




Department of Physics

Course Number	COE 328
Course Title	Digital Systems
Semester/Year	Spring 2022
Instructor	Arghavan Asad
TA Name	Yasaman Ahamdiadli

Lab/Tutorial Report No.	Lab 6
-------------------------	-------

Report Title	Design of a Simple General-Purpose Processor
--------------	--

Section No.	3
Group No.	N/A
Submission Date	Tuesday, June 28 th , 2022
Due Date	Tuesday, June 28 th , 2022

Student Name	Student ID	Signature*
Sanjay Sivapragasam	501045929	

**By signing above, you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at:*

<http://www.ryerson.ca/content/dam/senate/policies/pol60.pdf>

Table of Contents

Contents	2
Introduction	3
Components:	
• Latch (Storage Unit)	3
• 4:16 Decoder	5
• Finite State Machine (FSM)	7
• 7-Segment Display	11
• Modified 7-Segment Display (ALU part 3)	14
Arithmetic Logical Unit (ALU):	
• ALU for Part 1	16
• ALU for Part 2	21
• ALU for Part 3	25
Conclusion	30

[This part of page is left blank intentionally]

Introduction

The main purpose of this lab was to design and construct a general-purpose processor utilizing all the knowledge obtained from the completion of previous lab experiments. To accomplish this, 4 distinct components from both combinational and sequential circuits were constructed including: an Arithmetic and Logical Unit (ALU), 2 latches, and a control unit comprised of a finite state machine (FSM) and 4:16 Decoder. Mealy logic was implemented in this FSM. All the components were built in a VHDL environment on a FPGA board. Overall, this simple processor performs various operations to two 8-bit inputs based on the current state of the finite state machine (FSM) and a 16-bit input from the decoder (control unit). All the results were displayed via the connected 7-segment displays and consequently viewed on waveforms.

Components

I. Latch (Latch 1 and Latch 2)

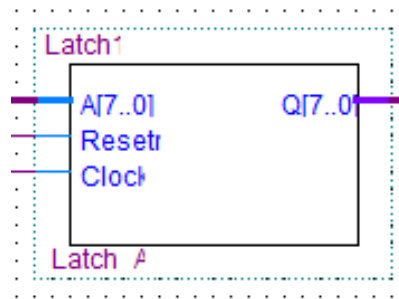
The latch is the main component of the storage unit (Register) which takes in a binary number for input and outputs it each cycle. As the name suggested, they “latch onto” the inputted information and delay the changing of state for the ALU to perform its operations. The circuit utilizes 2 basic latches, one for each respective binary number from the 2 inputs ‘A’ and ‘B’ used in the ALU, and both are D flip-flops as they are controlled by a clock transition (changes state at each rising edge).

VHDL code:

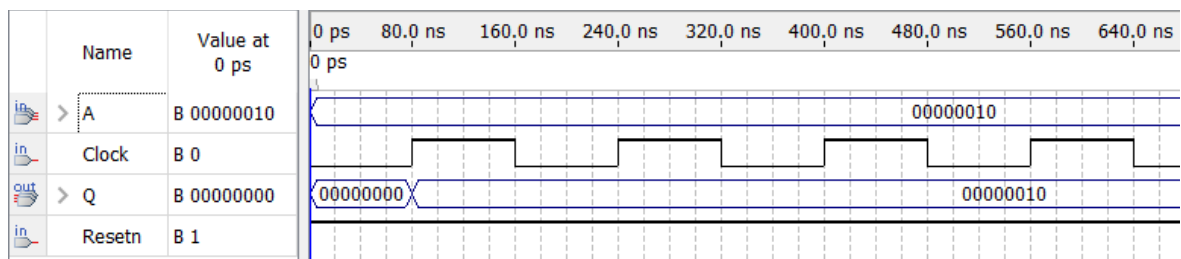
```

1  LIBRARY ieee;
2  Use ieee.std_logic_1164.all;
3  ENTITY Latch1 IS
4  PORT ( A : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- 8 bit A input
5        |      Resetn, Clock:IN STD_LOGIC; -- 1 bit clock and reset inputs
6        |      Q: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)); --
7  END Latch1;
8  ARCHITECTURE Behavior OF Latch1 IS
9  BEGIN
10     PROCESS (Resetn, Clock)
11     BEGIN
12         IF Resetn = '0' THEN
13             Q<="00000000";
14         ELSIF Clock'EVENT AND Clock = '1' THEN
15             Q<=A;
16         END IF;
17     END PROCESS;
18 END Behavior;
```

Circuit diagram:



Waveform for Latch:



In this waveform, a sample input of 2 in 8-bit binary was used. When the clock is 0, the output is 0, and when the clock is 1, the output matches the input. This accurately depicts the truth table seen below.

Truth table for Latch:

Clk	D	Q (t+1)
0	x	Q(t)
1	0	0
1	1	1

II. 4:16 Decoder

The 4:16 decoder takes in a 4-bit input and gives out a unique 16-bit output. This 4-bit input is from the state output of the FSM, as they are both sub-components that make up the overall control unit of the processor.

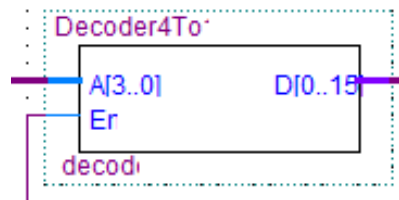
VHDL code:

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ENTITY Decoder4To16 IS
4  PORT (A : IN STD_LOGIC_VECTOR(3 DOWNT0 0);
5       En : IN STD_LOGIC; -- enable
6       D : OUT STD_LOGIC_VECTOR(0 TO 15));
7  END Decoder4To16;
8  ARCHITECTURE Behavior of Decoder4To16 IS
9  SIGNAL EnA : STD_LOGIC_VECTOR(4 DOWNT0 0);
10 BEGIN
11   EnA <= En & A;
12   WITH EnA SELECT
13   D<= "0000000000000001" WHEN "10000",
14       "0000000000000010" WHEN "10001",
15       "0000000000000100" WHEN "10010",
16       "0000000000001000" WHEN "10011",
17       "0000000000010000" WHEN "10100",
18       "0000000000100000" WHEN "10101",
19       "0000000001000000" WHEN "10110",
20       "0000000010000000" WHEN "10111",
21       "0000000100000000" WHEN "11000",
22       "0000001000000000" WHEN "11001",
23       "0000010000000000" WHEN "11010",
24       "0000100000000000" WHEN "11011",
25       "0001000000000000" WHEN "11100",
26       "0010000000000000" WHEN "11101",
27       "0100000000000000" WHEN "11110",
28       "1000000000000000" WHEN "11111",
29       "0000000000000000" WHEN OTHERS;
30 END Behavior;

