the ALU resets to its original state and output. This input, like the clock input, is shared between any components which can be reset; meaning any component in the circuit which can change states through the clock input. This way, when the FSM resets, the latch and ALU also reset to correspond to the current or initial state of the FSM.

Opcode is an input which controls which operation or function of the ALU will be completed. It is a 16-bit input, coming from the 4:16 decoder. It is essentially the 4-bit current state output of the FSM, decoded by the 4:16 decoder to become a 16-bit input for the ALU to use as an operation code. This means that we can assign each operation its own unique and respective operation code, which would be a 16-bit binary value.

The first ALU also has three outputs, denoted as R1, R2, for “Result 1” and “Result 2” and neg for “Negative”. As mentioned previously, there are two different results because some operations may result in more than 8-bits, and the 7-segment displays can only display hexadecimal values on each respective 7-segment display. Since hexadecimal values only encapsulate 8-bit values, we need two hexadecimal values for two 7-segment displays, hence the two results. This feature is common among both ALU 1 and ALU 2. The output denoted neg for “Negative” is a singular bit output, which is passed to the same two 7-segment displays which display the R1 and R2 values. It is used so that in the event that the result is a negative number,

the 7-segment displays can simply display the results in unsigned hexadecimal values, and place a negative sign in front of each value.

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

entity ALU is port
(
    clock, reset    : in std_logic;
    A, B            : in unsigned(7 downto 0);
    opcode          : in unsigned(15 downto 0);
    neg             : out std_logic;
    R1, R2          : out unsigned(3 downto 0));
end ALU;

architecture calculation of ALU is
    Signal Result : unsigned(7 downto 0);
begin

    process (clock, reset) begin
        if reset = '0' then
            Result <= "00000000";
        elsif (rising_edge(clock)) then
            case opcode is

                when "0000000000000001" => -- function 1 ~ addition
                    neg <= '0';
                    Result <= A + B;

                when "0000000000000010" => -- function 2 ~ subtraction
                    if(A < B) then
                        neg <= '1';
                        Result <= B - A;
                    else
                        neg <= '0';
                        Result <= A - B;
                    end if;

                when "0000000000000100" => -- function 3 ~ NOT
```

```

        neg <= '0';
        Result <= NOT(A);
    when "0000000000001000" => -- function 4 ~ NAND
        neg <= '0';
        Result <= NOT (A AND B);
    when "0000000000010000" => -- function 5 ~ NOR
        neg <= '0';
        Result <= A NOR B;
    when "000000000100000" => -- function 6 ~ AND
        neg <= '0';
        Result <= A AND B;
    when "000000001000000" => -- function 7 ~ XOR
        neg <= '0';
        Result <= A XOR B;
    when "000000010000000" => -- function 8 ~ OR
        neg <= '0';
        Result <= A OR B;

    when others => -- don't care
        end case;
    end if;
end process;

