TABLE OF CONTENTS:

Introduction

In this lab, we built a General Processing Unit (GPU), a key component in digital systems. The GPU was assembled using several parts: two latches, a Finite State Machine (FSM), a 4:16 decoder, and an Arithmetic Logic Unit (ALU). The latches stored two 8-bit numbers that were later processed by the ALU. The FSM, which has nine states represented by a four-bit number, worked with the decoder to form the control unit of the GPU. The decoder converted the FSM's four-bit number into a 16-bit output, which determined the operation for the ALU.

In addition to constructing the GPU, we also did Problems 2 and 3. These problems involved modifying the ALU core and the Control Unit (FSM) and sseg. These allowed us to understand how changes in its components can affect its overall functionality. This lab allowed us to apply our knowledge from previous experiments and gain a deeper understanding of how these components work together in a digital system.

Components

Part 1:

The student number used for this lab is 501175819. Therefore, A = $(58)_{16}$ and B = $(19)_{16}$. In binary, A and B are translated to A = $(0101\ 1000)_2$ and B = $(0001\ 1001)_2$

Latch1:

In this lab, the latch, a key component of the storage unit (Register), plays a crucial role. Specifically, latch 1 is designed to temporarily store an 8-bit binary number, which in this case is input A. The value for input A is derived from the third and fourth last digits of the student number, represented as $(58)_{16}$ or equivalently, A = $(0101\ 1000)_2$ in binary.

Latch are inputs for the Arithmetic Logic Unit (ALU), where further processing takes place.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity latch1 is
5     port (A: in std_logic_vector(7 downto 0);
6           Resetn, Clock: in std_logic;
7           q: out std_logic_vector (7 downto 0));
8  end latch1;
9
10 architecture behavior of latch1 is
11    begin
12       process (Resetn, Clock)
13          begin
14             if Resetn = '0' then
15                Q <= "00000000";
16             elsif Clock'event and Clock='1' then
17                Q <= A;
18             end if;
19       end process;
20 end behavior;
```

Figure 02: Latch1 VHD

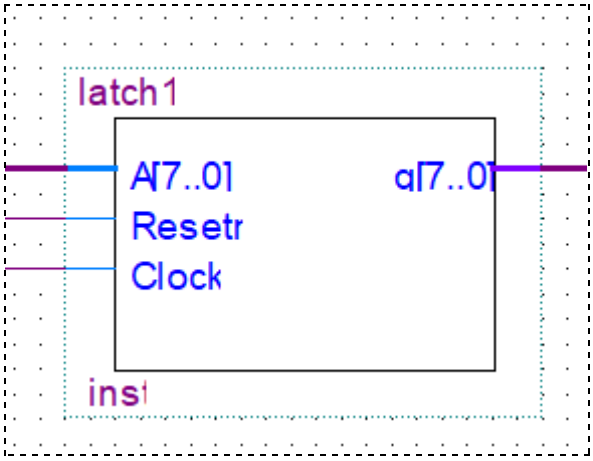


Figure 03: Latch1 block

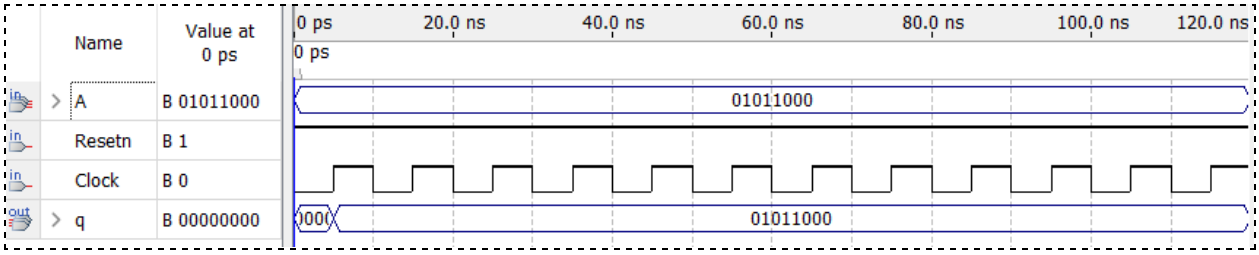


Figure 04: Latch1 VWF

The given waveform represents an 8-bit binary input set to 58. When the clock signal is at a low state (0), the output remains at 0. Conversely, when the clock signal is high (1), the output aligns with and reflects the input value. This waveform accurately corresponds to the truth table displayed below.

Latch2:

Similar to Latch 1, latch 2 has the same design and stores the number $(19)_{16}$ or equivalently, A = $(0001\ 1001)_2$ in binary which is 8 bits. This is the last 2 digits of the student number.

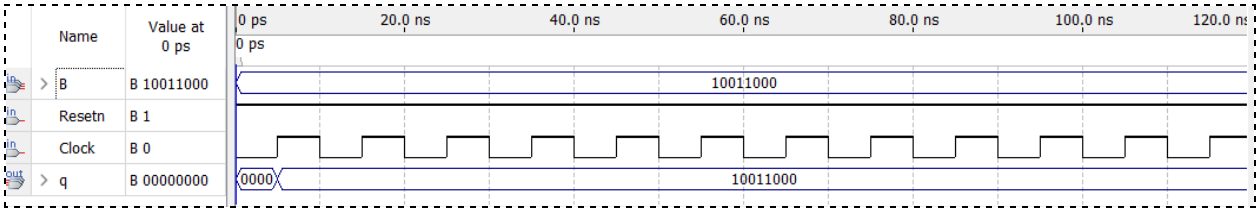


Figure 05: Latch2 VWF

The displayed waveform represents an 8-bit binary input set to 19. During the clock's low state (0), the output remains at 0. Conversely, when the clock transitions to the high state (1), the output aligns with and mirrors the input value. This waveform faithfully represents the truth table displayed below.

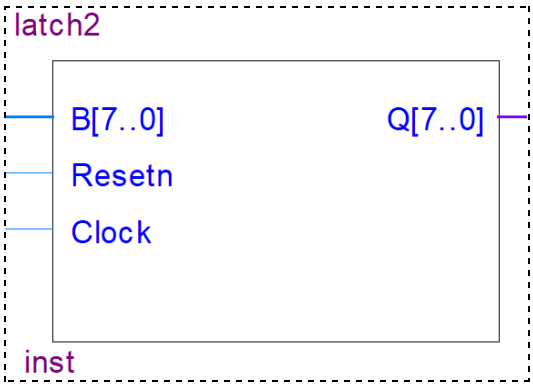
Clk	D	Q(t+1)
0	x	Q(t)
1	0	0
1	1	1

Figure 06: Latch Truth Table

Figure 07: Latch2 VHD

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity latch2 is
5      port (B: in std_logic_vector(7 downto 0);
6            Resetn, Clock: in std_logic;
7            q: out std_logic_vector (7 downto 0));
8   end latch2;
9
10  architecture behavior of latch2 is
11     begin
12        process (Resetn, Clock)
13           begin
14              if Resetn = '0' then
15                 Q <= "00000000";
16              elsif Clock'event and Clock='1' then
17                 Q <= B;
18              end if;
19        end process;
20  end behavior;
```

Figure 08: Latch2 block



FSM

The Finite State Machine (FSM) is a key part of the control unit in this processor. It's a Moore machine that cycles through nine states, from S0 to S8. The FSM transitions from one state to the next in a consecutive order when the data input is 1 and once it reaches S8, it loops back to S0, creating a cycle through these nine states.