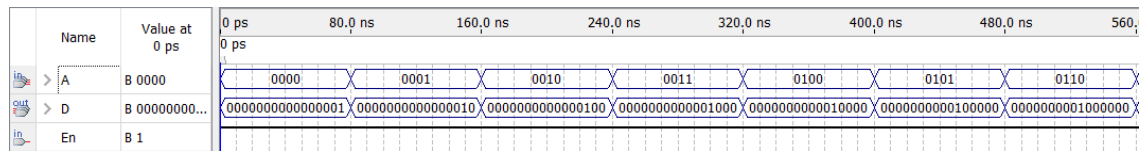
```

Circuit Diagram of 4:16 Decoder:

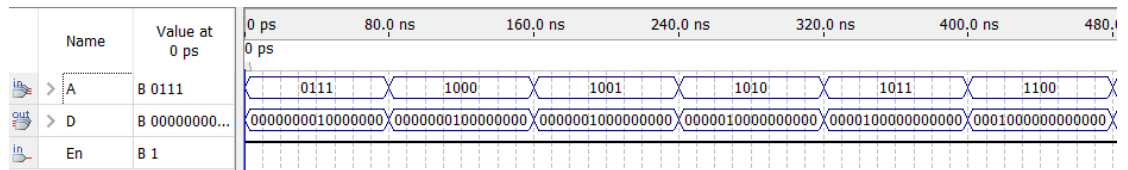


Waveform of 4:16 Decoder:

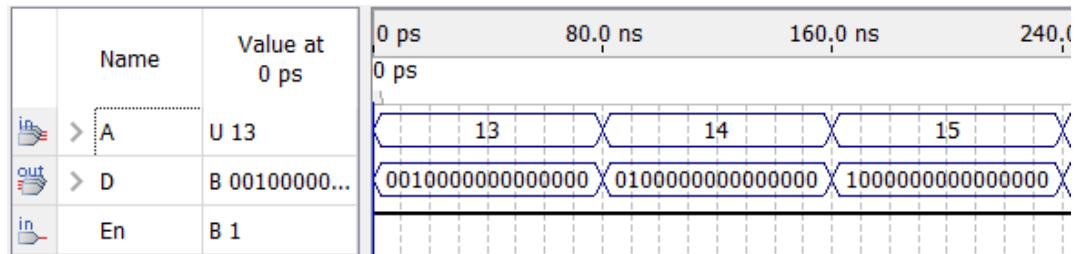
Inputs 0-6:



Inputs 7-12:



Inputs 13-15:



Truth table for 4:16 Decoder:

A (4-bit input)	D (16-bit output)
0000	0000000000000001
0001	0000000000000010
0010	0000000000000100
0011	0000000000001000
0100	0000000000010000
0101	0000000000100000
0110	0000000001000000
0111	0000000010000000
1000	0000000100000000
1001	0000001000000000
1010	0000010000000000
1011	0000100000000000
1100	0001000000000000
1101	0010000000000000
1110	0100000000000000
1111	1000000000000000

III. FSM

The FSM implemented in this processor is a Mealy machine. It cycles through the different states from S_0 to S_8 . This is achieved as when the data input is 1, it moves from the current state to the next state in a consecutive order. When it reaches the final state of S_8 , it will transition back to S_0 and essentially loops between these 9 different states. As each state is reached, it outputs a respective digit of my student ID until all 9 digits are outputted in chronological order. For example, the third state S_3 would give the third digit of my student ID (1) which is easily identifiable from the waveform. As a result, the output values are determined by both the current state and the current inputs. The states are also used as inputs for the 4:16 decoder, which consequently act as the microcode's for the ALU. This is because the FSM and 4:16 decoder make up the overall control unit of this processor. Therefore, the main purpose of the FSM is that it will decide the pattern of the controller sequence, functioning similarly to an up counter while feeding this data to the 4:16 decoder.

VHDL code:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity lab6fsm is
4  | port
5  | (
6  |   clk      : in std_logic;
7  |   data_in   : in std_logic;
8  |   reset     : in std_logic;
9  |   student_id : out std_logic_vector(3 downto 0);
10 |   current_state: out std_logic_vector(3 DOWNTO 0) );
11 | end entity;
12 | architecture fsm of lab6fsm is
13 | type state_type is (s0, s1, s2, s3, s4, s5, s6,
14 |   s7, s8);
15 | signal yfsm : state_type;
16 | begin
17 | process (clk, reset)
18 | begin
19 |   if reset = '1' then
20 |     yfsm <= s0;
21 |   elsif (clk 'EVENT AND clk = '1') then
22 |     case yfsm is
23 |     when s0 =>
24 |       if data_in = '1' then
25 |         yfsm <= s1;
26 |       else yfsm <= s0;
27 |       end if;
28 |     when s1 =>
29 |       if data_in = '1' then
30 |         yfsm <= s2;
31 |       else yfsm <= s1;
32 |     end if;
33 |
34 |     when s2 =>
35 |       if data_in = '1' then
36 |         yfsm <= s3;
37 |       else yfsm <= s2;
38 |       end if;
39 |     when s3 =>
40 |       if data_in = '1' then
41 |         yfsm <= s4;
42 |       else yfsm <= s3;
43 |       end if;
44 |     when s4 =>
45 |       if data_in = '1' then
46 |         yfsm <= s5;
47 |       else yfsm <= s4;
48 |       end if;
49 |     when s5 =>
50 |       if data_in = '1' then
51 |         yfsm <= s6;
52 |       else yfsm <= s5;
53 |       end if;
54 |     when s6 =>
55 |       if data_in = '1' then
56 |         yfsm <= s7;
57 |       else yfsm <= s6;
58 |       end if;
59 |     when s7 =>
60 |       if data_in = '1' then
61 |         yfsm <= s8;
62 |       else yfsm <= s7;
63 |       end if;
64 |     when s8 =>
65 |       if data_in = '1' then
66 |         yfsm <= s0;
67 |       else yfsm <= s8;
68 |       end if;
69 |     end case;
70 |   end if;
71 | end process;
72 |
73 | student_id <= student_id(3 downto 0) & yfsm(3 downto 0);
74 |
75 | end architecture;

```

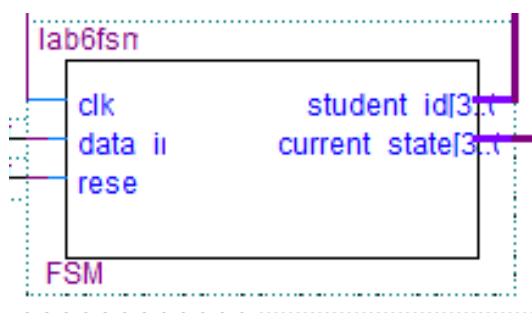


```

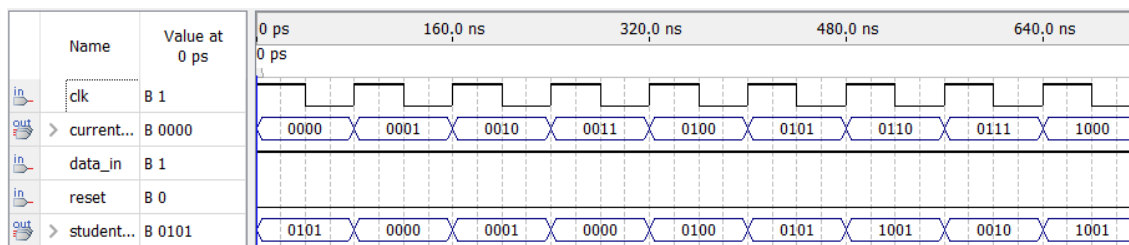
67  if data_in = '1' then
68  | yfsm <= s8;
69  else yfsm <= s7;
70  end if;
71  |
72  when s8 =>
73  if data_in = '1' then
74  | yfsm <= s0;
75  else yfsm <= s8;
76  end if;
77  end case;
78  end if;
79  end process;
80  |
81  process(yfsm, data_in)
82  begin
83  case yfsm is
84  when s0 => -- what s8 points to
85  student_id <="0101"; -- 5
86  current_state <= "0000";
87  when s1=> -- what s0 points to
88  student_id <="0000"; -- 0
89  current_state <= "0001";
90  when s2 => -- what s1 points to
91  student_id <="0001"; -- 1
92  current_state <= "0010";
93  when s3 => -- what s2 points to
94  student_id <="0000"; -- 0
95  current_state <= "0011";
96  when s4 => -- what s3 points to
97  student_id <="0100"; -- 4
98  current_state <= "0100";
99  when s5 => -- what s4 points to
100 student_id <="0101"; -- 5
101 current_state <= "0101";
102 when s6 => -- what s5 points to
103 student_id <="1001"; -- 9
104 current_state <= "0110";
105 when s7 => -- what s6 points to
106 student_id <="0010"; -- 2
107 current_state <= "0111";
108 when s8 => -- what s7 points to
109 student_id <="1001"; -- 9
110 current_state <= "1000";
111
112 end case;
113 end process;
114 end architecture;

```

Circuit diagram:



Waveform:



Truth table:

Present State	Next State		Output Student_id
	data_in = 0	data_in = 1	
0000	0000	0001	0101
0001	0001	0010	0000
0010	0010	0011	0001
0011	0011	0100	0000
0100	0100	0101	0100
0101	0101	0110	0101
0110	0110	0111	1001
0111	0111	1000	0010
1000	1000	0000	1001

In the truth table above, it is evident that when the data in is a 1, the current state always points to the next state and repeats when the eighth state points to S_0 . The student ID column above reads my student number (501045929) in binary.