R1 <= Result(3 downto 0);
R2 <= Result(7 downto 4);
end calculation;

```

Figure 14: ALU 1 VHDL Code

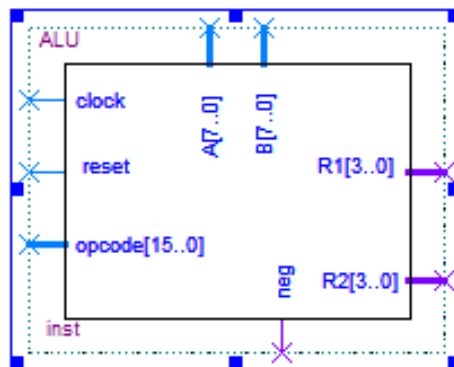


Figure 15: ALU 1 Block Symbol

0000000000100000	00011110	30	00000100	4	0	4
0000000000100000	00011110	30	00000100	4	1	A
0000000010000000	00011110	30	00000100	4	1	E

Table 7: ALU 1 Expected Outputs

Handwritten work for ALU 1 Operation Outputs:

- $A+B = 30+4 = 34 \xrightarrow{\text{Hex}} 22$
- $A-B = 30-4 = 26 \xrightarrow{\text{Hex}} 1A$
- $\overline{A} = \overline{00011110} = 11100001 \xrightarrow{\text{Hex}} E1$
- $\overline{A \cdot B} : 00011110$
 $\text{NAND } \frac{00000100}{1111011} \xrightarrow{\text{Hex}} FB$
- $\overline{A+B} : 00011110$
 $\text{NOR } \frac{00000100}{11100001} \xrightarrow{\text{Hex}} E1$
- $A \cdot B : 00011110$
 $\text{AND } \frac{00000100}{00000100} \xrightarrow{\text{Hex}} 04$
- $A \oplus B : 00011110$
 $\text{XOR } \frac{00000100}{00011010} \xrightarrow{\text{Hex}} 1A$
- $A+B : 00011110$
 $\text{OR } \frac{00000100}{00011110} \xrightarrow{\text{Hex}} 1E$

Figure 17: Handwritten Work for ALU 1 Operation Outputs

As we can see from the handwritten work, ALU 1 output table and the ALU 1 waveform, these three are in correspondence with each other, confirming that the expected outputs are being achieved and the implementation is correct so far.

Combined ALU 1:

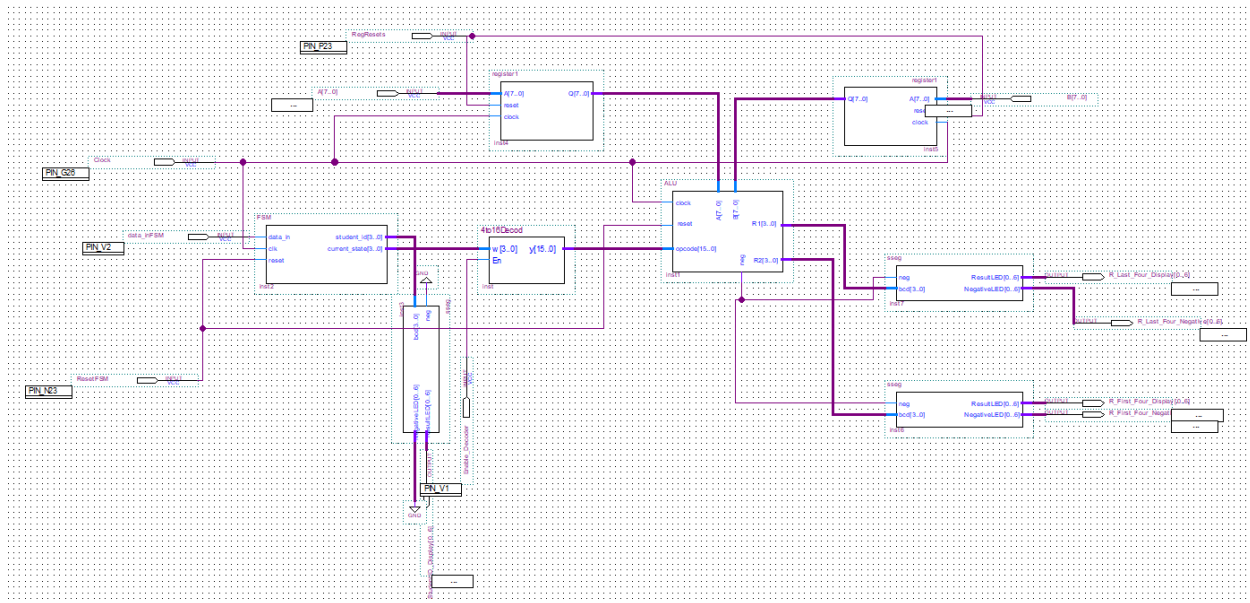


Figure 18: Combined ALU 1 Block Diagram

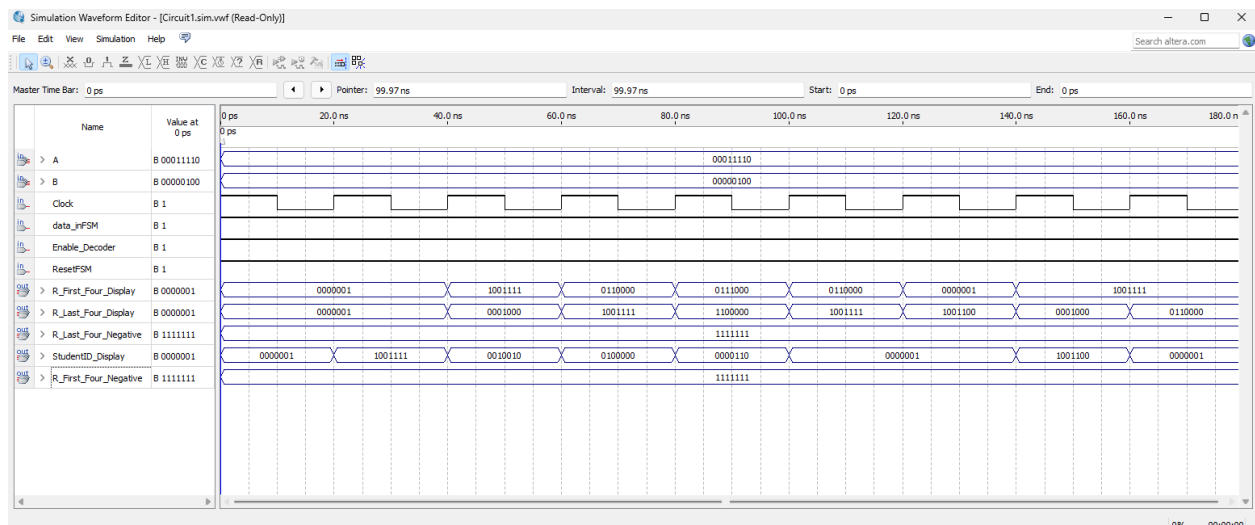


Figure 19: Combined ALU 1 Waveform

In the waveform above, we can verify the outputs being correct, although it is tedious. To verify the outputs, we must convert the “R_First_Four_Display”, “R_Last_Four_Display”, and “StudentID_Display” values into actual numerical values. However, these values provided in the waveforms are not numerical values, rather they are the binary outputs to the 7-segment displays,

so we must draw the outputs by hand to confirm the values. I have done this to confirm that the circuit functions correctly, and the demonstration of this won't be necessary as we have the actual 7-segment displays working on the board which will be demonstrated below.




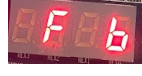
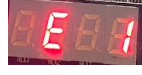
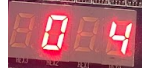


Operation		Expected Output (Hex)	Empirical Output (Hex)
1	$sum(A, B)$	22	
2	$dif(A, B)$	1A	
3	\bar{A}	E1	
4	$\overline{A \cdot B}$	FB	
5	$\overline{A + B}$	E1	
6	$A \cdot B$	04	
7	$A \oplus B$	1A	
8	$A + B$	1E	

Table 8: Combined ALU 1 Empirical Outputs

in	A[7]	Input	PIN_U4	1	B1_N0	PIN_U4	3.3-V LV..default	24mA (default)	
in	A[6]	Input	PIN_U3	1	B1_N0	PIN_U3	3.3-V LV..default	24mA (default)	
in	A[5]	Input	PIN_T7	1	B1_N0	PIN_T7	3.3-V LV..default	24mA (default)	
in	A[4]	Input	PIN_P2	1	B1_N0	PIN_P2	3.3-V LV..default	24mA (default)	
in	A[3]	Input	PIN_P1	1	B1_N0	PIN_P1	3.3-V LV..default	24mA (default)	
in	A[2]	Input	PIN_N1	2	B2_N1	PIN_N1	3.3-V LV..default	24mA (default)	
in	A[1]	Input	PIN_A13	4	B4_N1	PIN_A13	3.3-V LV..default	24mA (default)	
in	A[0]	Input	PIN_B13	4	B4_N1	PIN_B13	3.3-V LV..default	24mA (default)	
in	B[7]	Input	PIN_C13	3	B3_N0	PIN_C13	3.3-V LV..default	24mA (default)	
in	B[6]	Input	PIN_AC13	8	B8_N0	PIN_AC13	3.3-V LV..default	24mA (default)	
in	B[5]	Input	PIN_AD13	8	B8_N0	PIN_AD13	3.3-V LV..default	24mA (default)	
in	B[4]	Input	PIN_AF14	7	B7_N1	PIN_AF14	3.3-V LV..default	24mA (default)	
in	B[3]	Input	PIN_AE14	7	B7_N1	PIN_AE14	3.3-V LV..default	24mA (default)	
in	B[2]	Input	PIN_P25	6	B6_N0	PIN_P25	3.3-V LV..default	24mA (default)	
in	B[1]	Input	PIN_N26	5	B5_N1	PIN_N26	3.3-V LV..default	24mA (default)	
in	B[0]	Input	PIN_N25	5	B5_N1	PIN_N25	3.3-V LV..default	24mA (default)	
in	Clock	Input	PIN_G26	5	B5_N0	PIN_G26	3.3-V LV..default	24mA (default)	
in	data_inFSM	Input	PIN_V2	1	B1_N0	PIN_V2	3.3-V LV..default	24mA (default)	
in	Enable_Decoder	Input	PIN_V1	1	B1_N0	PIN_V1	3.3-V LV..default	24mA (default)	
out	R_First_Four_Display[0]	Output	PIN_AB23	6	B6_N1	PIN_AB23	3.3-V LV..default	24mA (default)	
out	R_First_Four_Display[1]	Output	PIN_V22	6	B6_N1	PIN_V22	3.3-V LV..default	24mA (default)	
out	R_First_Four_Display[2]	Output	PIN_AC25	6	B6_N1	PIN_AC25	3.3-V LV..default	24mA (default)	