Each state corresponds to a digit of the student ID. As the FSM transitions through each state, it outputs the respective digit. The output values depend only on the current state for the decoder. The main role of the FSM is to determine the pattern of the controller sequence. It functions like an up counter while feeding data to the decoder. This ensures the correct operation is selected in the ALU based on the current state of the FSM.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity machine is
4     port( clk            : in std_logic;
5           data_in        : in std_logic;
6           reset          : in std_logic;
7           student_id     : out std_logic_vector(3 downto 0);
8           current_state  : out std_logic_vector(3 downto 0));
9  end machine;
10 architecture fsm of machine is
11 -- build an enumerated type with 9 states for the state machine
12 --(9 states for parsing 9 digits of student id)
13 type state_type is (s0, s1, s2, s3, s4, s5, s6, s7, s8);
14 -- register to hold the current state
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 = '0') then
25                yfsm <= s0;
26             ELSE
27                yfsm <= s1;
28             END IF;
29          when s1=>
30             IF (data_in = '0') THEN
31                yfsm <= s1;
32             ELSE
33                yfsm <= s2;
34             END IF;
35          when s2=>
36             IF (data_in ='0') THEN
37                yfsm <= s2;
38             ELSE
39                yfsm <= s3;
40             END IF;
```

Figure 09: FSM VHD 1

```
40             END IF;
41          when s3=>
42             IF (data_in = '0') THEN
43                yfsm <= s3;
44             ELSE
45                yfsm <= s4;
46             END IF;
47          when s4=>
48             IF (data_in = '0') THEN
49                yfsm <= s4;
50             ELSE
51                yfsm <= s5;
52             END IF;
53          when s5=>
54             IF (data_in ='0') THEN
55                yfsm <= s5;
56             ELSE
57                yfsm <= s6;
58             END IF;
59          when s6=>
60             IF (data_in ='0') THEN
61                yfsm <= s6;
62             ELSE
63                yfsm <= s7;
64             END IF;
65          when s7=>
66             IF (data_in ='0') THEN
67                yfsm <= s7;
68             ELSE
69                yfsm <= s8;
70             END IF;
71          when s8=>
72             IF (data_in ='0') THEN
73                yfsm <= s8;
74             ELSE
75                yfsm <= s0;
76             END IF;
77          END CASE;
78       END IF;
79    END process;
```

Figure 10: FSM VHD 2

