TABLE OF CONTENTS:

Introduction

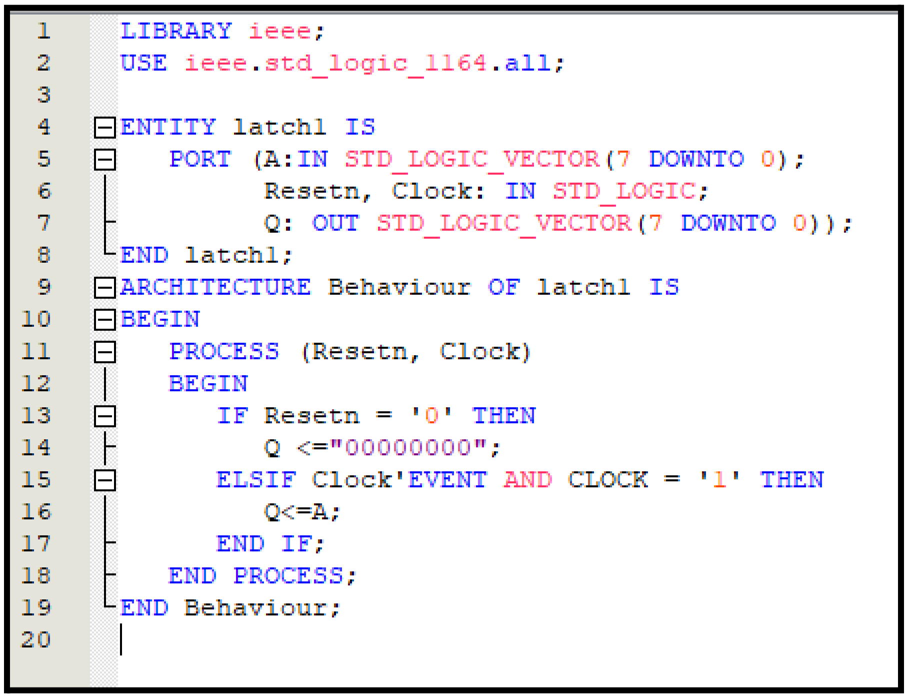
This lab called the *Design of a Simple General-Purpose Processor* was a result of concepts covered from previous labs. The components included Seven Segment Displays (Sseg), Decoder, and the Finite State Machine (FSM) were used from previous labs. New components that were implemented in this lab included the Storage Unit (Register) and the Arithmetic and Logic Unit (ALU). The main objective of the lab was to implement and design a functional ALU which should be able to execute multiple operations and display it on the FPGA board.

Components: Latch 1, Latch 2, 4:16 Decoder, FSM

Latch 1

The latches in this lab were used as temporary storage units. This means latch 1 would temporarily store input A which was the third and fourth last digits of my student number. In this case, $A(58)_{16} = (0101\ 1000)_2$. It takes an 8-bit binary number and reads the values on the rising edge of the clock as inputs and passes the values to the output on the next rising edge of the clock signal. These outputs then proceed towards the ALU component as inputs.