IV. 7-Segment

The 7-segment display code utilized was the same code that worked successfully in lab experiment 3. It effectively displays values from 0-15 in hexadecimal code and accounts for a scenario in which the resultant output is a negative value. Multiple 7-segment displays were used as the inputs varied from the student ID from the FSM to the values of the ALU results. The ALU required two 7-segments to display the values as the ALU output is 8-bit, but the 7-segment uses 4-bit input.

VHDL code:

```

1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;
3  ENTITY sseg IS
4  PORT ( bcd : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5        neg : IN STD_LOGIC ;
6        leds : OUT STD_LOGIC_VECTOR (0 TO 6);
7        ledss : OUT STD_LOGIC_VECTOR(0 TO 6) ) ;
8  END sseg ;
9  ARCHITECTURE Behavior OF sseg IS
10 BEGIN
11   PROCESS ( bcd )
12   BEGIN
13     CASE bcd IS
14       WHEN "0000" => leds <= "1111110" ; -- 0
15       WHEN "0001" => leds <= "0110000" ; -- 1
16       WHEN "0010" => leds <= "1101101" ; -- 2
17       WHEN "0011" => leds <= "1111001" ; -- 3
18       WHEN "0100" => leds <= "0110011" ; -- 4
19       WHEN "0101" => leds <= "1011011" ; -- 5
20       WHEN "0110" => leds <= "1011111" ; -- 6
21       WHEN "0111" => leds <= "1110000" ; -- 7
22       WHEN "1000" => leds <= "1111111" ; -- 8
23       WHEN "1001" => leds <= "1111011" ; -- 9
24       -- numbers 10 to 15 in HEX
25       WHEN "1010" => leds <= "1110111" ; -- A
26       WHEN "1011" => leds <= "0011111" ; -- b
27       WHEN "1100" => leds <= "1001110" ; -- C
28       WHEN "1101" => leds <= "0111101" ; -- d
29       WHEN "1110" => leds <= "1001111" ; -- E
30       WHEN "1111" => leds <= "1000111" ; -- F
31     END CASE ;
32   END PROCESS ;
33   PROCESS (neg)
34   BEGIN
35     IF (neg = '1') THEN
36       ledss <= "0000001";
37     ELSE
38       ledss <= "0000000";
39     END IF;
40   END PROCESS;
41 END Behavior ;

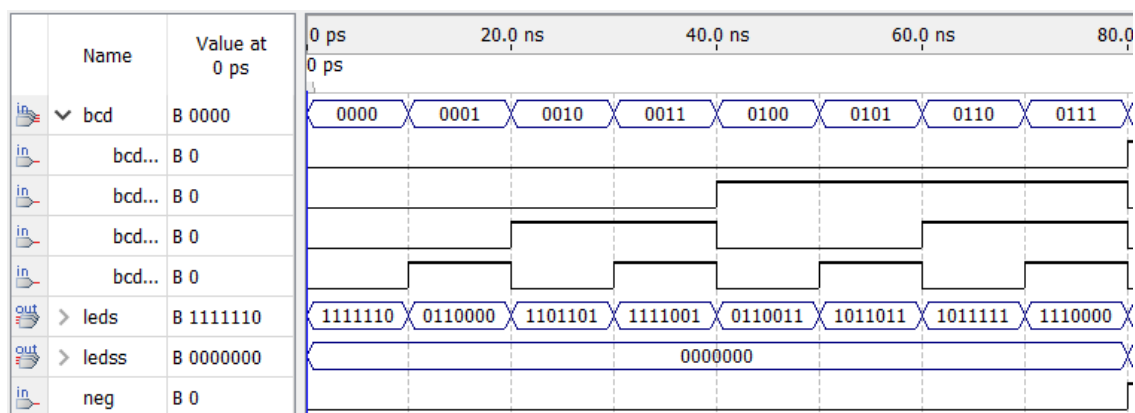
```

Circuit diagram:

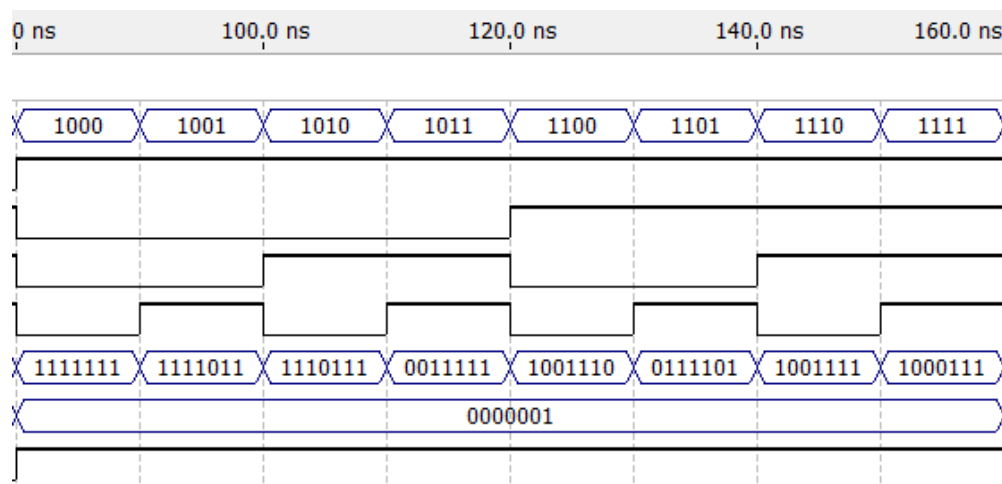


Waveform:

From 0 to 7:



From 8 to F:



The image of the second half of the waveform shows that the output of ledss is 0000001 because the neg input is set to high (1). This is done to show that the neg input functions correctly, and not that the remaining values (A-F) are specifically negative. Overall, this shows every component of the 7-segment works correctly – regardless of there being a positive or negative value.

7-segment truth table for positive input:

bcd (input)	leds (abcdefg)
0000	1111110
0001	0110000
0010	1101101
0011	1111001
0100	0110011
0101	1011011
0110	1011111
0111	1110000
1000	1111111
1001	1110011
1010	1110111 (A)
1011	0011111 (b)
1100	1001110 (C)
1101	0111101 (d)
1110	1001111 (E)
1111	1000111 (F)

7-segment truth table for negative input:

neg	ledss (abcdefg)
0	0000000
1	0000001

V. Modified 7-Segment (used in ALU part 3)

- i) For each microcode instruction, 'y' if one of the 2 digits of B are less than FSM output (**student_id**) and 'n' otherwise. Use the microcode instruction from part 1 of the lab.

For the modified 7-segment display, the main change was including new input and output variables to determine if the displayed output is a 'y' or 'n' based on the condition seen in the screenshot above.

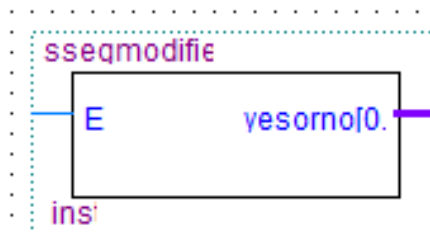
VHDL code:

```

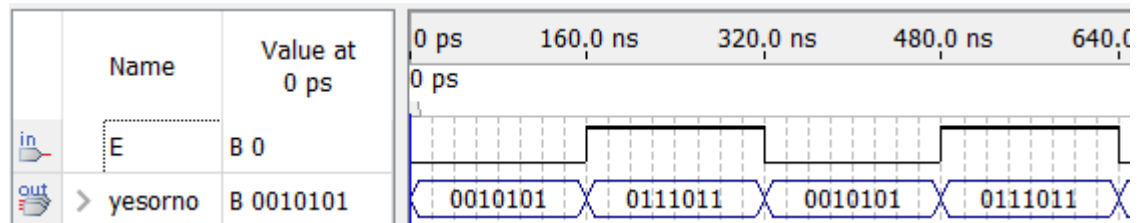
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;
3  ENTITY ssegmodified IS
4  PORT ( --bcd : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5        E : IN STD_LOGIC ;
6        --leds : OUT STD_LOGIC_VECTOR (0 TO 6);
7        yesorno : OUT STD_LOGIC_VECTOR(0 TO 6));
8  END ssegmodified ;
9  ARCHITECTURE Behavior OF ssegmodified IS
10 BEGIN
11 |
12 PROCESS (E)
13 |   BEGIN
14 |   IF (E = '1') THEN
15 |     yesorno <= "0111011"; -- yes (y)
16 |   ELSE
17 |     yesorno <= "0010101"; -- no (n)
18 |   END IF;
19 |   END PROCESS;
20 END Behavior ;

```

Circuit Diagram:



Waveform:



In the waveform above, when E is set to 1 it outputs a 'y' and a 'n' when set 0 respectively. This used to output the yes or no condition from the ALU for part 3, alongside the original 7-segment display used in previously, specifically for the student ID FSM output only in part 3.