```
80    -- implement the moore or mealy logic here (MOORE)
81    process (yfsm, data_in) -- data_in if read only
82    begin
83       case yfsm is
84          when s0=>
85             student_id <= "0101"; --5 (d1)
86             current_state <= "0000";
87          when s1=>
88             student_id <= "0000"; --0 (d2)
89             current_state <= "0001";
90          when s2=>
91             student_id <= "0001"; --1 (d3)
92             current_state <= "0010";
93          when s3=>
94             student_id <= "0001"; --1 (d4)
95             current_state <= "0011";
96          when s4=>
97             student_id <= "0111"; --7 (d5)
98             current_state <= "0100";
99          when s5=>
100            student_id <= "0101"; --5 (d6)
101            current_state <= "0101";
102         when s6=>
103            student_id <= "1000"; --8 (d7)
104            current_state <= "0110";
105         when s7=>
106            student_id <= "0001"; --1 (d8)
107            current_state <= "0111";
108         when s8=>
109            student_id <= "1001"; --9 (d9)
110            current_state <= "0100";
111      END CASE;
112   END process;
113   END fsm;
```

Figure 11: FSM VHD 3

The provided VHDL code outlines the implementation of the FSM. It includes the process of transitioning between states based on the data input and the clock signal. The FSM takes the clock, reset, and data in as inputs. Its outputs include the student number and current state

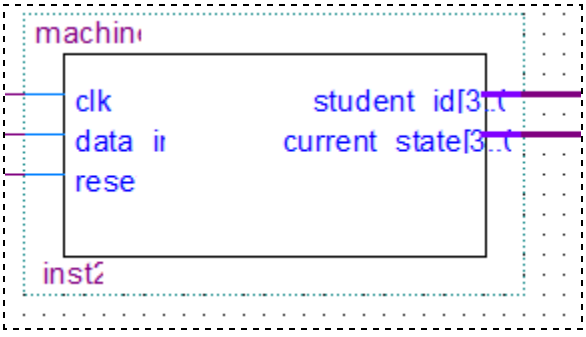


Figure 12: FSM block

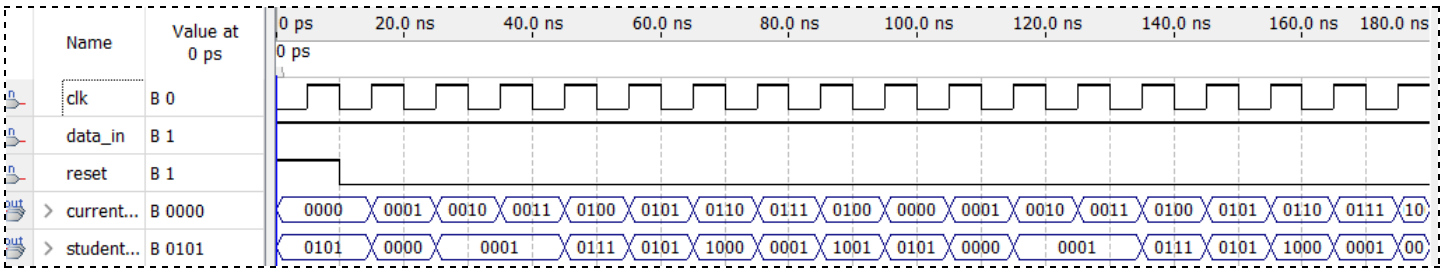


Figure 13: FSM VWF

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	0001
0100	0100	0101	0111
0101	0101	0110	0101
0110	0110	0111	1000
0111	0111	1000	0001
1000	1000	0000	1001

Figure 14: Truth Table: Moore Machine

The truth table provided demonstrates that whenever the data input is 1, the current state consistently progresses to the subsequent state, cycling back to state S0 once the eighth state reaches it. The column labeled "Student ID" in the truth table showcases the student number (501175819) represented in binary form. It's evident that the outputs generated from the truth table align perfectly with those observed in the waveform. Hence, it's valid to infer that the waveform is accurate and in line with the expected outputs based on the truth table.

Decoder:

The 4:16 decoder takes 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. The 4:16 decoder is made up of two 3 to 8 decoders as specified in the lab guide.

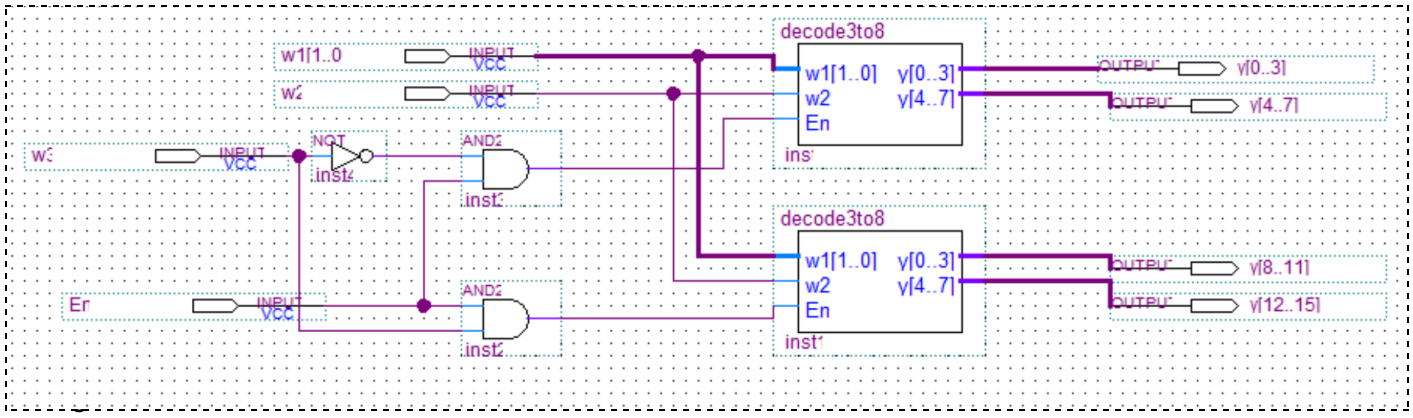


Figure 15: Decoder Block

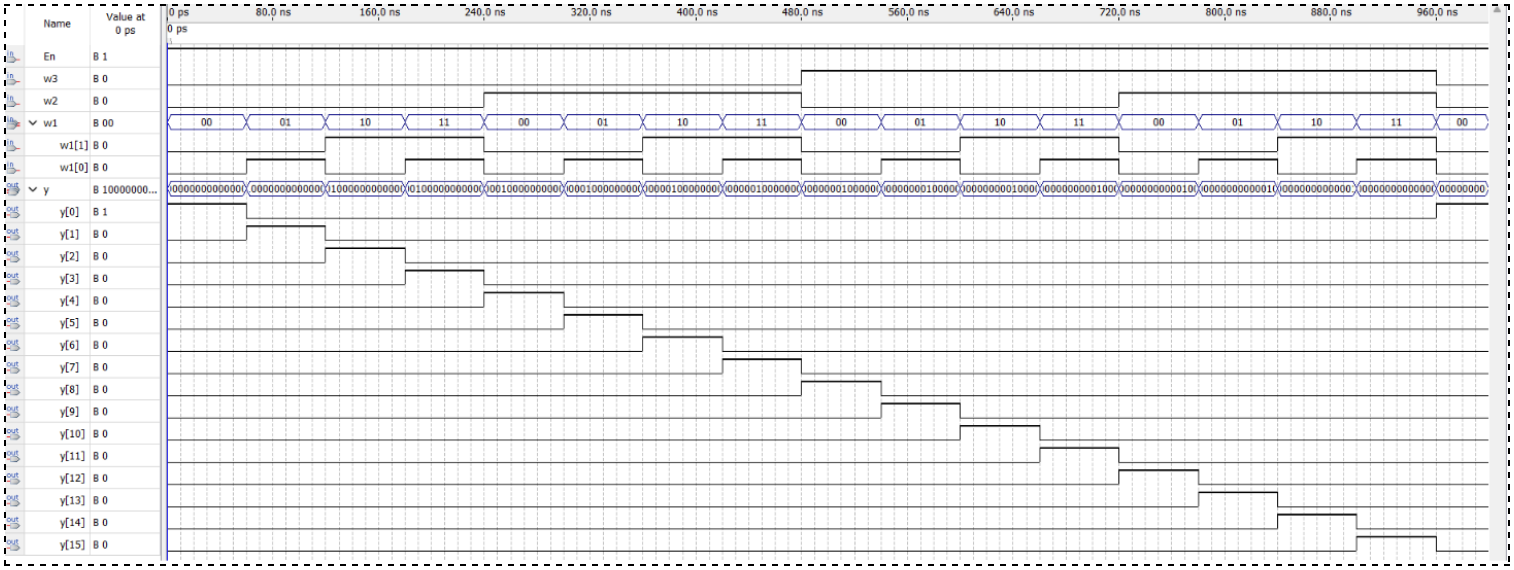


Figure 16: Decoder VWF

As can be seen from the waveform, the 4-bit input has been extended to an 8-bit input for each case. Therefore, the waveform can be said to be correct as it provides the correct outputs.

Figure 17: Decoder Truth Table

En	w3	w2	w1	w0	Output															
1	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**

7-Segment Display:

The 7-segment display code used in this lab is similar to the one used in Lab 3. It's designed to display values from 0 to 15 in hexadecimal format and can handle negative outputs. We used several 7-segment displays: one to display the student ID from the FSM and two others to display the output from the ALU. The ALU outputs an 8-bit value, so two 7-segment displays are needed because each one can only interpret a 4-bit input.

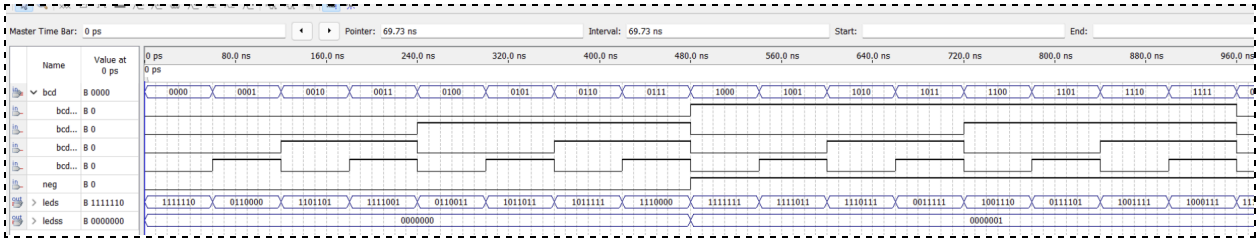


Figure 18: SSEG VWF