Figure 0: Student ID



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

ENTITY latch1 IS
    PORT (A:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
          Resetn, Clock: IN STD_LOGIC;
          Q: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END latch1;
ARCHITECTURE Behaviour OF latch1 IS
BEGIN
    PROCESS (Resetn, Clock)
    BEGIN
        IF Resetn = '0' THEN
          Q <="00000000";
        ELSIF Clock'EVENT AND CLOCK = '1' THEN
          Q<=A;
        END IF;
    END PROCESS;
END Behaviour;
```

Figure 1: VHDL Code for Latch 1

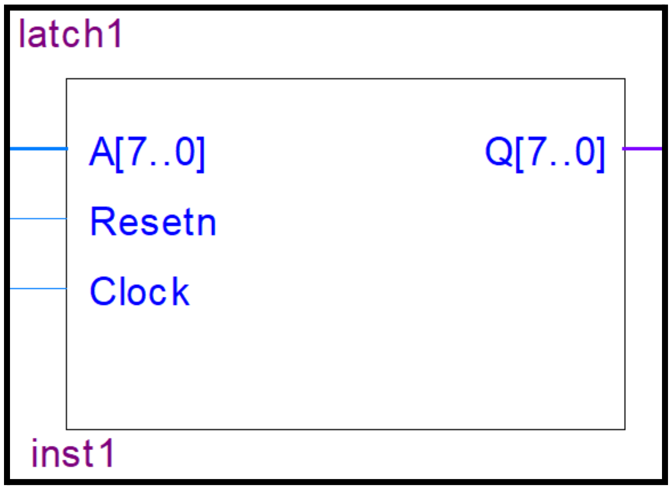


Figure 2: Latch 1 BDF

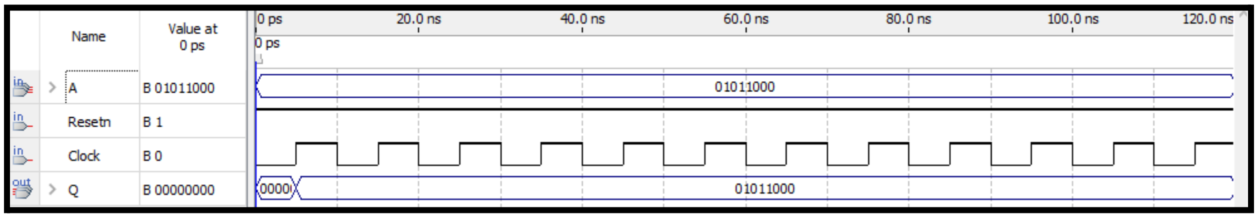
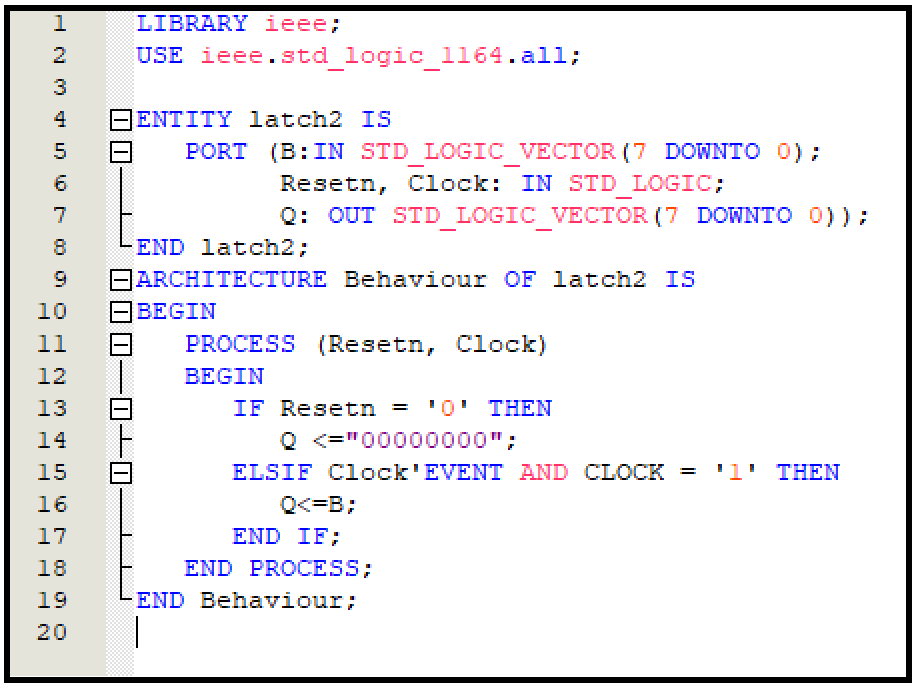


Figure 3: Latch 1 Waveform (Input A)

Latch 2

Latch 2 has the same process and design as Latch 1. It also stores an 8-bit binary number, but it stores the second and first last digits of my student number. This means it stores the values of B. In this case, $B(87)_{16} = (1000\ 0111)_2$.



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

ENTITY latch2 IS
    PORT (B:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            Resetn, Clock: IN STD_LOGIC;
            Q: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END latch2;
ARCHITECTURE Behaviour OF latch2 IS
BEGIN
    PROCESS (Resetn, Clock)
    BEGIN
        IF Resetn = '0' THEN
            Q <="00000000";
        ELSIF Clock'EVENT AND CLOCK = '1' THEN
            Q<=B;
        END IF;
    END PROCESS;
END Behaviour;
```