out	R_First_Four_Display[3]	Output	PIN_AC26	6	B6_N1	PIN_AC26	3.3-V LV..default	24mA (default)	
out	R_First_Four_Display[4]	Output	PIN_AB26	6	B6_N1	PIN_AB26	3.3-V LV..default	24mA (default)	
out	R_First_Four_Display[5]	Output	PIN_AB25	6	B6_N1	PIN_AB25	3.3-V LV..default	24mA (default)	
out	R_First_Four_Display[6]	Output	PIN_Y24	6	B6_N1	PIN_Y24	3.3-V LV..default	24mA (default)	
out	R_First_Four_Negative[0]	Output	PIN_Y23	6	B6_N1	PIN_Y23	3.3-V LV..default	24mA (default)	
out	R_First_Four_Negative[1]	Output	PIN_AA25	6	B6_N1	PIN_AA25	3.3-V LV..default	24mA (default)	
out	R_First_Four_Negative[2]	Output	PIN_AA26	6	B6_N1	PIN_AA26	3.3-V LV..default	24mA (default)	
out	R_First_Four_Negative[3]	Output	PIN_Y26	6	B6_N1	PIN_Y26	3.3-V LV..default	24mA (default)	
out	R_First_Four_Negative[4]	Output	PIN_Y25	6	B6_N1	PIN_Y25	3.3-V LV..default	24mA (default)	
out	R_First_Four_Negative[5]	Output	PIN_U22	6	B6_N1	PIN_U22	3.3-V LV..default	24mA (default)	
out	R_First_Four_Negative[6]	Output	PIN_W24	6	B6_N1	PIN_W24	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Display[0]	Output	PIN_AF10	8	B8_N0	PIN_AF10	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Display[1]	Output	PIN_AB12	8	B8_N0	PIN_AB12	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Display[2]	Output	PIN_AC12	8	B8_N0	PIN_AC12	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Display[3]	Output	PIN_AD11	8	B8_N0	PIN_AD11	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Display[4]	Output	PIN_AE11	8	B8_N0	PIN_AE11	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Display[5]	Output	PIN_V14	8	B8_N0	PIN_V14	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Display[6]	Output	PIN_V13	8	B8_N0	PIN_V13	3.3-V LV..default	24mA (default)	

out	R_Last_Four_Negative[0]	Output	PIN_V20	6	B6_N1	PIN_V20	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Negative[1]	Output	PIN_V21	6	B6_N1	PIN_V21	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Negative[2]	Output	PIN_W21	6	B6_N1	PIN_W21	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Negative[3]	Output	PIN_Y22	6	B6_N1	PIN_Y22	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Negative[4]	Output	PIN_AA24	6	B6_N1	PIN_AA24	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Negative[5]	Output	PIN_AA23	6	B6_N1	PIN_AA23	3.3-V LV..default	24mA (default)	
out	R_Last_Four_Negative[6]	Output	PIN_AB24	6	B6_N1	PIN_AB24	3.3-V LV..default	24mA (default)	
in	RegResets	Input	PIN_P23	6	B6_N0	PIN_P23	3.3-V LV..default	24mA (default)	
in	ResetFSM	Input	PIN_N23	5	B5_N1	PIN_N23	3.3-V LV..default	24mA (default)	
out	StudentID_Display[0]	Output	PIN_U9	1	B1_N0	PIN_U9	3.3-V LV..default	24mA (default)	
out	StudentID_Display[1]	Output	PIN_U1	1	B1_N0	PIN_U1	3.3-V LV..default	24mA (default)	
out	StudentID_Display[2]	Output	PIN_U2	1	B1_N0	PIN_U2	3.3-V LV..default	24mA (default)	
out	StudentID_Display[3]	Output	PIN_T4	1	B1_N0	PIN_T4	3.3-V LV..default	24mA (default)	
out	StudentID_Display[4]	Output	PIN_R7	1	B1_N0	PIN_R7	3.3-V LV..default	24mA (default)	
out	StudentID_Display[5]	Output	PIN_R6	1	B1_N0	PIN_R6	3.3-V LV..default	24mA (default)	
out	StudentID_Display[6]	Output	PIN_T3	1	B1_N0	PIN_T3	3.3-V LV..default	24mA (default)	

Figure 20: Combined ASU 1 Pin Planner

Problem 2:

ALU 2:

In problem two we are asked to create a new ALU with new operations and a new surrounding circuit to form a CPU with a different purpose. The purpose of the second ALU (ALU 2) is archetypally identical to the first ALU (ALU 1), however with a different set of operations/functions. The function set that was assigned to me for this problem was function set (a):

Function #	Operation / Function
1	Increment A by 2
2	Shift B to the right by two bits, input bit = 0 (SHR)
3	Shift A to the right by four bits, input bit = 1 (SHR)
4	Find the smaller value of A and B and produce the results (Min(A , B))
5	Rotate A to the right by two bits (ROR)
6	Invert the bit-significance order of B
7	Produce the result of XORing A and B
8	Produce the summation of A and B , then decrease it by 4

Table 9: ALU 2 Core Operations

The operations above are practically the only difference which separates ALU 2 from ALU 1; besides for these differences, the rest of the circuit and system created around the ALUs are virtually identical.