```
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           sign : IN STD_LOGIC;
6           leds, ledss : OUT STD_LOGIC_VECTOR (0 TO 6));
7      END sseg;
8
9   ARCHITECTURE Behavior OF sseg IS
10
11  BEGIN
12     PROCESS(bcd,sign)
13        BEGIN
14        IF sign = '0'
15        THEN
16           ledss <= "1111111";
17        ELSE
18           ledss <= "1111110";
19        END IF;
20
21        CASE bcd IS
22              WHEN "0000" =>leds<="0000001";  --0
23              WHEN "0001" =>leds<="1001111";  --1
24              WHEN "0010" =>leds<="0010010";  --2
25              WHEN "0011" =>leds<="0000110";  --3
26              WHEN "0100" =>leds<="1001100";  --4
27              WHEN "0101" =>leds<="0100100";  --5
28              WHEN "0110" =>leds<="0100000";  --6
29              WHEN "0111" =>leds<="0001111";  --7
30              WHEN "1000" =>leds<="0000000";  --8
31              WHEN "1001" =>leds<="0001100";  --9
32
33              WHEN "1010" =>leds<="0001000";  --A
34              WHEN "1011" =>leds<="1100000";  --b
35              WHEN "1100" =>leds<="0110001";  --C
36              WHEN "1101" =>leds<="1000010";  --d
37              WHEN "1110" =>leds<="0110000";  --E
38              WHEN "1111" =>leds<="0111000";  --F
39
40                 WHEN OTHERS =>leds<="1111111";
41           END CASE;
42     END PROCESS;
43  END Behavior;
```

Figure 19: SSEG VHD

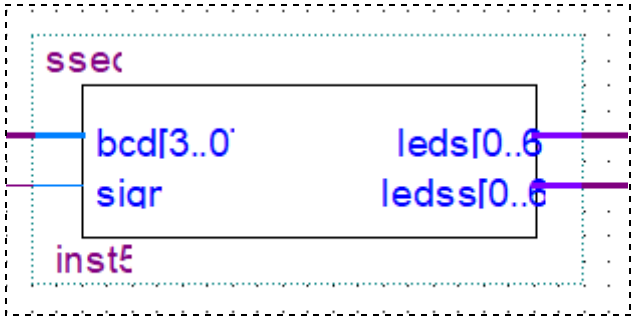


Figure 20: SSEG block

The provided VHDL code outlines the implementation of the 7-segment display. It includes a process that takes in a binary-coded decimal (bcd) and a sign as inputs. If the sign is '0', all segments of the display are turned on. Otherwise, all segments except one are turned on. The code then uses a case statement to determine which segments to turn on based on the bcd input. This ensures that the correct hexadecimal value is displayed on the 7-segment display.

Figure 21: 7 - Segment Truth Table: bcd input

bcd (input)	leds (abcdefg)	Letter/Number Input
0000	0000001	0
0001	1001111	1
0010	0010010	2
0011	0000110	3
0100	1001100	4
0101	0100100	5
0110	0100000	6
0111	0001111	7
1000	0000000	8
1001	0001100	9
1010	0001000	A
1011	1100000	B
1100	0110001	C
1101	1000010	D
1110	0110000	E
1111	0111000	F

Figure 22: 7 - Segment Truth Table: neg input

neg	leds (abcdefg)
0	0000000
1	0000001

ALU

The Arithmetic Logic Unit (ALU) is the central component of the processor, performing operations based on the states of the Finite State Machine (FSM). It handles nine operations, including addition, subtraction, and various Boolean functions. It uses five inputs: Clock, A, B, student_id, and OP. On each clock's rising edge, the ALU checks OP, output of the decoder, and performs the corresponding operation on A and B.

The ALU has three outputs: Neg, R1, and R2. Neg indicates a negative output, allowing the 7-segment display to show a negative number by lighting up segment g if Neg is equal to 1. Since the 7-segment display takes a 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. Each one goes to a separate 7-segment display to display the binary output of each half in hexadecimal. Combined, they display the expected 8-bit output.

Figure 23: ALU VWF

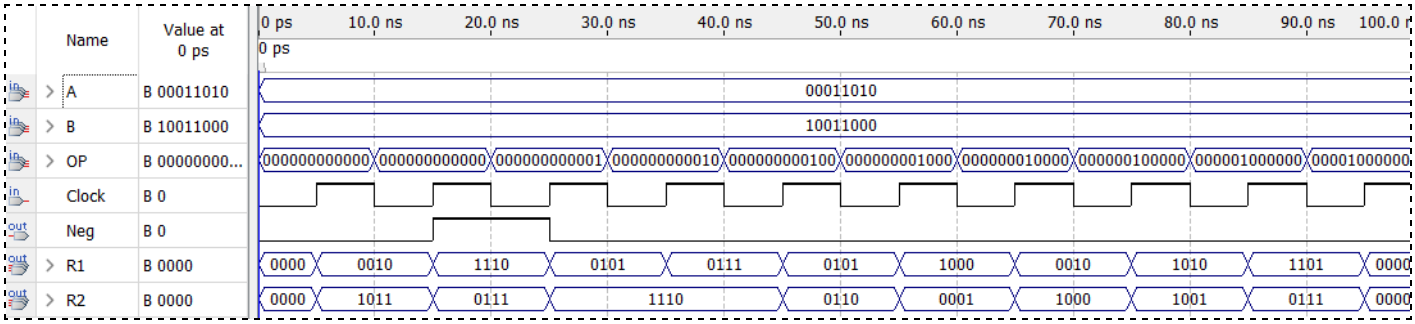
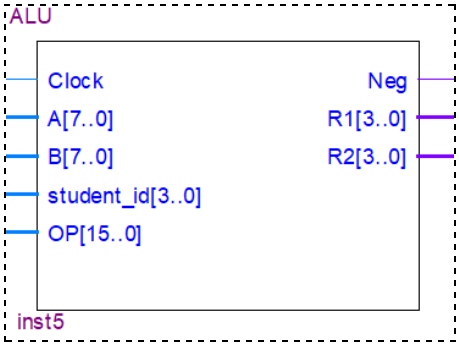


Figure 24: ALU VHD 1

```
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 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      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 fourbits 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;--temporarily store a in reg1 local variable
19 Reg2 <= B;--temporarily store b in reg2 local variable
20 process(Clock,OP)
21     begin
22     if(rising_edge(Clock)) THEN
23         case OP is
24
25             WHEN "0000000000000001" => -- 1 sum of a and b
26                   Result <= Reg1 + Reg2;
27                   Neg <= '0';
28             WHEN "0000000000000010" => --2 difference of a and b
29                   IF Reg1 > Reg2 THEN
30                      Result <= (Reg1 - Reg2);
31                      Neg <= '0';
32                   ELSE
33                      Result <= (Reg2 - Reg1);
34                      Neg <= '1';
35                   END IF;
36             WHEN "0000000000000100" => --3 not a
37                   Result <= NOT(Reg1);
38                   Neg <= '0';
39             WHEN "0000000000001000" => --4 a nand b
40                   Result <= (Reg1 NAND Reg2);
```

Figure 25: ALU Block