Truth table:

E (input)	yesorno (abcdefg)
1	0111011 ('y')
0	0010101 ('n')

[This part of page is left blank intentionally]

Arithmetic Logical Unit (ALU) – Part 1

The main **purpose** of the ALU component is to perform the required operations based on the states of the FSM. In this part, these 9 total operations include addition, subtraction, and all Boolean operations. It was constructed by adding the necessary new code to the skeleton code provided in the lab manual. In total it has 5 inputs as seen below:

1. Clock
2. A
3. B
4. Student_id
5. OP

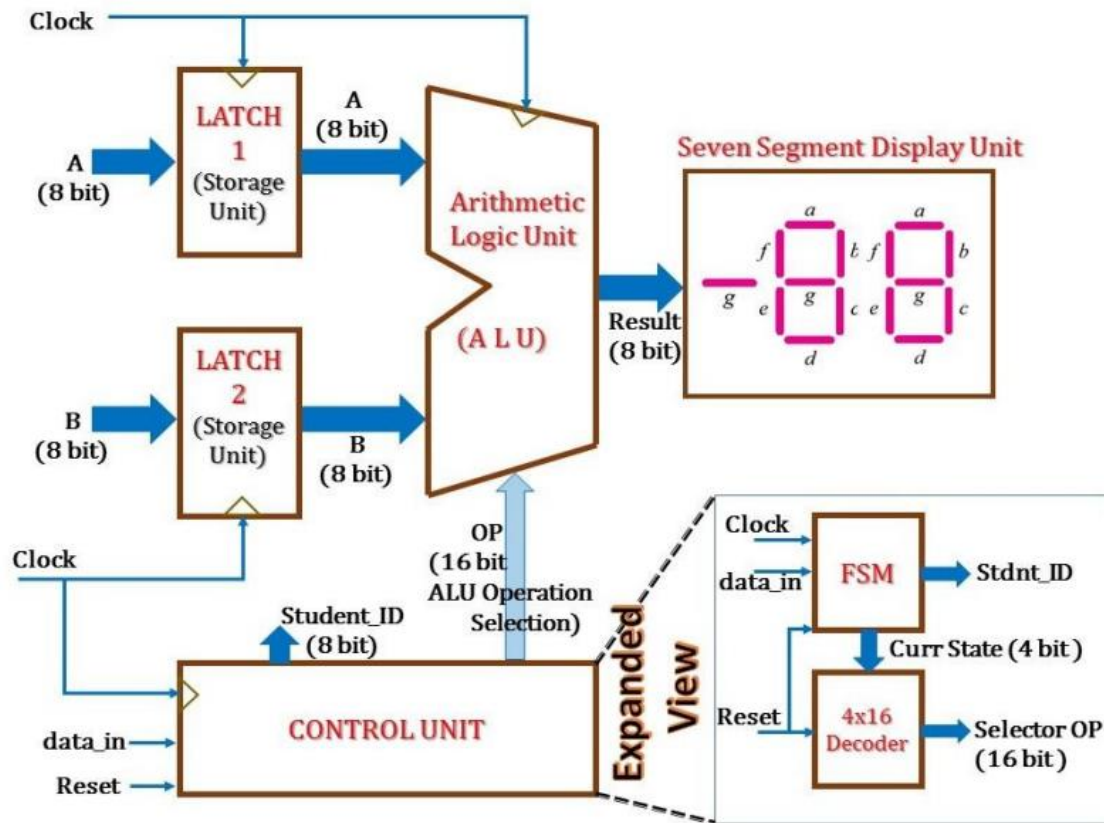
The clock input is shared between all components except the 7-segment display and 4:16 decoder as they are combinational circuits whereas the rest are sequential. This clock input alternates between 0 and 1, and when a rising edge occurs (input changes from 0 to 1) the ALU will check the OP input which is the output of the 4:16 decoder. This will correspond to the case statement and perform the respective operation on inputs A and B according to the table in the lab manual. For all the remaining sections of the lab experiment involving the ALU including part 1, the inputs used will be based on the last 4 digits of my student number: 5929. Therefore, input A will be equal to 59 and input B will be equal to 29 respectively. They are the inputs of the storage unit, only when the input data_in is 1. The last input of student_id serves no purpose in this part and therefore is grounded in the block diagram.

The ALU has 3 outputs:

1. Neg
2. R1
3. R2

The neg refers to a negative output so the 7-segment display can light up segment g if neg is equal to 1 when the value is a negative number. Since the 7-segment display is 4-bit input but the ALU works in 8-bit, R1 and R2 are used together to split the result into two, 4-bit outputs. Due to this, each one will go to a separate 7-segment display to display in binary output of each half in hexadecimal, but when combined it reads the expected 8-bit output.

Block diagram of the GPU from lab manual:



ALU core operations (microcode's generated by decoder) for problem 1:

Function #	Microcode	Boolean Operation / Function
1	0000000000000001	$\text{sum}(A, B)$
2	0000000000000010	$\text{diff}(A, B)$
3	0000000000000100	\overline{A}
4	0000000000001000	$\overline{A \cdot B}$
5	0000000000010000	$\overline{A + B}$
6	0000000000100000	$A \cdot B$
7	0000000001000000	$A \oplus B$
8	0000000010000000	$A + B$
9	0000000100000000	$\overline{A \oplus B}$