Figure 4: VHDL Code for Latch 2

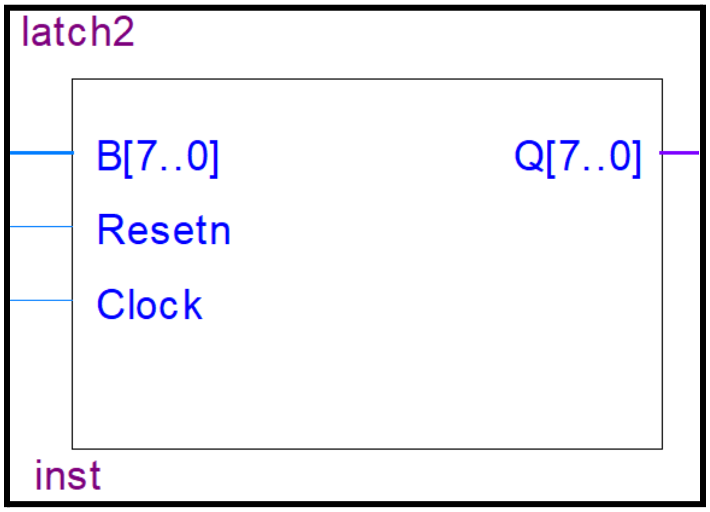


Figure 5: Latch 2 BDF

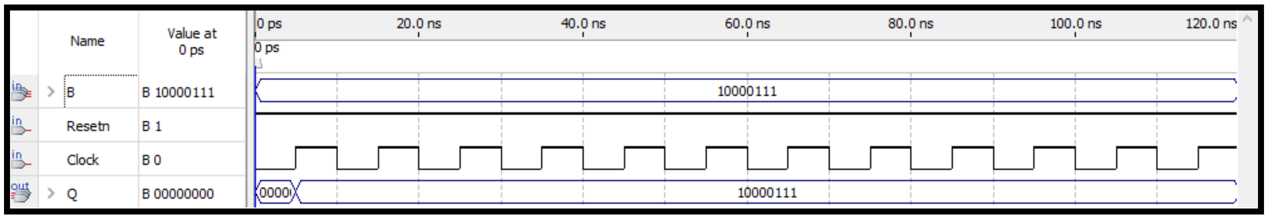


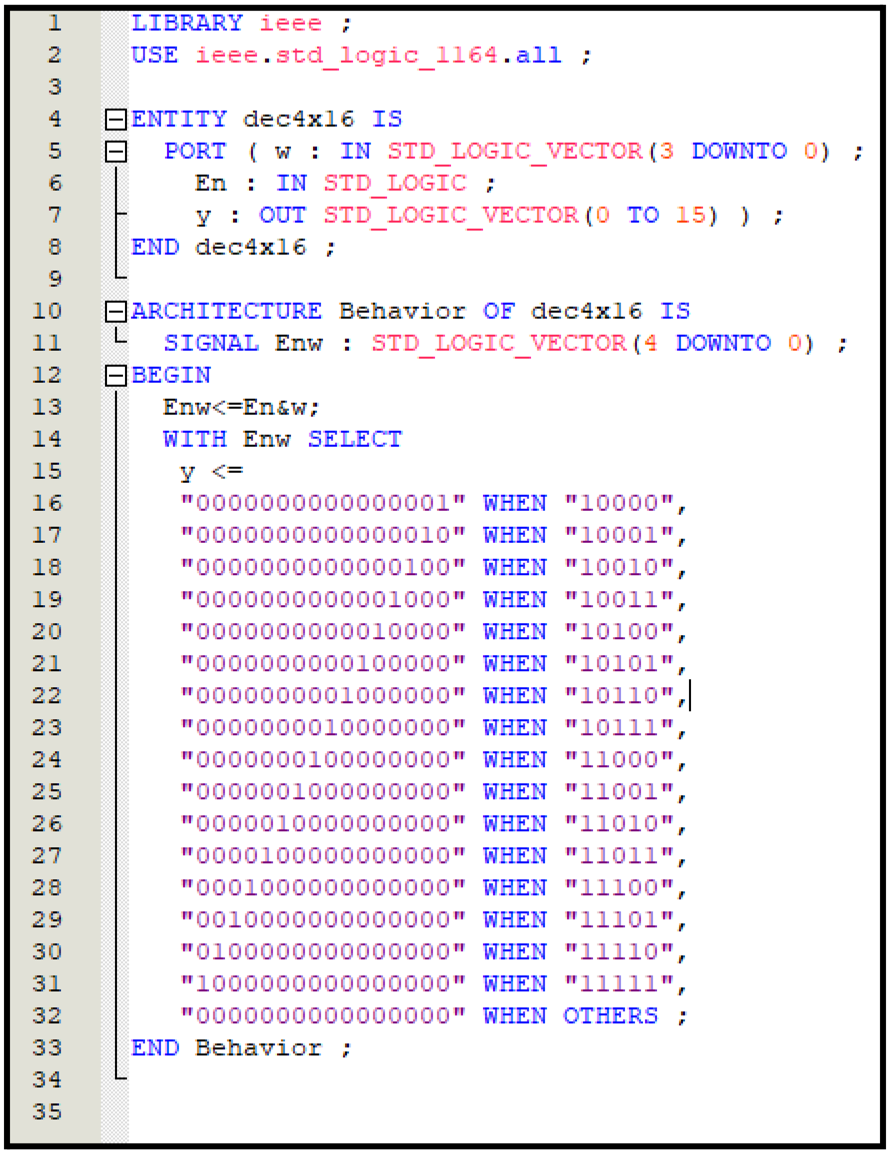
Figure 6: Latch 2 Waveform (Input B)