```
41                Neg <= '0';
42            WHEN "0000000000010000" => --5 a nor b
43                Result <= (Reg1 NOR Reg2);
44                Neg <= '0';
45            WHEN "0000000000100000" => --6 a and b
46                Result <= (Reg1 AND Reg2);
47                Neg <= '0';
48            WHEN "0000000001000000" => --7 a xor b
49                Result <= (Reg1 XOR Reg2);
50                Neg <= '0';
51            WHEN "0000000010000000" => --8  a or b
52                Result <= (Reg1 OR Reg2);
53                Neg <= '0';
54            WHEN "0000000100000000" => --9 a xnor b
55                Result<= (Reg1 XNOR Reg2);
56                Neg <= '0';
57            WHEN OTHERS =>
58                Result<= "00000000";
59                Neg <= '0';
60          end case;
61       end if;
62    end process;
63
64 R1 <= Result(3 downto 0);
65 R2 <= Result(7 downto 4);
66 end calculation;
```

Figure 26: ALU VHD 2

The provided VHDL code details the ALU's implementation. It includes a process that takes in the clock and OP as inputs. If a rising edge of the clock is detected, it checks the value of OP and performs the corresponding operation on inputs A and B. The result is then split into two 4-bit outputs, R1 and R2, which are displayed on the 7-segment display.

PROBLEM 1: Combined ALU -

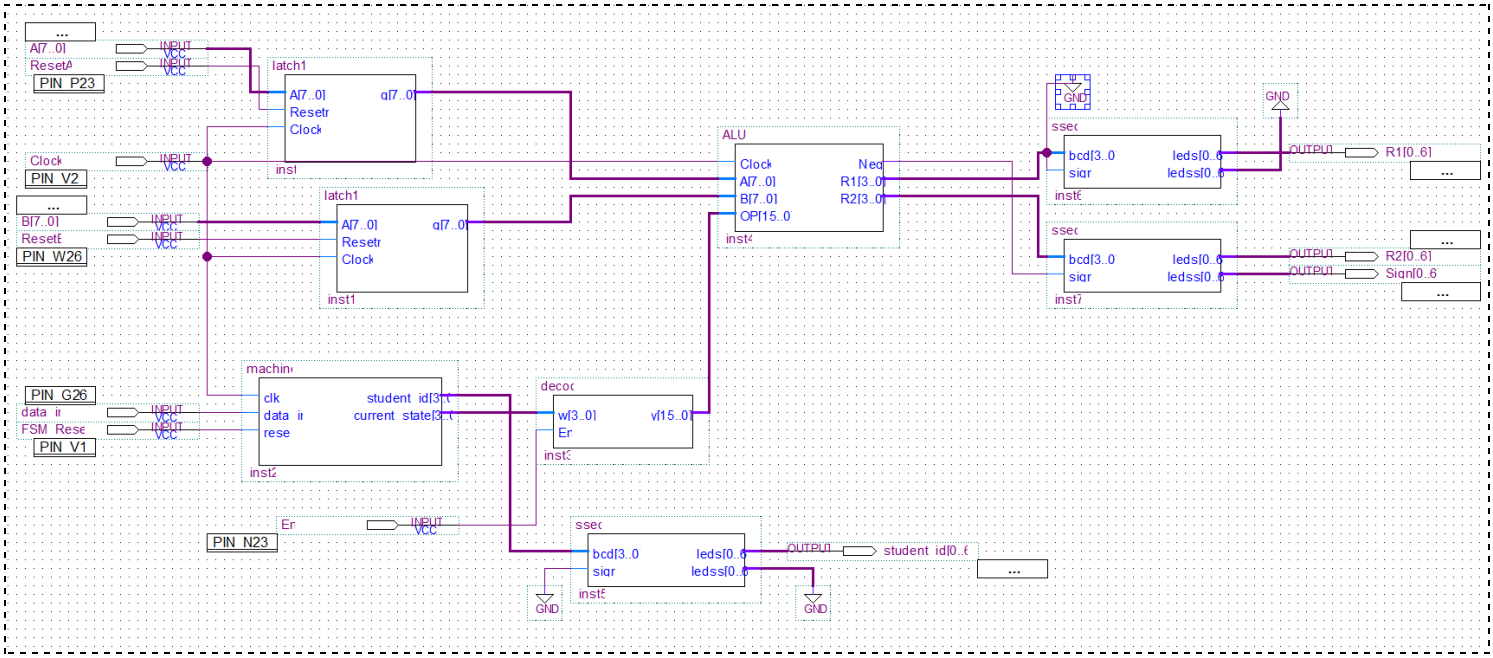


Figure 27: Combined ALU Block

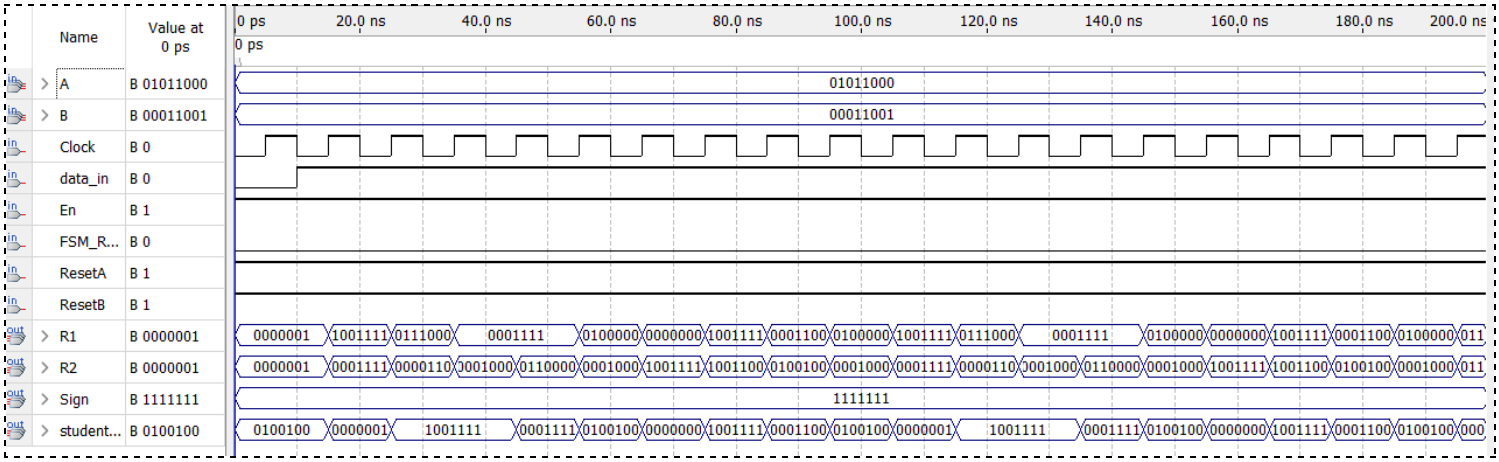


Figure 28: Combined ALU VWF

In the analysis of Figure 28, it's observed that the initial output is "0000", which doesn't correspond to the expected microcode operations. This discrepancy arises due to a delay between the decoder and the ALU. The ALU performs operations at the rising edge of the clock, while the decoder doesn't follow this timing. The ALU operates based on its preceding state from the FSM. Hence, the initial "0000" output occurs because the FSM doesn't have a state prior to 0, leading to this output. This delay is also noticeable in ALU 2 and ALU 3.

Figure 29: Table of the inputs and outputs for ALU 1:

Input	Purpose
Clock	Input signal to carry out operations.
A[7..0]	Input value received from Latch 1.
B[7..0]	Input value received from Latch 2.
OP	16 bit microcode based on the FSM state.
Output	**Purpose**
R1	First four bits of the result calculated from the ALU.
R2	Last four bits of the result calculated from the ALU.
Neg	Shows if the result is positive or negative.

Figure 30: Calculations

Function #	Microcode	Operation / Function	Binary Result	Hexadecimal Result
1	0000000000000001	Sum(**A**, **B**)	0111 0001	71
2	0000000000000010	Diff(**A**, **B**)	0011 1111	3F
3	0000000000000100	$\overline{A}$	1010 0111	A7
4	0000000000001000	$\overline{A \cdot B}$	1110 0111	E7
5	0000000000010000	$\overline{A + B}$	1000 1110	8E
6	0000000000100000	$A \cdot B$	0001 1000	18
7	0000000001000000	$A \oplus B$	0100 0001	41
8	0000000010000000	$A + B$	0111 0001	71
9	0000000100000000	$\overline{A \oplus B}$	0100 0001	BE

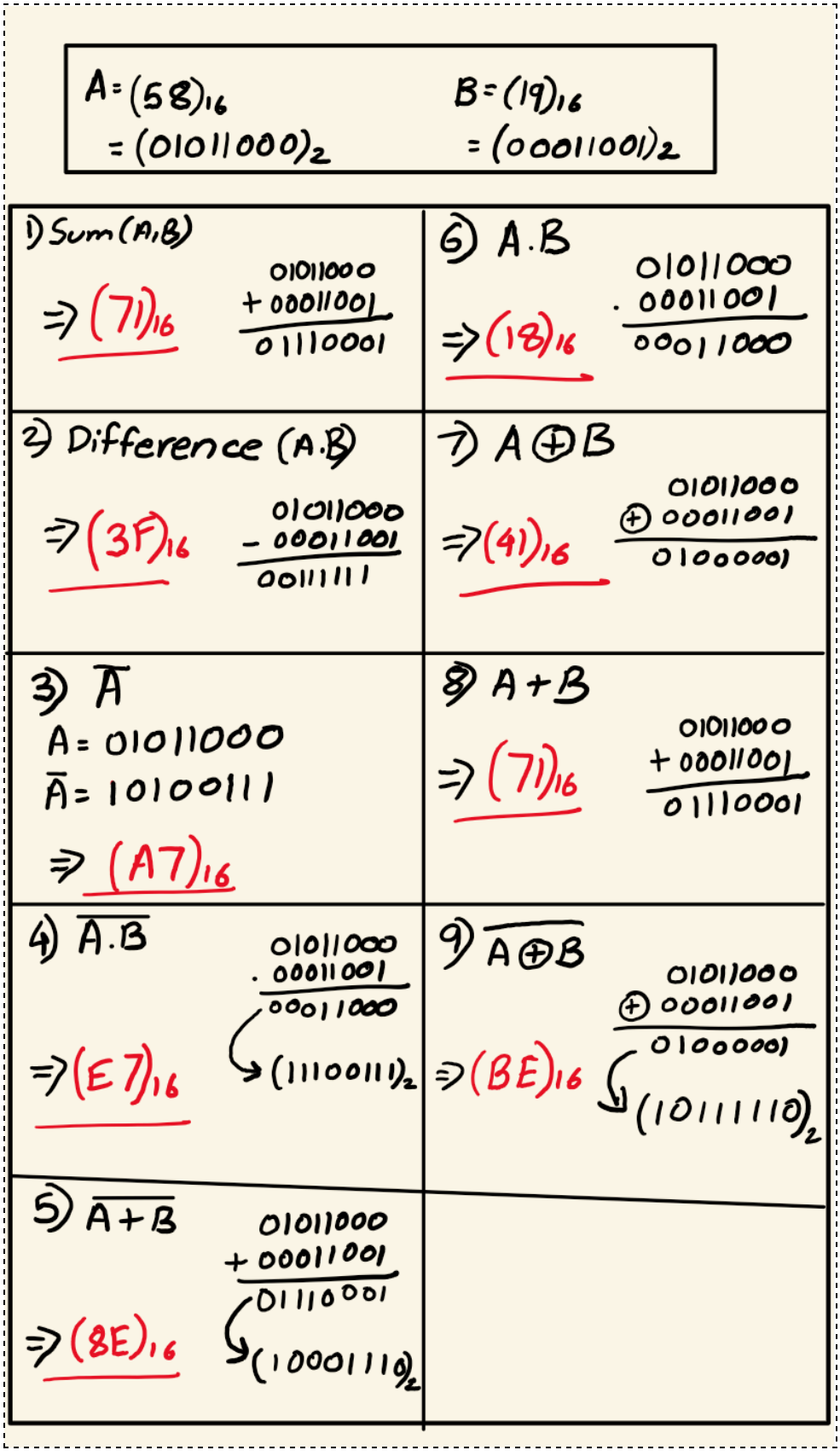


Figure 31: Calculations

Problem 2 (a): ALU 2