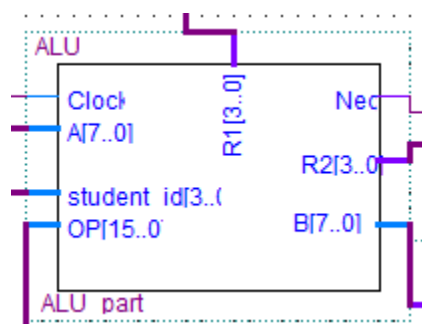
VHDL code:

```

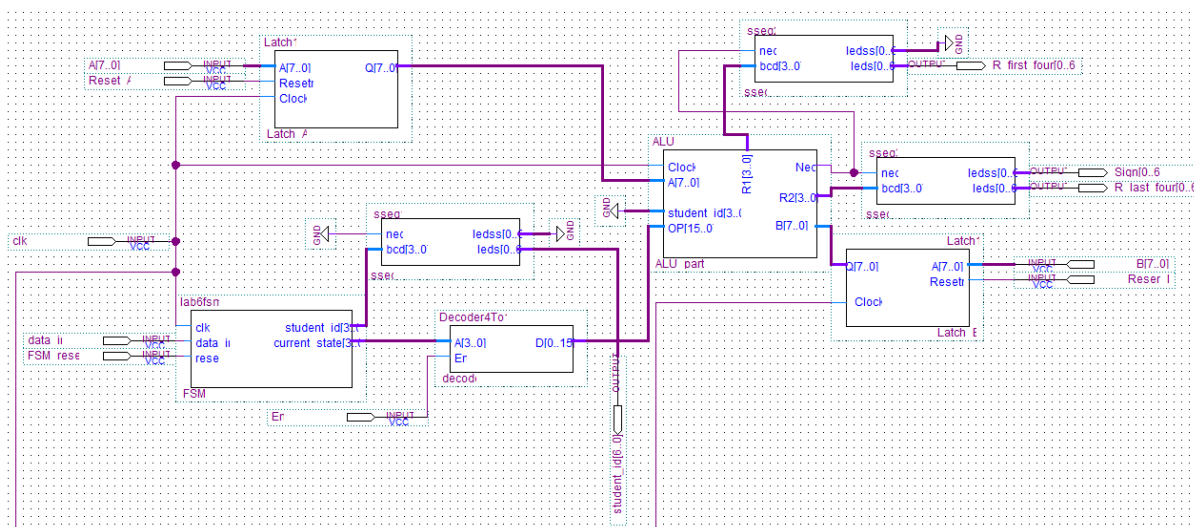
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5  ENTITY ALU IS
6  port (Clock: in std_logic; --input clock signal
7        A,B : in unsigned(7 downto 0); --8-bit inputs from latches A and B
8        student_id : in unsigned(3 downto 0); --4-bit student id from FSM
9        OP: in unsigned(15 downto 0);--16-bit selector for operation from decoder
10       Neg: out std_logic;--is the result negative? set -ve bit output
11       R1: out unsigned (3 downto 0);--lower 4-bits of 8-bit result output
12       R2: out unsigned (3 downto 0);--higher 4-bits of 8-bit result output
13  end ALU;
14  ARCHITECTURE calculation of ALU IS --temporary signal declaration
15  |   signal Reg1, Reg2, Result: unsigned (7 downto 0);--:= (others=>'0');
16  |   signal Reg4 : unsigned (0 to 7);
17  |   begin
18  |       Reg1 <= A; --temp storing A in Reg1 local variable
19  |       Reg2 <= B; --temp storing B in Reg2 local variable
20  |   process(Clock,OP)
21  |   begin
22  |       if(rising_edge (Clock)) Then -- calculation @ positive edge of clock
23  |       case OP is
24  |       When "0000000000000001"=>--!!!!!!!
25  |           Result <= (Reg1 + Reg2); --addition
26  |       When "0000000000000010"=>
27  |       if(Reg2>Reg1) Then
28  |           Result <= (Reg1 + (NOT Reg2 +1));
29  |           Neg<='1'; -- setting neg bit
30  |       else
31  |           Result <= (Reg1 - Reg2) ;
32  |           Neg<='0';
33  |       end if;
34  |       When "0000000000000100"=>
35  |           Result <= (NOT Reg1); --inverse
36  |           --Neg<='0';
37  |       When "00000000000001000"=>
38  |           Result <= (Reg1 NAND Reg2); --boolean NAND
39  |           --Neg<='0';
40  |       When "000000000000010000"=>
41  |           Result <= (Reg1 NOR Reg2); --boolean NOR
42  |           --Neg<='0';
43  |       When "0000000000000100000"=>
44  |           Result <= (Reg1 AND Reg2); --boolean AND
45  |           --Neg<='0';
46  |       When "00000000000001000000"=>
47  |           Result <= (Reg1 XOR Reg2); --boolean OR
48  |           --Neg<='0';
49  |       When "000000000000010000000"=>
50  |           Result <= (Reg1 OR Reg2); --boolean XOR
51  |           --Neg<='0';
52  |       When "0000000000000100000000"=>
53  |           Result <= (Reg1 XNOR Reg2); --boolean XNOR
54  |           --Neg<='0';
55  |       When Others =>
56  |           -- don't care, do nothing
57  |           Result <="-----" ;
58  |       end case;
59  |   end if;
60  | end process;
61  | R1 <= Result(3 downto 0); --7-seg is only 4-bits
62  | R2 <= Result(7 downto 4); --so split 8-bit
63  | end calculation;

```

Circuit diagram of ALU component (part 1):

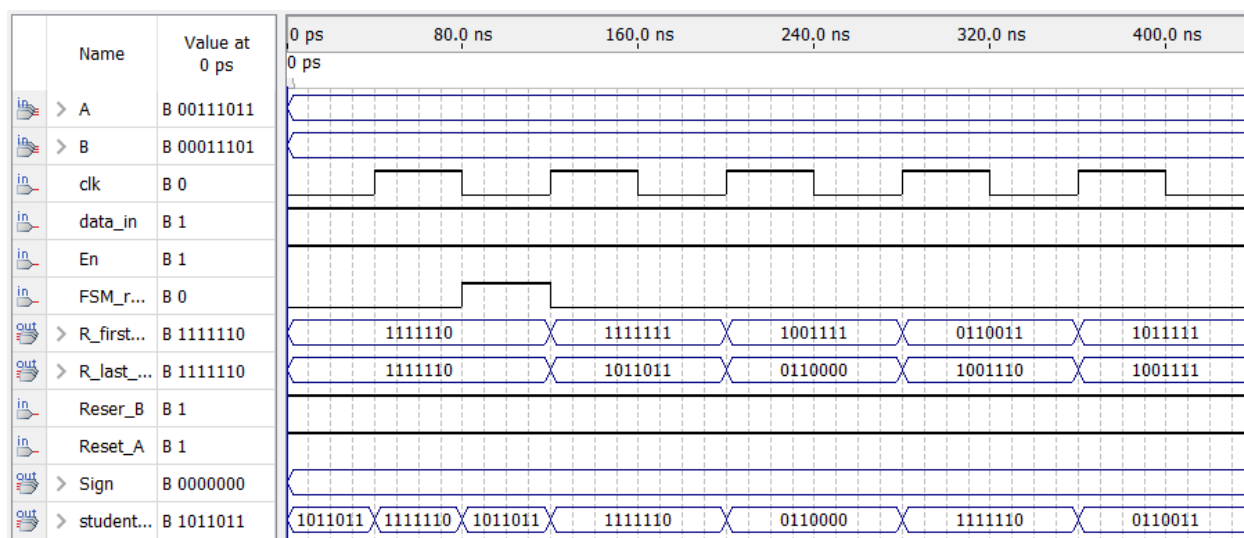


Circuit Diagram of ALU part 1:

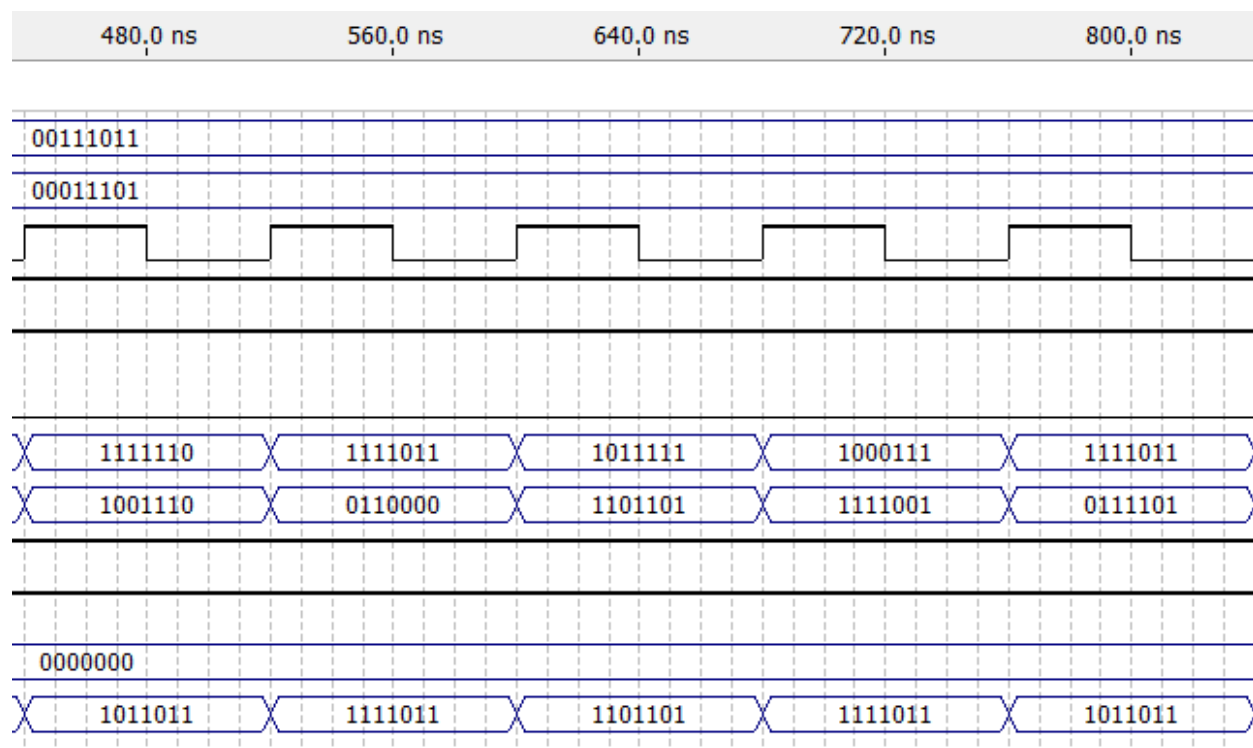


Waveform of ALU part 1:

First half of waveform:



Second half of waveform:



Arithmetic Logical Unit (ALU) – Part 2

The main **purpose** of the ALU in part 2 is essentially the same as part 1. The ALU will perform various operations based on the states of the FSM and receive the same inputs of A (59) and B (29) from the latches. The last 4 digits of my student number (5929) calculated with **mod 8** returns a 1. Therefore, problem set **a)** was selected and can be seen below.