```

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

entity ALU2 is port
(
    clock, reset    : in std_logic;
    A, B            : in unsigned(7 downto 0);
    opcode          : in unsigned(15 downto 0);
    neg             : out std_logic;
    R1, R2          : out unsigned(3 downto 0));
end ALU2;

architecture calculation of ALU2 is
    Signal Result : unsigned(7 downto 0);

```

```

begin

    process (clock, reset) begin
        -- PROBLEM 2: MODIFIED
        ALU CORE 1, ALU OPERATIONS ASSIGNED: a)
        if reset = '0' then
            Result <= "00000000";
        elsif (rising_edge(clock)) then
            case opcode is
                when "0000000000000001" => -- function 1 ~
Increment A by 2
                    neg <= '0';
                    Result <= A + 2;
                when "0000000000000010" => -- function 2 ~
Shift B to the right by 2 bits, input bit = 0
                    neg <= '0';
                    Result <= "00" & B(7 downto 2);
                when "0000000000000100" => -- function 3 ~
Shift A to the right by four bits, input bit = 1
                    neg <= '0';
                    Result <= "1111" & A(7 downto 4);
                when "0000000000001000" => -- function 4 ~
Min(A, B)
                    if (A < B) then
                        Result <= A;
                    else
                        Result <= B;
                    end if;
                    if (Result < 0) then
                        neg <= '1';
                    else
                        neg <= '0';
                    end if;
                when "0000000000010000" => -- function 5 ~
Rotate A to the right by two bits (ROR)
                    neg <= '0';
                    Result <= A(1 downto 0) & A(7 downto 2);
                when "0000000000100000" => -- function 6 ~
Invert the bit-significance order of B
                    neg <= '0';

```