In the second part, the Arithmetic Logic Unit (ALU) has identical inputs and outputs, as explained in the ALU section. Inputs A and B, both 8-bit values, are stored in the latches when data_in is set to 1. When a rising edge occurs (transition from 0 to 1), the OP signal determines the specific operation to perform based on the microcode, as OP is derived from the output of the 4-to-16 decoder. R1 and R2 denote the two halves of the resulting outputs from the ALU, representing bits 7-4 and 3-0, respectively. When the neg signal is set to 1, it signifies a negative value and ensures that segment g will be illuminated to indicate a negative sign.

For problem 2, function set a) was selected. The functions for a) can be seen below in figure 33.

Figure 32: Input and Output Table

Input	Purpose
Clock	Input signal to carry out operations.
A[7..0]	Input value received from Latch 1.
B[7..0]	Input value received from Latch 2.
OP	16 bit microcode based on the FSM state.
Output	**Purpose**
R1	First four bits of the result calculated from the ALU.
R2	Last four bits of the result calculated from the ALU.
Neg	Shows if the result is positive or negative.

Figure 33: (a) Functions

a)

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

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

entity ALU_Problem2 is
port(Clk:in std_logic;
A,B : in unsigned(7 downto 0);
OP : in unsigned(15 downto 0);
Neg : out std_logic;
R1: out unsigned(3 downto 0);
R2: out unsigned(3 downto 0));
end entity;
architecture calculation of ALU_Problem2 is
signal Reg1,Reg2,Result : unsigned(7 downto 0) :=(others=> '0');
begin
Reg1 <= A;
Reg2 <= B;
process(Clk,OP)
   begin
   if(rising_edge(Clk)) THEN
      case OP is -- Problem 2 (a)

         WHEN "0000000000000001" => -- 1 Increment A by 2
         Result <= Reg1 + "00000010";

         WHEN "0000000000000010" => -- 2 Shift B to right by two bits, input bit = 0 (SHR)
         Result <= ("00" & Reg2(7 downto 2) );


         WHEN "0000000000000100" => -- 3 Shift A to right by four bits, input bit = 1 (SHR)
         Result <= ("1111" & Reg1(7 downto 4));


         WHEN "0000000000001000" => -- 4 Find the smaller value of A and B and produce the results (Min(A,B))
               if(Reg1 < Reg2) then
                  Result <= Reg1;
               else
                  Result <= Reg2;
               end if;


         WHEN "0000000000010000" => -- 5 Rotate A to right by two bits (ROR)
         Result <= (Reg1(1 downto 0) & Reg1(7 downto 2));


         WHEN "0000000000100000" => -- 6 Invert the bit-significance order of B
               Result(0)<= Reg2(7);
               Result(1)<= Reg2(6);
               Result(2)<= Reg2(5);
               Result(3)<= Reg2(4);
               Result(4)<= Reg2(3);
               Result(5)<= Reg2(2);
               Result(6)<= Reg2(1);
               Result(7)<= Reg2(0);
               Neg <='0';


         WHEN "0000000001000000" => -- 7 Produce the result of XORing A and B
         Result <= Reg1 xor Reg2;

         WHEN "0000000010000000" => -- 8 Produce the summation of A and B, then decrease it by 4