ALU core operations (microcode's generated by decoder) for problem 2:

Function #	Operation / Function
1	Increment A by 2
2	Shift B to right by two bits, input bit = 0 (SHR)
3	Shift A to 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 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
9	Produce all high bits on the output

These specific modifications will result in brand new results in the waveform as the ALU core and its functionalities will be completely different, despite not changing the other components of the processor.

In part 2, the ALU has the same exact inputs and outputs seen below:

- Inputs
 - Clock
 - A
 - B
 - Student_id
 - OP
- Outputs
 - Neg
 - R1
 - R2

Similarly explained in part 1, inputs A and B are the 8-bit inputs stored in the latches when data_in remains 1. The clock input is shared between all sequential circuit components including the FSM, latches, and ALU. When a rising edge occurs (0 to 1), the OP input determines which operation to perform based on the microcode since OP is the output of the 4:16 decoder.

Student_id has no purpose in this part and therefore it is grounded in the block diagram.

Moreover, R1 and R2 represent 2 halves of the final outputs from the ALU (bits 7-4 and 3-0).

This is because the 7-segment display requires a 4-bit input. Lastly, when neg is 1, it refers to a negative value and ensures segment g will be lit to represent a negative sign.

VHDL code:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5  --use IEEE.STD_LOGIC_ARITH.ALL;
6  ENTITY ALUpart2modified IS
7  port (Clock: in std_logic; --input clock signal
8  A,B : in unsigned(7 downto 0); --8-bit inputs from latches A and B
9  student_id : in unsigned(3 downto 0); --4-bit student id from FSM
10 OP: in unsigned(15 downto 0);--16-bit selector for operation from decoder
11 Neg: out std_logic;--is the result negative? set -ve bit output
12 R1: out unsigned (3 downto 0);--lower 4-bits of 8-bit result output
13 R2: out unsigned (3 downto 0);--higher 4-bits of 8-bit result output
14 end ALUpart2modified;
15 ARCHITECTURE calculation of ALUpart2modified IS --temporary signal declaration
16     signal Reg1, Reg2, Result: unsigned (7 downto 0) := (others=>'0');
17     signal Reg4 : unsigned (0 to 7);
18     begin
19     Reg1 <= A; --temp storing A in Reg1 local variable
20     Reg2 <= B; --temp storing B in Reg2 local variable
21     process(Clock,OP)
22     begin
23         if(rising_edge (Clock)) Then -- calculation @ positive edge of clock
24         case OP is
25             When "0000000000000001"=>--1
26                 Result <= 2 + Reg1;--increment by 2
27                 Neg <='0';
28             When "0000000000000010"=>--2
29                 -- input bit = 0
30                 Result <= shift_right(unsigned(Reg2), 2); -- shift to right by 2
31             When "0000000000000100"=>--3
32                 Result <= shift_right(unsigned(Reg1), 4); -- shift to right by 4
33                 Result(7) <= '1'; -- input bit = 1
34
35                 Result(6) <= '1';
36                 Result(5) <= '1';
37                 Result(4) <= '1';
38             When "0000000000001000"=>--4
39                 if (Reg2>Reg1) Then
40                     Result <= Reg1;
41                 else
42                     Result <= Reg2;
43                 end if;
44                 Neg<='0';
45             When "0000000000010000"=>--5
46                 Result <= Reg4; --ROR A by 2
47                 Result (7) <= Reg1 (1);
48                 Result (6) <= Reg1 (0);
49                 Result (5) <= Reg1 (7);
50                 Result (4) <= Reg1 (6);
51                 Result (3) <= Reg1 (5);
52                 Result (2) <= Reg1 (4);
53                 Result (1) <= Reg1 (3);
54                 Result (0) <= Reg1 (2);
55                 Neg<='0';
56             When "0000000000100000"=> --6
57                 Result <= Reg4; --invert bit significance
58                 Result (7) <= Reg2 (0);
59                 Result (6) <= Reg2 (1);
60                 Result (5) <= Reg2 (2);
61                 Result (4) <= Reg2 (3);
62                 Result (3) <= Reg2 (4);
63                 Result (2) <= Reg2 (5);
64                 Result (1) <= Reg2 (6);
65                 Result (0) <= Reg2 (7);
66                 Neg<='0';
67             When "0000000001000000"=>--7