```

        Result(0) <= B(7);
        Result(1) <= B(6);
        Result(2) <= B(5);
        Result(3) <= B(4);
        Result(4) <= B(3);
        Result(5) <= B(2);
        Result(6) <= B(1);
        Result(7) <= B(0);
        when "000000001000000" => -- function 7 ~
Produce the result of XORing A and B
        neg <= '0';
        Result <= A XOR B;
        when "000000001000000" => -- function 8 ~ Sum
of A and B, decreased by 4
        Result <= ((A + B) - 4);
        if (Result < 0) then
        neg <= '1';
        else
        neg <= '0';
        end if;
        when others => -- don't
care

        end case;
    end if;
end process;

R1 <= Result(3 downto 0);
R2 <= Result(7 downto 4);
end calculation;

```

Figure 21: ALU 2 VHDL Code

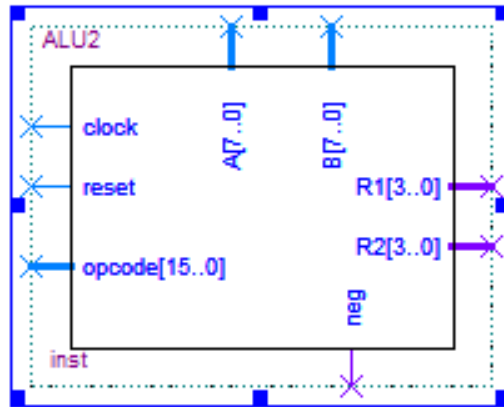


Figure 22: ALU 2 Block Symbol

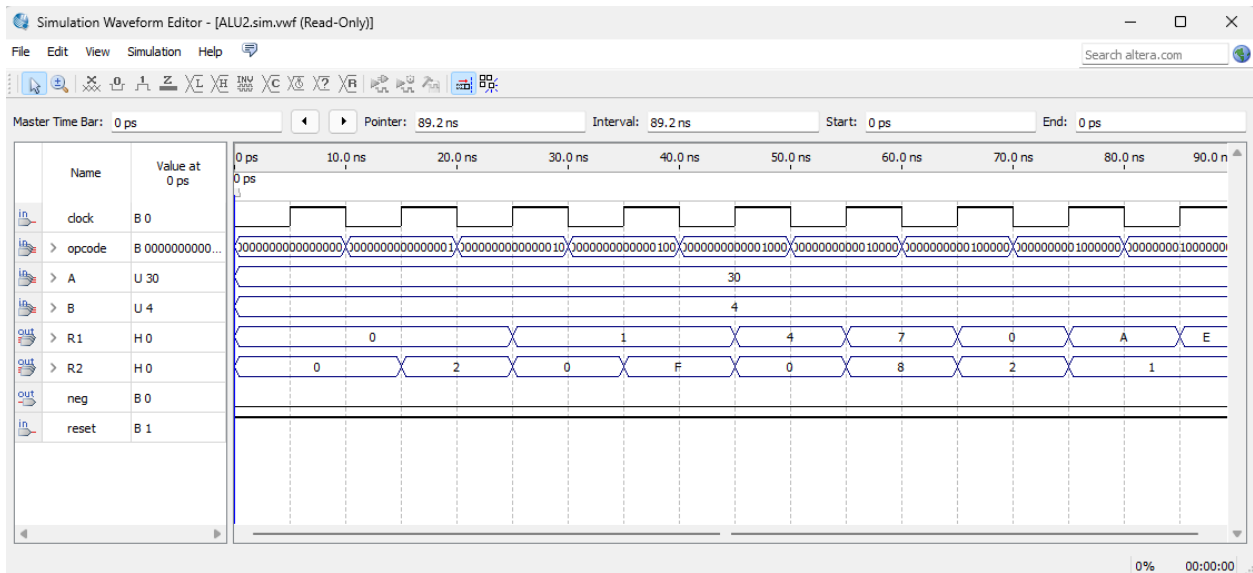


Figure 23: ALU 2 Waveform

Opcode	A		B		R2	R1
Binary	Binary	Decimal	Binary	Decimal	Hexadecimal	Hexadecimal
0000000000000001	00011110	30	00000100	4	2	0
0000000000000010	00011110	30	00000100	4	0	1
00000000000000100	00011110	30	00000100	4	F	1
00000000000001000	00011110	30	00000100	4	0	4
00000000000010000	00011110	30	00000100	4	8	7

0000000000100000	00011110	30	00000100	4	2	0
0000000000100000	00011110	30	00000100	4	1	A
0000000001000000	00011110	30	00000100	4	1	E

Table 10: ALU 2 Expected Outputs

Figure 25: Combined ALU 2 Block Diagram

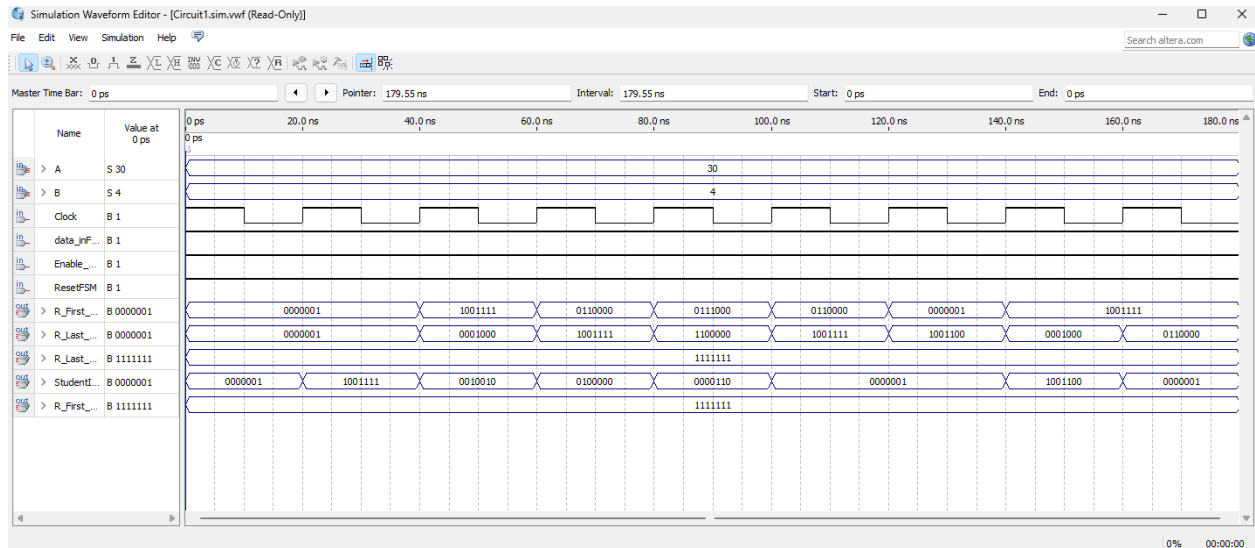






Figure 26: Combined ALU 2 Waveform

Once again, as mentioned previously in the section of the combined ALU 1 waveform, the demonstration of the verification of these outputs is tedious and unnecessary. Instead the verification of the circuit will be demonstrated below with the outputs.

Operation		Expected Output (Hex)	Empirical Output (Hex)
1	Increment A by 2	20	
2	Shift B to the right by two bits, input bit = 0 (SHR)	01	
3	Shift A to the right by four bits, input bit = 1 (SHR)	F1	
4	Find the smaller value of A and B and produce the results: Min(A , B)	04	





5	Rotate A to the right by two bits (ROR)	87	
6	Invert the bit-significance order of B	20	
7	Produce the result of XORing A and B	1A	
8	Produce the summation of A and B , then decrease it by 4	1E	

Table 11: Combined ALU 2 Empirical Outputs

in_ A[7]	Input	PIN_U4	1	B1_N0	PIN_U4	3.3-V LV...default	24mA (default)
in_ A[6]	Input	PIN_U3	1	B1_N0	PIN_U3	3.3-V LV...default	24mA (default)
in_ A[5]	Input	PIN_T7	1	B1_N0	PIN_T7	3.3-V LV...default	24mA (default)
in_ A[4]	Input	PIN_P2	1	B1_N0	PIN_P2	3.3-V LV...default	24mA (default)
in_ A[3]	Input	PIN_P1	1	B1_N0	PIN_P1	3.3-V LV...default	24mA (default)
in_ A[2]	Input	PIN_N1	2	B2_N1	PIN_N1	3.3-V LV...default	24mA (default)
in_ A[1]	Input	PIN_A13	4	B4_N1	PIN_A13	3.3-V LV...default	24mA (default)
in_ A[0]	Input	PIN_B13	4	B4_N1	PIN_B13	3.3-V LV...default	24mA (default)
in_ B[7]	Input	PIN_C13	3	B3_N0	PIN_C13	3.3-V LV...default	24mA (default)
in_ B[6]	Input	PIN_AC13	8	B8_N0	PIN_AC13	3.3-V LV...default	24mA (default)
in_ B[5]	Input	PIN_AD13	8	B8_N0	PIN_AD13	3.3-V LV...default	24mA (default)
in_ B[4]	Input	PIN_AF14	7	B7_N1	PIN_AF14	3.3-V LV...default	24mA (default)
in_ B[3]	Input	PIN_AE14	7	B7_N1	PIN_AE14	3.3-V LV...default	24mA (default)
in_ B[2]	Input	PIN_P25	6	B6_N0	PIN_P25	3.3-V LV...default	24mA (default)
in_ B[1]	Input	PIN_N26	5	B5_N1	PIN_N26	3.3-V LV...default	24mA (default)
in_ B[0]	Input	PIN_N25	5	B5_N1	PIN_N25	3.3-V LV...default	24mA (default)
in_ Clock	Input	PIN_G26	5	B5_N0	PIN_G26	3.3-V LV...default	24mA (default)
in_ data_inFSM	Input	PIN_V2	1	B1_N0	PIN_V2	3.3-V LV...default	24mA (default)
in_ Enable_Decoder	Input	PIN_V1	1	B1_N0	PIN_V1	3.3-V LV...default	24mA (default)
out_ R_First_Four_Display[0]	Output	PIN_AB23	6	B6_N1	PIN_AB23	3.3-V LV...default	24mA (default)
out_ R_First_Four_Display[1]	Output	PIN_V22	6	B6_N1	PIN_V22	3.3-V LV...default	24mA (default)
out_ R_First_Four_Display[2]	Output	PIN_AC25	6	B6_N1	PIN_AC25	3.3-V LV...default	24mA (default)
out_ R_First_Four_Display[3]	Output	PIN_AC26	6	B6_N1	PIN_AC26	3.3-V LV...default	24mA (default)
out_ R_First_Four_Display[4]	Output	PIN_AB26	6	B6_N1	PIN_AB26	3.3-V LV...default	24mA (default)
out_ R_First_Four_Display[5]	Output	PIN_AB25	6	B6_N1	PIN_AB25	3.3-V LV...default	24mA (default)
out_ R_First_Four_Display[6]	Output	PIN_Y24	6	B6_N1	PIN_Y24	3.3-V LV...default	24mA (default)
out_ R_First_Four_Negative[0]	Output	PIN_Y23	6	B6_N1	PIN_Y23	3.3-V LV...default	24mA (default)
out_ R_First_Four_Negative[1]	Output	PIN_AA25	6	B6_N1	PIN_AA25	3.3-V LV...default	24mA (default)
out_ R_First_Four_Negative[2]	Output	PIN_AA26	6	B6_N1	PIN_AA26	3.3-V LV...default	24mA (default)