         Result <= (Reg1 + Reg2) - "00000100";

         WHEN "0000000100000000" => -- 9 Produce all high bits on the output
         Result <= "11111111";

         WHEN OTHERS =>
         Result<= "--------";


         end case;
      end if;
   end process;

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

Figure 34: ASU Problem 2(a) VHD

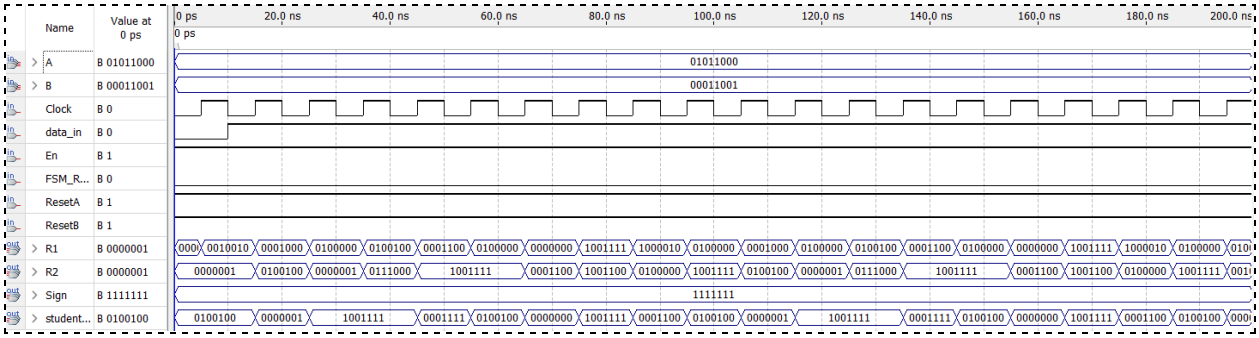


Figure 35: Problem 2 (a) VWF

As seen in the waveform the outputs align with the calculated values shown above, for example when student number is 5, the output on the FPGA board should be ‘5A’. R1 should show 5 (01000100) and R2 should show A (00010000), when Student Number is 5 (01000100) ignoring the delay caused by the decoder and ALU.

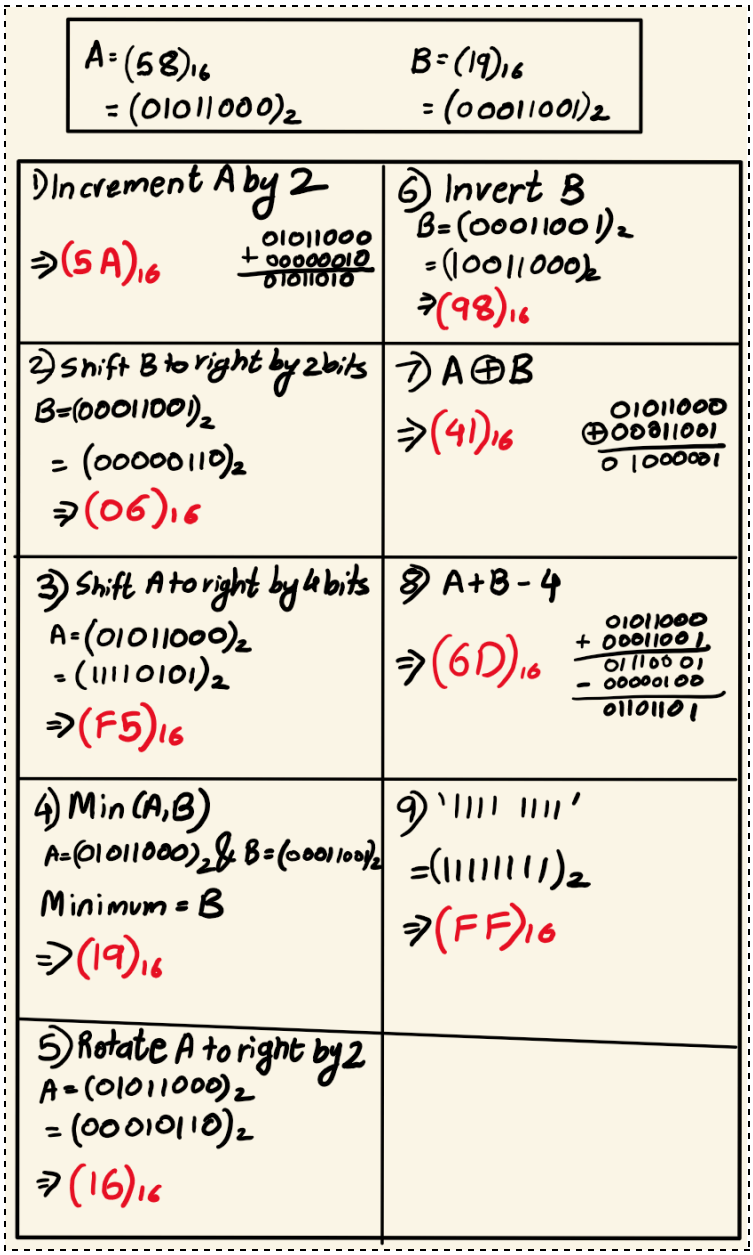


Figure 36: Problem 2 (a) Calculations

Function #	Microcode	Boolean Operation / Function	Binary Result	Hexadecimal Result
1	0000000000000001	Increment A by 2	0101 1010	5A
2	0000000000000010	Shift B to right by two bits, input bit = 0 (SHR)	0000 0110	06
3	0000000000000100	Shift A to right by four bits, input bit = 1 (SHR)	1111 0101	F5
4	0000000000001000	Find the smaller value of A and B and produce the results (Min(A,B)	0101 1000	19
5	0000000000010000	Rotate A to right by two bits (ROR)	0001 0110	16
6	0000000000100000	Invert the bit-significance order of B	1001 1000	98
7	0000000001000000	Produce the result of XORing A and B	0100 0001	41
8	0000000010000000	Produce the summation of A and B, decrease it by 4	0110 1101	6D
9	0000000100000000	Produce all high bits on the output	1111 1111	FF

The match between the expected outputs and the waveform suggests that the obtained outputs are in line with expectations, leading to the conclusion that the output is accurate as they correspond to each other.

Problem 3 (a): Yes or No output

In problem set 3, the assignment was to change the ALU so that it could determine if the student number's digits were odd or even more than nine clock cycles. On a single seven-segment display, "y" is shown if the student number's digit is odd, and "n" is shown if it is even. In order to accomplish this, the ALU code was expanded to include conditional statements and the input "student_id." Rather than performing a boolean function on the current state, which is sent to the ALU between A and B, the ALU only verifies if the input value "student_id" is odd or even.

Figure 37: Input and Output Table

Input	Purpose
Clock	Input signal to carry out operations.
A[7..0]	Input value received from Latch 1.
B[7..0]	Input value received from Latch 2.
OP	16 bit microcode based on the FSM state.
Output	**Purpose**
YN	Yes or No output

Figure 38: Problem 3 (a) BDF

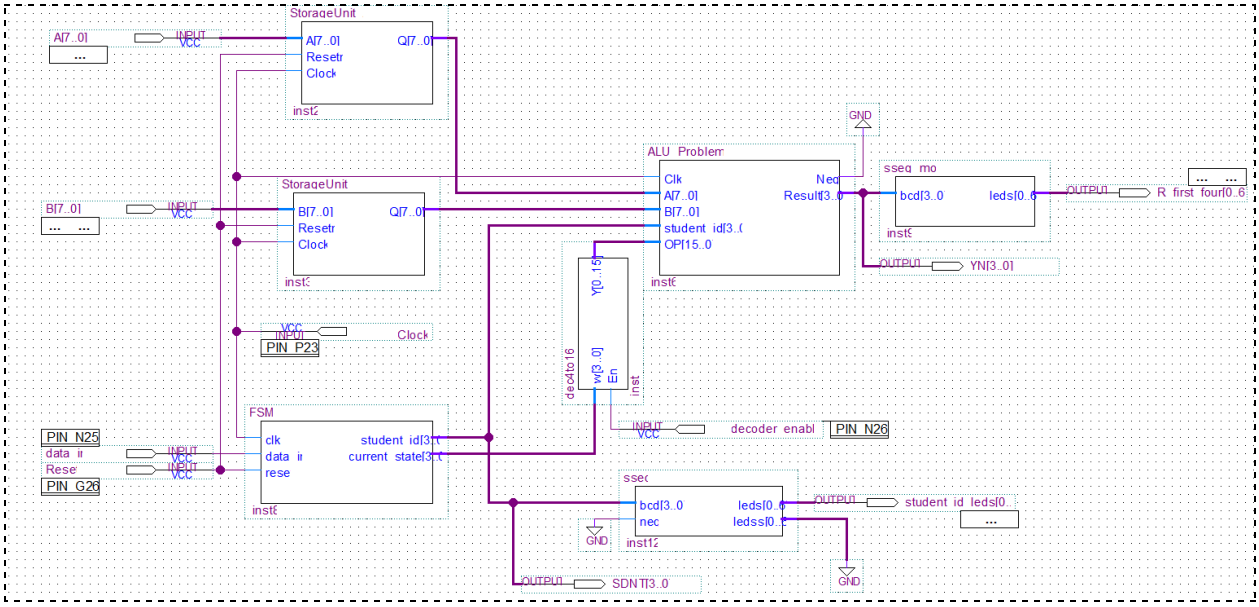
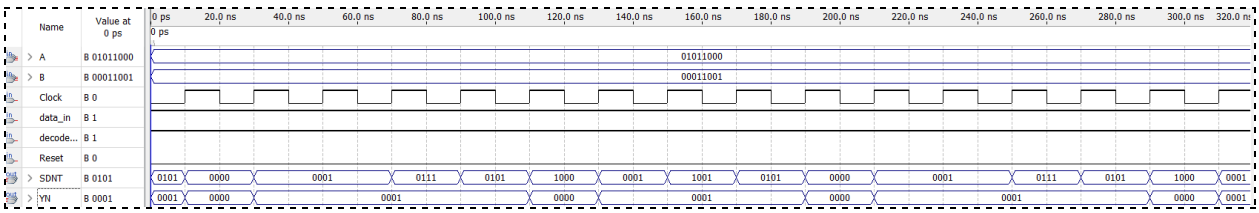


Figure 39: Problem 3 (a) VWF



As seen in this waveform, when the student number is any odd number (5,1,7,5 or 9) it outputs '0001' which when sent to the SSEG outputs '0111011' on the 7-Segment Display showing 'Y' and when its '0000' it outputs '0010101' showing 'N'.