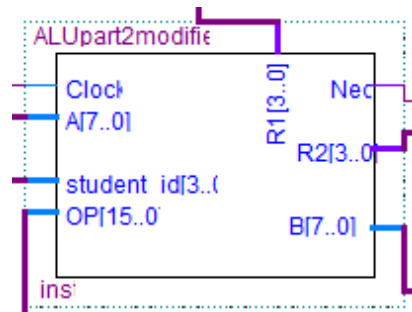
```

```

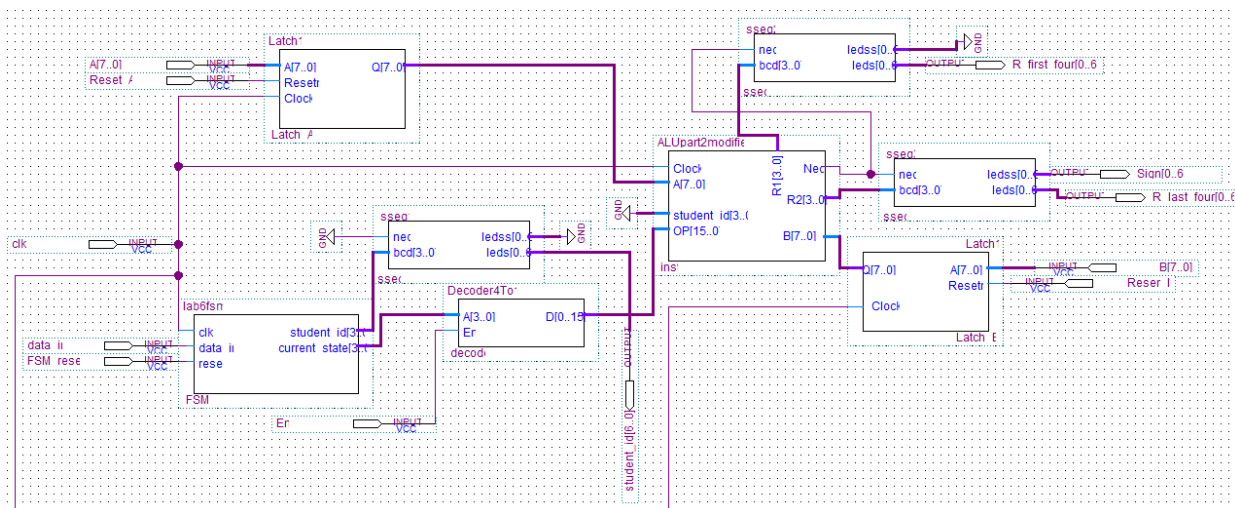
67      Result <= (Reg1 XOR Reg2); --boolean XOR
68      Neg<='0';
69      When "0000000010000000"=>--8
70      Result <= ((Reg1 + Reg2) - 4); --summation minus 4
71      Neg<='0';
72      When "0000000100000000"=>--9
73      Result (7) <= Reg1 (7); -- setting all high bits
74      Result (6) <= Reg1 (7);
75      Result (5) <= Reg1 (7);
76      Result (4) <= Reg1 (7);
77      Result (3) <= Reg1 (7);
78      Result (2) <= Reg1 (7);
79      Result (1) <= Reg1 (7);
80      Result (0) <= Reg1 (7);
81      Neg<='0';
82      When Others =>
83      -- don't care, do nothing
84      Result <="-----" ;
85  end case;
86 end if;
87 end process;
88 R1 <= Result(3 downto 0); --7-seg is only 4-bits
89 R2 <= Result(7 downto 4); --so split 8-bit
90 end calculation;

```

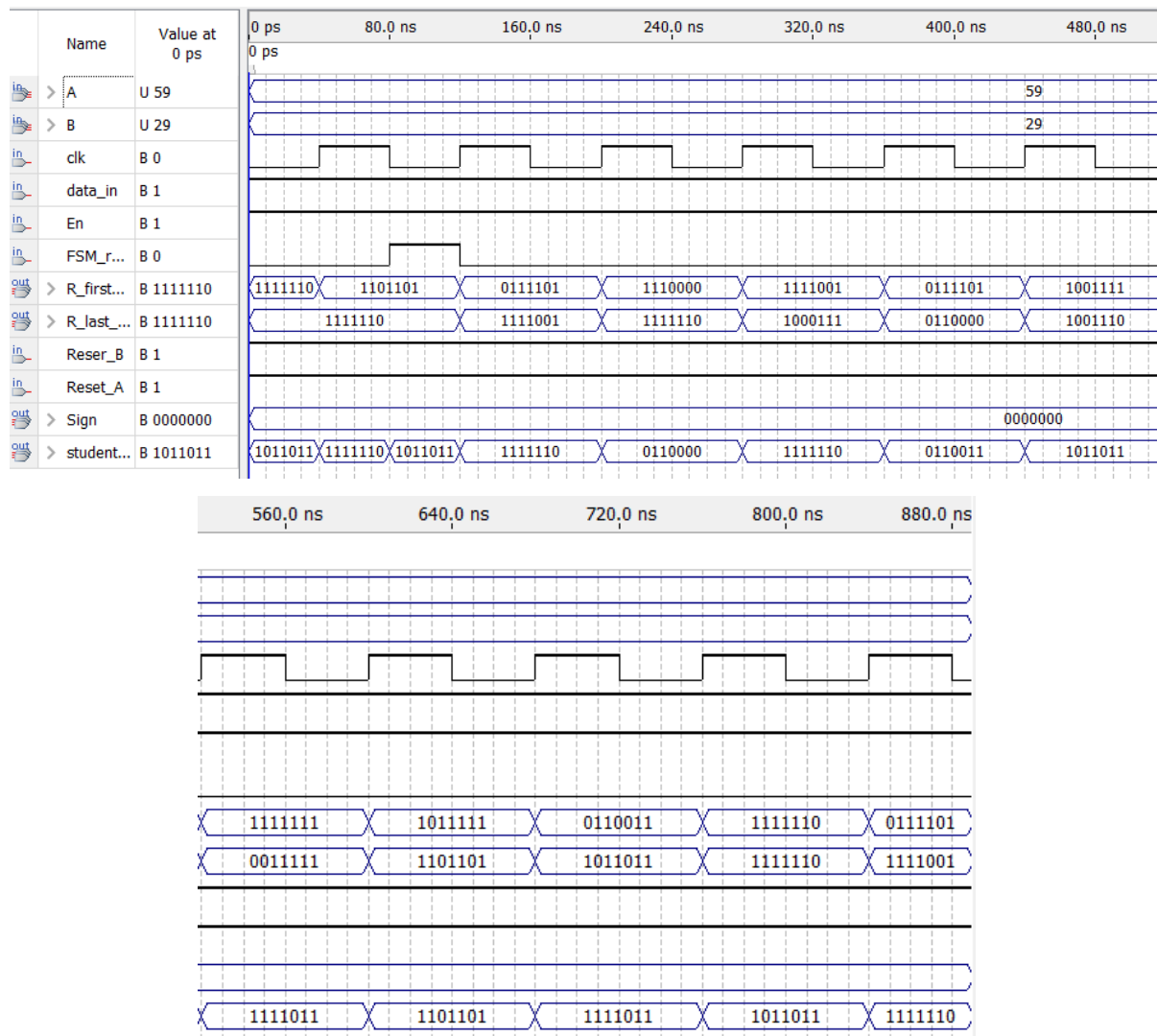
Circuit diagram of ALU component (part 2):



Circuit Diagram of ALU part 2:



Waveform:



Arithmetic Logical Unit (ALU) – Part 3

The main **purpose** of ALU part 3 required different modifications to the VHDL code and block diagram compared to the previous parts. Problem **i)** seen below was selected as the last 4 digits of my student number (5929) calculated with **mod 10** returns a **9** (problem i).

i) For each microcode instruction, 'y' if one of the 2 digits of B are less than FSM output (**student_id**) and 'n' otherwise. Use the microcode instruction from part 1 of the lab.

The final outputs will yield a 'y' or 'n' in the 7-segment display depending on the new condition. From input B (29), if one of the 2 digits are less than the corresponding digit of my student number (based on the current state of the FSM), the result will be a 'y', otherwise a 'n'. In part 3, the previous requirements of part 1 and 2 no longer serve a purpose as it would simply yield the same repeated results as last time with additional waveforms for this new condition of 'y' or 'n', so they were excluded from the new VHDL code for efficiency and simplification purposes. Furthermore, as previously showcased in the components section of this lab report, a newly modified 7-segment VHDL code was compiled for these new changes and condition for ALU part 3.

As a result, ALU part 3 has 5 inputs but only one output:

- Inputs
 - Clock
 - A
 - B
 - Student_id
 - OP
- Outputs
 - E

The clock is an input that alternates between 0 and 1 and is shared amongst all sequential circuit components including the FSM, latches, and ALU. The 8-bit inputs of A and B are stored in the latches if data_in remains 1. OP is the output of the 4:16 decoder and an input of the ALU that will determine which operation occurs based on the states of the FSM. This occurs when the clock input changes from 0 to 1 (rising edge). Unlike the previous parts, student_id finally serves a purpose as it will be compared with the digits of input B based on the condition if the digit of the student_id is greater than either digit of B. The ALU checks the values and performs the comparisons to output the calculated result. For instance, since the first digit of my student number is greater than 2 (the digit in the tens place of 29), the expected result will be a 'y' in 7-segment. Finally, E is the only output present as the ALU is no longer using both A and B to

perform various calculations and therefore will not require R1 and R2 to split the 8-bit result into 2 halves. E is the singular output of the ALU that will either be a 1 or 0, which is determined by the conditional statement implemented from problem i). Based on this value, the modified 7-segment display will use E as an input to display a 'y' or 'n' when E is 1 or 0 respectively.

ALU core operations (microcode's generated by decoder) for problem 3:

Function	Operation	Student #	Expected Results
1	If student # is > 2 or 9, 'Y'; else 'N'	5	Y
2	If student # is > 2 or 9, 'Y'; else 'N'	0	N
3	If student # is > 2 or 9, 'Y'; else 'N'	1	N
4	If student # is > 2 or 9, 'Y'; else 'N'	0	N
5	If student # is > 2 or 9, 'Y'; else 'N'	4	Y
6	If student # is > 2 or 9, 'Y'; else 'N'	5	Y
7	If student # is > 2 or 9, 'Y'; else 'N'	9	Y
8	If student # is > 2 or 9, 'Y'; else 'N'	2	N
9	If student # is > 2 or 9, 'Y'; else 'N'	9	Y

In the VHDL code, the operation rem was used to separate input B into 2 separate digits of 2 and 9 respectively. This is because if 29 in binary was imply split into 2 halves like using R1 and R2 to split the result into 2 halves, it would equal different numbers than 2 and 9. The operation rem and its respective calculations ensure to separate 29 accurately before performing the conditional statements. Additionally, since 2 is always less than 9, and the condition requires that one of digits are less than the student number, only using 2 in the if statement would also yield the same results. However, both digits are considered in VHDL code to ensure the instructed condition was indeed achieved.

VHDL code:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5  ENTITY ALUpart3modified IS
6  port (Clock: in std_logic; --input clock signal
7  A,B : in unsigned(7 downto 0); --8-bit inputs from latches A and B
8  student_id : in unsigned(3 downto 0); --4-bit student id from FSM
9  OP: in unsigned(15 downto 0); --16-bit selector for operation from decoder
10 --Neg: out std_logic;--is the result negative? set -ve bit output
11 E: out std_logic); -- yes or no input for modified ALU
12 --R1: out unsigned (3 downto 0);--lower 4-bits of 8-bit result output
13 --R2: out unsigned (3 downto 0);--higher 4-bits of 8-bit result output
14 end ALUpart3modified;
15 ARCHITECTURE calculation of ALUpart3modified IS --temporary signal declaration
16 |   signal Reg1, Reg2: unsigned (7 downto 0):= (others=>'0');
17 |   signal Reg4, Reg5 : unsigned (0 to 7);
18 |   begin
19 |   --Reg1 <= A; --temp storing A in Reg1 local variable
20 |   -- Part 3 only requires input B, so Reg 2 is used
21 |   Reg2 <= B; --temp storing B in Reg2 local variable
22 |   Reg4 <= ((Reg2 rem 100)/10); -- tens place digit (2)
23 |   Reg5 <= (Reg2 rem 100); -- ones place digit (9)
24 |   process(Clock,OP)
25 |   begin
26 |       if(rising_edge (Clock)) Then -- calculation @ positive edge of clock
27 |       case OP is
28 |       When "0000000000000001"=>--1
29 |           if (student_id > (Reg4) or (student_id > Reg5)) Then
30 |               E <= '1';
31 |           else
32 |               E <= '0';
33 |           end if;
34 |       When "0000000000000010"=>--2
35 |           if (student_id > (Reg4) or (student_id > Reg5)) Then
36 |               E <= '1';
37 |           else
38 |               E <= '0';
39 |           end if;
40 |       When "0000000000000100"=>--3
41 |           if (student_id > (Reg4) or (student_id > Reg5)) Then
42 |               E <= '1';
43 |           else
44 |               E <= '0';
45 |           end if;
46 |       When "0000000000001000"=>--4
47 |           if (student_id > (Reg4) or (student_id > Reg5)) Then
48 |               E <= '1';
49 |           else
50 |               E <= '0';
51 |           end if;
52 |       When "0000000000010000"=>--5
53 |           if (student_id > (Reg4) or (student_id > Reg5)) Then
54 |               E <= '1';
55 |           else
56 |               E <= '0';
57 |           end if;
58 |       When "0000000000100000"=> --6
59 |           if (student_id > (Reg4) or (student_id > Reg5)) Then
60 |               E <= '1';

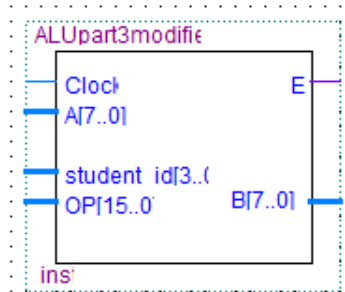
```

```

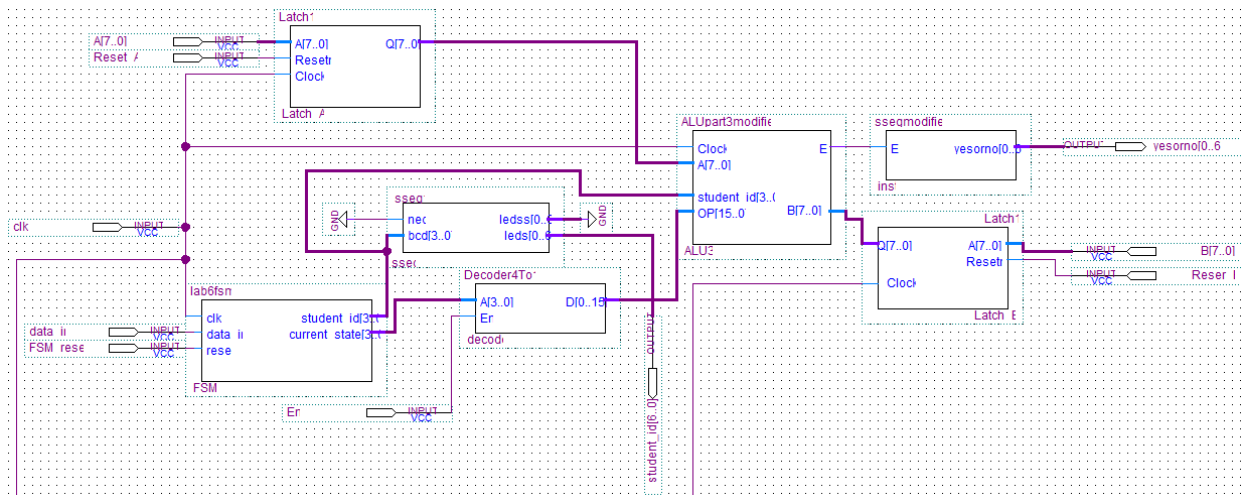
61         else
62             E <= '0';
63         end if;
64         When "0000000001000000"=>--7
65             if (student_id > (Reg4) or (student_id > Reg5)) Then
66                 E <= '1';
67             else
68                 E <= '0';
69             end if;
70         When "0000000010000000"=>--8
71             if (student_id > (Reg4) or (student_id > Reg5)) Then
72                 E <= '1';
73             else
74                 E <= '0';
75             end if;
76         When "0000000100000000"=>--9
77             if (student_id > (Reg4) or (student_id > Reg5)) Then
78                 E <= '1';
79             else
80                 E <= '0';
81             end if;
82         When Others =>
83             -- don't care, do nothing
84         end case;
85     end if;
86 end process;
87 --R1 <= Result(3 downto 0); --7-seg is only 4-bits
88 --R2 <= Result(7 downto 4); --so split 8-bit
89 end calculation;