out_ R_First_Four_Negative[3]	Output	PIN_Y26	6	B6_N1	PIN_Y26	3.3-V LV...default	24mA (default)
out_ R_First_Four_Negative[4]	Output	PIN_Y25	6	B6_N1	PIN_Y25	3.3-V LV...default	24mA (default)
out_ R_First_Four_Negative[5]	Output	PIN_U22	6	B6_N1	PIN_U22	3.3-V LV...default	24mA (default)
out_ R_First_Four_Negative[6]	Output	PIN_W24	6	B6_N1	PIN_W24	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Display[0]	Output	PIN_AF10	8	B8_N0	PIN_AF10	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Display[1]	Output	PIN_AB12	8	B8_N0	PIN_AB12	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Display[2]	Output	PIN_AC12	8	B8_N0	PIN_AC12	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Display[3]	Output	PIN_AD11	8	B8_N0	PIN_AD11	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Display[4]	Output	PIN_AE11	8	B8_N0	PIN_AE11	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Display[5]	Output	PIN_V14	8	B8_N0	PIN_V14	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Display[6]	Output	PIN_V13	8	B8_N0	PIN_V13	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Negative[0]	Output	PIN_V20	6	B6_N1	PIN_V20	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Negative[1]	Output	PIN_V21	6	B6_N1	PIN_V21	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Negative[2]	Output	PIN_W21	6	B6_N1	PIN_W21	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Negative[3]	Output	PIN_Y22	6	B6_N1	PIN_Y22	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Negative[4]	Output	PIN_AA24	6	B6_N1	PIN_AA24	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Negative[5]	Output	PIN_AA23	6	B6_N1	PIN_AA23	3.3-V LV...default	24mA (default)
out_ R_Last_Four_Negative[6]	Output	PIN_AB24	6	B6_N1	PIN_AB24	3.3-V LV...default	24mA (default)
in_ RegResets	Input	PIN_P23	6	B6_N0	PIN_P23	3.3-V LV...default	24mA (default)
in_ ResetFSM	Input	PIN_N23	5	B5_N1	PIN_N23	3.3-V LV...default	24mA (default)
out_ StudentID_Display[0]	Output	PIN_U9	1	B1_N0	PIN_U9	3.3-V LV...default	24mA (default)
out_ StudentID_Display[1]	Output	PIN_U1	1	B1_N0	PIN_U1	3.3-V LV...default	24mA (default)
out_ StudentID_Display[2]	Output	PIN_U2	1	B1_N0	PIN_U2	3.3-V LV...default	24mA (default)
out_ StudentID_Display[3]	Output	PIN_T4	1	B1_N0	PIN_T4	3.3-V LV...default	24mA (default)
out_ StudentID_Display[4]	Output	PIN_R7	1	B1_N0	PIN_R7	3.3-V LV...default	24mA (default)
out_ StudentID_Display[5]	Output	PIN_R6	1	B1_N0	PIN_R6	3.3-V LV...default	24mA (default)
out_ StudentID_Display[6]	Output	PIN_T3	1	B1_N0	PIN_T3	3.3-V LV...default	24mA (default)

Figure 27: Combined ALU 2 Pin Planner

Problem 3:

ALU 3:

In problem three we are asked to create a third and final ALU with a very different set of operations to form another CPU with a new purpose. The purpose of the third ALU is very different from the purpose of the first two iterations. Once again, similarly to problem 2, a function set was assigned for this problem, and this function set was (i):

“For each opcode submitted to the ALU, display ‘y’ if one of the 2 digits of B is less than the student_id signal value and ‘n’ otherwise”.

Essentially, we want to display the student ID, and if one of the two digits of B are less than the student ID digit being displayed, the letter ‘y’ should be displayed. Otherwise the letter ‘n’ should be displayed. We first must separate the 8-bit B value into two 4-bit digits. Since the B value is equal to 4 which is 00000100_2 in binary, the two corresponding digits would be:

$$d1 = 0000_2 = 0_{10}$$

$$d2 = 0100_2 = 4_{10}$$

Therefore for any non-zero values of the studentID, we want the display to show ‘y’, and for all zero values we want the display to show ‘n’.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALU3 is
    port (
        clock      : in std_logic;
        reset      : in std_logic;
        A, B       : in unsigned(7 downto 0);
        opcode     : in unsigned(15 downto 0);
```

```

        student_id : in unsigned(3 downto 0);
        neg        : out std_logic; -- Not used but left in for
completeness
        Result     : out unsigned(3 downto 0)
    );
end ALU3;

architecture calculation of ALU3 is
    SIGNAL b1, b2    : UNSIGNED(3 downto 0);
begin
    b1 <= B(7 downto 4);
    b2 <= B(3 downto 0);
    process (clock, reset)
    begin
        if reset = '0' then
            Result <= "0000"; -- Initialize to 0000
        else
            if (b1 < student_id) or (b2 < student_id) then
                Result <= "1111"; -- Assign 1111 ~ y
            else
                Result <= "0000"; -- Assign 0000 ~ n
            end if;
        end if;
    end process;
end calculation;