```
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 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       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 fourbits 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;--temporarily store a in reg1 local variable
19  Reg2 <= B;--temporarily store b in reg2 local variable
20  process(Clock,OP)
21      begin
22      if (student_id(0) = '1') then
23            Result <= "0001"; -- Y
24      else
25            Result <= "0000"; -- N
26      end if;
27      end process;
28
29  end calculation;
30
```

Figure 40: Problem 3 (a) ALU VHD

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   ENTITY sseg_mod IS
4   PORT (bcd: IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5       leds : OUT STD_LOGIC_VECTOR(0 TO 6));
6
7   END sseg_mod;
8
9   ARCHITECTURE Behavior OF sseg_mod IS
10  BEGIN
11  PROCESS (bcd)
12  BEGIN
13
14  CASE bcd IS -- abcdefg
15      WHEN "0001" => leds <= not "0111011"; -- y
16      WHEN "0000" => leds <= not "0010101"; -- n
17      When Others => leds <= not "1001111"; -- E
18
19  END CASE;
20  END PROCESS;
21  END Behavior;
```

Figure 40: Problem 3 (a) SSEG VHD

The sseg outputs the letters 'Y' and 'N' to the display unit while the ALU does the calculations of whether each student number is odd or even.

Figure 41: SSEG outputs

Function #	Microcode	Boolean Operation / Function
1	0000000000000001	5/y
2	0000000000000010	0/n
3	0000000000000100	1/y
4	0000000000001000	1/y
5	0000000000010000	7/y
6	0000000000100000	5/y
7	0000000001000000	8/n
8	0000000010000000	9/y
9	0000000100000000	1/y

Conclusion:

In Lab 6, we successfully built three versions of an Arithmetic Logic Unit (ALU) using knowledge from previous labs and on sequential circuits. The lab's objective was to construct a processor by integrating various components, including latches, a Finite State Machine (FSM), a 4:16 decoder, and an ALU.

The latches stored 8-bit inputs A and B, while the FSM, operating on Moore logic, and the 4:16 decoder formed the control unit. The ALU executed operations based on these inputs. The decoder and 7-segment display, being combinational circuits, operated independently of these inputs. However, the FSM, ALU, and latches, being sequential circuits, required a clock input that alternated between 1 and 0.

The lab was successful, with all components functioning cohesively to produce the expected results. The VHDL codes and block diagrams for each component were accurately compiled, and the waveform diagrams showed precise values. This confirmed the functionality of each component and the successful creation of a fully operational general-purpose processor.