```

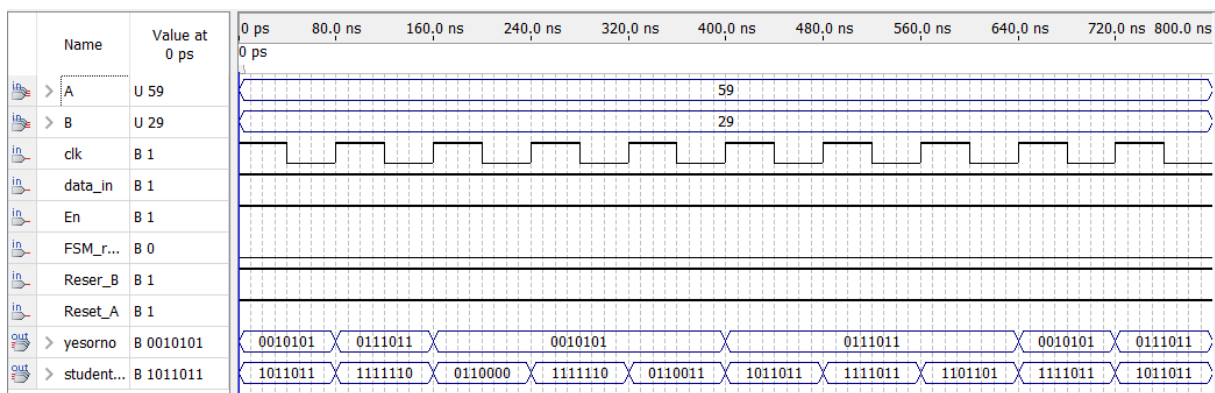
Circuit diagram of ALU component (part 3):



Circuit diagram of ALU part 3:



Waveform:



[This part of page is left blank intentionally]

Conclusion

Overall, the main objective of lab 6 was to use our old lab experiments and new knowledge of sequential circuits to create 3 variations of an arithmetic logical unit (ALU). All the work was conducted on a FPGA board using VHDL code. In summary, each individual component work together to build a processor. Latches 1 and 2 are D flip-flops that act as storage units for the 8-bit inputs of A and B. The FSM was created via Mealy logic implementation. So, the output is dependent on the current state and the current inputs. The 4:16 decoder uses those states from the FSM to essentially convert it into the microcode's of the ALU. The FSM and 4:16 decoder are sub-components that comprise of the overall control unit that fetch the instructions and signals for the ALU. The ALU core then performs the required operations based on all these inputs. The FSM, ALU and latches are sequential circuits, so they all have a clock input that alternates between 1 and 0, whereas the decoder and 7-segment display are combinational circuits so they do not require this input. In all 3 parts of the lab, every individual component worked together cohesively to produce results. In conclusion, the entire lab 6 experiment was successful and yielded all the correct and expected results. All the VHDL codes and block diagrams for every component compiled successfully, and every waveform diagram displayed accurate values. This indicates that each individual component works correctly, and when combined a fully functioning general-purpose processor was achieved.