```

Figure 28: ALU 3 VHDL Code

Note: Since the function set assigned is requesting the same operation for every opcode, it would be redundant to use a case-when statement as used previously. Therefore we can simply remove this part of the code, and move on straight to the logic of the function. The opcode input remained present and connected within the circuit for completeness and modularity however. Another thing to note is that the format of the result has changed. This is because we will be

using a new iteration of the 7-segment display component to display the letters ‘y’ and ‘n’. The new 7-segment display’s inner-workings will be demonstrated below.

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

ENTITY sseg3 IS
PORT (
    bcd      : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    ResultLED: OUT STD_LOGIC_VECTOR(0 TO 6));
END sseg3 ;

ARCHITECTURE Behavior OF sseg3 IS
BEGIN
PROCESS (bcd) BEGIN

CASE bcd IS --abcdefg
WHEN "0000" => ResultLED <= ("1101010"); -- n
WHEN "1111" => ResultLED <= ("1000100"); -- y
WHEN OTHERS => ResultLED <= ("-----");
END CASE;
END PROCESS;
END Behavior;
```

Figure 29: New 7-Segment Display Iteration VHDL Code

In this 7-Segment display’s VHDL code, we simply have two possible outputs, which I set to be “0000” and “1111”, and each one corresponds to the letters ‘n’ and ‘y’ respectively to be displayed.

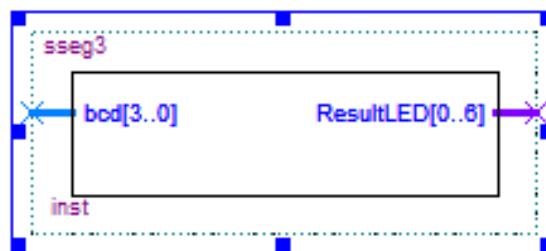


Figure 30: New 7-Segment Display Iteration Block Symbol

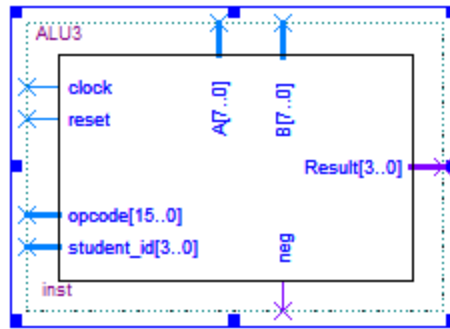


Figure 31: ALU 3 Block Symbol

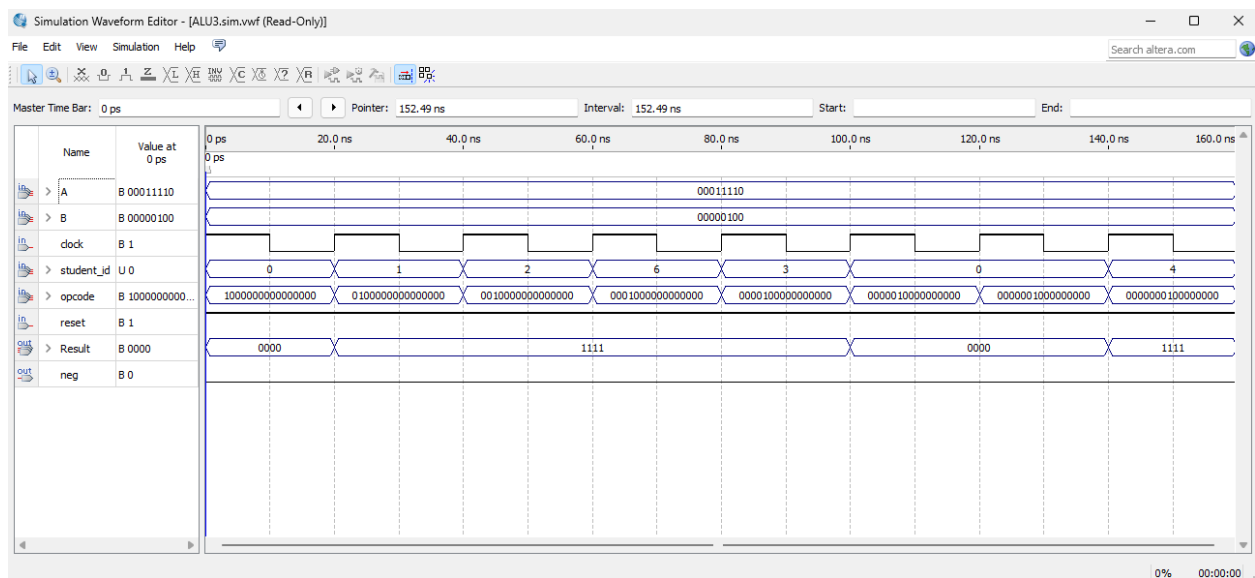


Figure 32: ALU 3 Waveform

Opcode	Student ID	B		Result
Binary	Decimal	Digit 1	Digit 2	Letter 'y' or 'n'
0000000000000001	0	0	4	n
0000000000000010	1	0	4	y
0000000000000100	2	0	4	y
0000000000001000	6	0	4	y
0000000000010000	3	0	4	y
0000000000100000	0	0	4	n

0000000001000000	0	0	4	n
0000000001000000	4	0	4	y

Table 12: ALU 3 Expected Outputs

The operation for this function set is fairly straightforward and does not require any handwritten work to verify that it is functioning correctly.

Combined ALU 3:

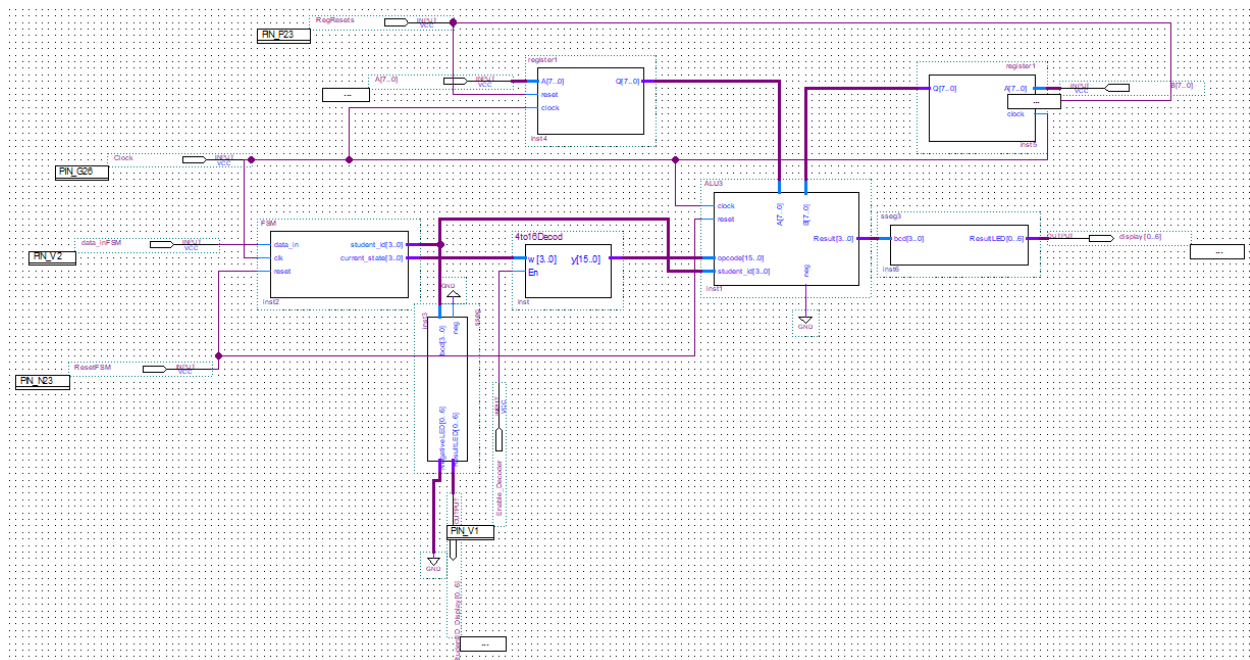


Figure 33: Combined ALU 3 Block Diagram

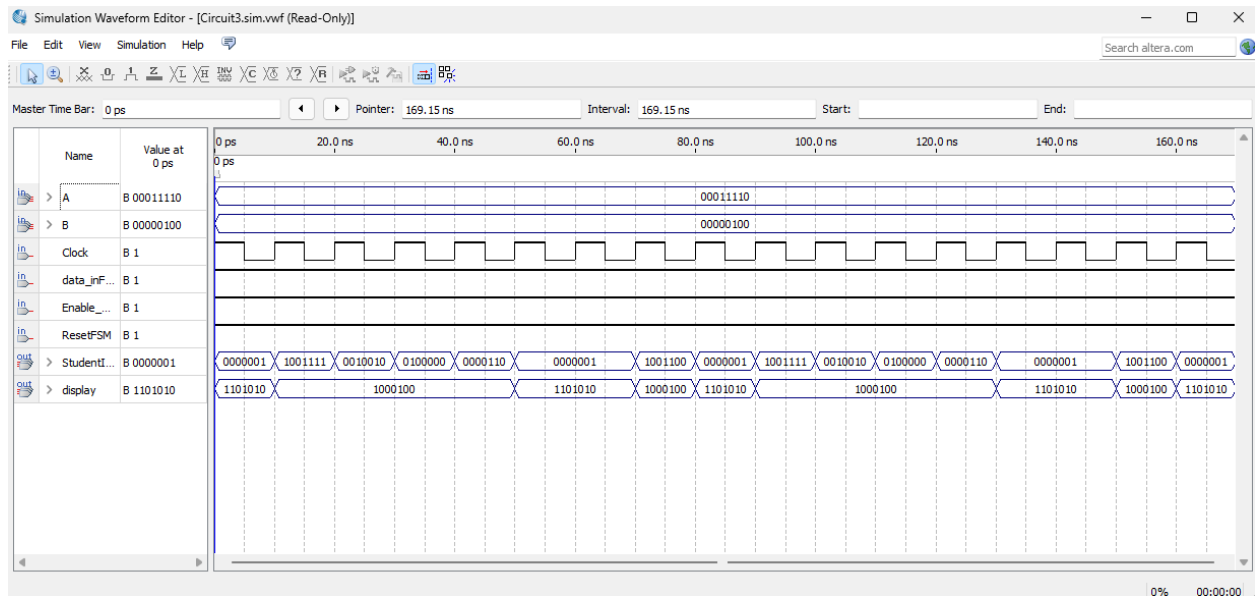


Figure 34: Combined ALU 3 Waveform

Operation (Display 'y' if one of the two digits of B is less than the student ID digit displayed)	Student ID (Decimal)	Expected Output ('y' or 'n')	Empirical Output ('y' or 'n')
1	0	n	
2	1	y	
3	2	y	
4	6	y	
5	3	y	
6	0	n	



7	0	n	
8	4	y	

Table 13: Combined ALU 3 Empirical Output

As observed, we can see that there is a perfect correlation between the empirical values, expected values, and the values provided by the waveform of Combined ALU 3. Thus we can soundly conclude that the implementation of all components in this circuit is correct and that the CPU functions correctly.

in	A[7]	Input	PIN_U4	1	B1_N0	PIN_U4	3.3-V LV...default		24mA (default)	
in	A[6]	Input	PIN_U3	1	B1_N0	PIN_U3	3.3-V LV...default		24mA (default)	
in	A[5]	Input	PIN_T7	1	B1_N0	PIN_T7	3.3-V LV...default		24mA (default)	
in	A[4]	Input	PIN_P2	1	B1_N0	PIN_P2	3.3-V LV...default		24mA (default)	
in	A[3]	Input	PIN_P1	1	B1_N0	PIN_P1	3.3-V LV...default		24mA (default)	
in	A[2]	Input	PIN_N1	2	B2_N1	PIN_N1	3.3-V LV...default		24mA (default)	
in	A[1]	Input	PIN_A13	4	B4_N1	PIN_A13	3.3-V LV...default		24mA (default)	
in	A[0]	Input	PIN_B13	4	B4_N1	PIN_B13	3.3-V LV...default		24mA (default)	
in	B[7]	Input	PIN_C13	3	B3_N0	PIN_C13	3.3-V LV...default		24mA (default)	
in	B[6]	Input	PIN_AC13	8	B8_N0	PIN_AC13	3.3-V LV...default		24mA (default)	
in	B[5]	Input	PIN_AD13	8	B8_N0	PIN_AD13	3.3-V LV...default		24mA (default)	
in	B[4]	Input	PIN_AF14	7	B7_N1	PIN_AF14	3.3-V LV...default		24mA (default)	
in	B[3]	Input	PIN_AE14	7	B7_N1	PIN_AE14	3.3-V LV...default		24mA (default)	
in	B[2]	Input	PIN_P25	6	B6_N0	PIN_P25	3.3-V LV...default		24mA (default)	
in	B[1]	Input	PIN_N26	5	B5_N1	PIN_N26	3.3-V LV...default		24mA (default)	
in	B[0]	Input	PIN_N25	5	B5_N1	PIN_N25	3.3-V LV...default		24mA (default)	
in	Clock	Input	PIN_G26	5	B5_N0	PIN_G26	3.3-V LV...default		24mA (default)	
in	data_inFSM	Input	PIN_V2	1	B1_N0	PIN_V2	3.3-V LV...default		24mA (default)	
out	display[0]	Output	PIN_AF10	8	B8_N0	PIN_AF10	3.3-V LV...default		24mA (default)	
out	display[1]	Output	PIN_AB12	8	B8_N0	PIN_AB12	3.3-V LV...default		24mA (default)	
out	display[2]	Output	PIN_AC12	8	B8_N0	PIN_AC12	3.3-V LV...default		24mA (default)	
out	display[3]	Output	PIN_AD11	8	B8_N0	PIN_AD11	3.3-V LV...default		24mA (default)	
out	display[4]	Output	PIN_AE11	8	B8_N0	PIN_AE11	3.3-V LV...default		24mA (default)	
out	display[5]	Output	PIN_V14	8	B8_N0	PIN_V14	3.3-V LV...default		24mA (default)	
out	display[6]	Output	PIN_V13	8	B8_N0	PIN_V13	3.3-V LV...default		24mA (default)	
in	Enable_Decoder	Input	PIN_V1	1	B1_N0	PIN_V1	3.3-V LV...default		24mA (default)	
in	RegResets	Input	PIN_P23	6	B6_N0	PIN_P23	3.3-V LV...default		24mA (default)	
in	ResetFSM	Input	PIN_N23	5	B5_N1	PIN_N23	3.3-V LV...default		24mA (default)	
out	StudentID_Display[0]	Output	PIN_U9	1	B1_N0	PIN_U9	3.3-V LV...default		24mA (default)	
out	StudentID_Display[1]	Output	PIN_U1	1	B1_N0	PIN_U1	3.3-V LV...default		24mA (default)	
out	StudentID_Display[2]	Output	PIN_U2	1	B1_N0	PIN_U2	3.3-V LV...default		24mA (default)	
out	StudentID_Display[3]	Output	PIN_T4	1	B1_N0	PIN_T4	3.3-V LV...default		24mA (default)	
out	StudentID_Display[4]	Output	PIN_R7	1	B1_N0	PIN_R7	3.3-V LV...default		24mA (default)	
out	StudentID_Display[5]	Output	PIN_R6	1	B1_N0	PIN_R6	3.3-V LV...default		24mA (default)	
out	StudentID_Display[6]	Output	PIN_T3	1	B1_N0	PIN_T3	3.3-V LV...default		24mA (default)	

Figure 35: Combined ALU 3 Pin Planner

Conclusion:

In conclusion, the successful design and implementation of three distinct ALUs was achieved in this lab and demonstrates a practical application of digital systems concepts such as finite state machines (FSMs), decoders, latches, and arithmetic logic units (ALUs). Each ALU was designed to handle specific operations, and the components were integrated seamlessly to form a functional CPU. The results of each ALU were verified through truth tables, waveform simulations, and empirical testing, ensuring that all expected outputs were achieved with precision. This lab highlighted the importance of modular design in digital systems, where individual components can be developed, tested, and integrated effectively in numerous different systems. It also accentuates the significance of thorough testing methods to validate functionality, from initial simulations to hardware implementation. These exercises not only deepen the understanding of CPU architecture but also provide essential experience in debugging and complex system design.