4:16 Decoder

The 4:16 decoder was designed to take a 4-bit input and turn it into a 16-bit output. It is one of the two control units needed for operations to work in order for it to be executed by the ALU. The 4-bit input is taken from the Finite State Machine (FSM) and then converts the state into a 16-bit microcode. These outputs then proceed towards the ALU as inputs.

w3	w2	w1	w0	y15	y14	y13	y12	y11	y10	y9	y8	y7	y6	y5	y4	y3	y2	y1	y0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	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
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1: Truth Table for 4:16 Decoder



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

ENTITY dec4x16 IS
  PORT ( w : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
     En : IN STD_LOGIC ;
     y : OUT STD_LOGIC_VECTOR(0 TO 15) ) ;
END dec4x16 ;

ARCHITECTURE Behavior OF dec4x16 IS
  SIGNAL Enw : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
  Enw<=En&w;
  WITH Enw SELECT
   y <=
   "0000000000000001" WHEN "10000",
   "0000000000000010" WHEN "10001",
   "0000000000000100" WHEN "10010",
   "0000000000001000" WHEN "10011",
   "0000000000010000" WHEN "10100",
   "0000000000100000" WHEN "10101",
   "0000000001000000" WHEN "10110",
   "0000000010000000" WHEN "10111",
   "0000000100000000" WHEN "11000",
   "0000001000000000" WHEN "11001",
   "0000010000000000" WHEN "11010",
   "0000100000000000" WHEN "11011",
   "0001000000000000" WHEN "11100",
   "0010000000000000" WHEN "11101",
   "0100000000000000" WHEN "11110",
   "1000000000000000" WHEN "11111",
   "0000000000000000" WHEN OTHERS ;
END Behavior ;
```

Figure 7: VHDL Code for 4:16 Decoder

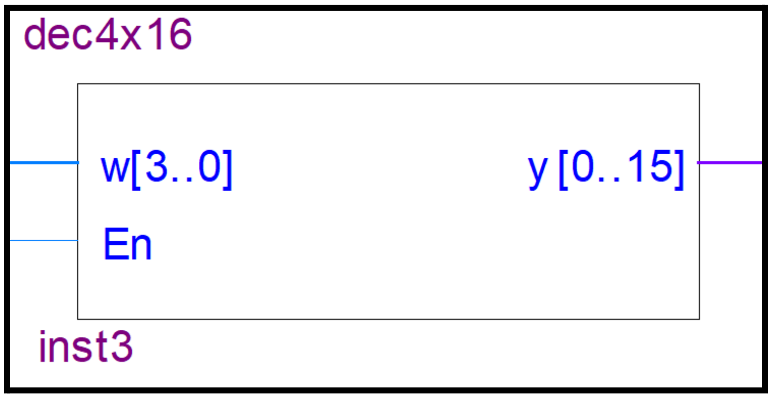


Figure 8: 4:16 Decoder BDF

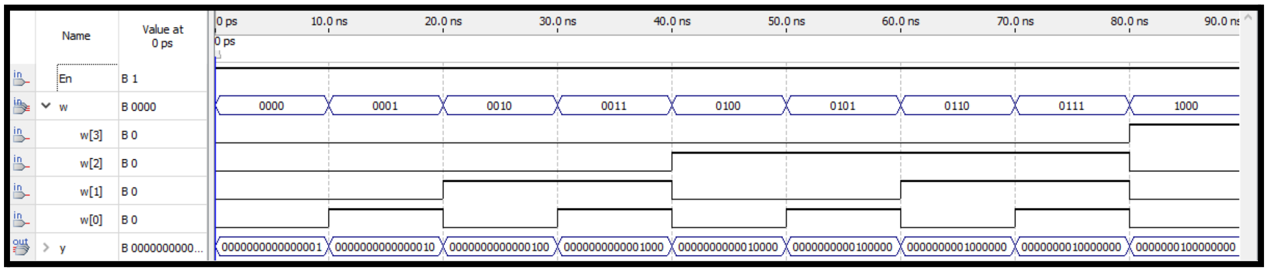


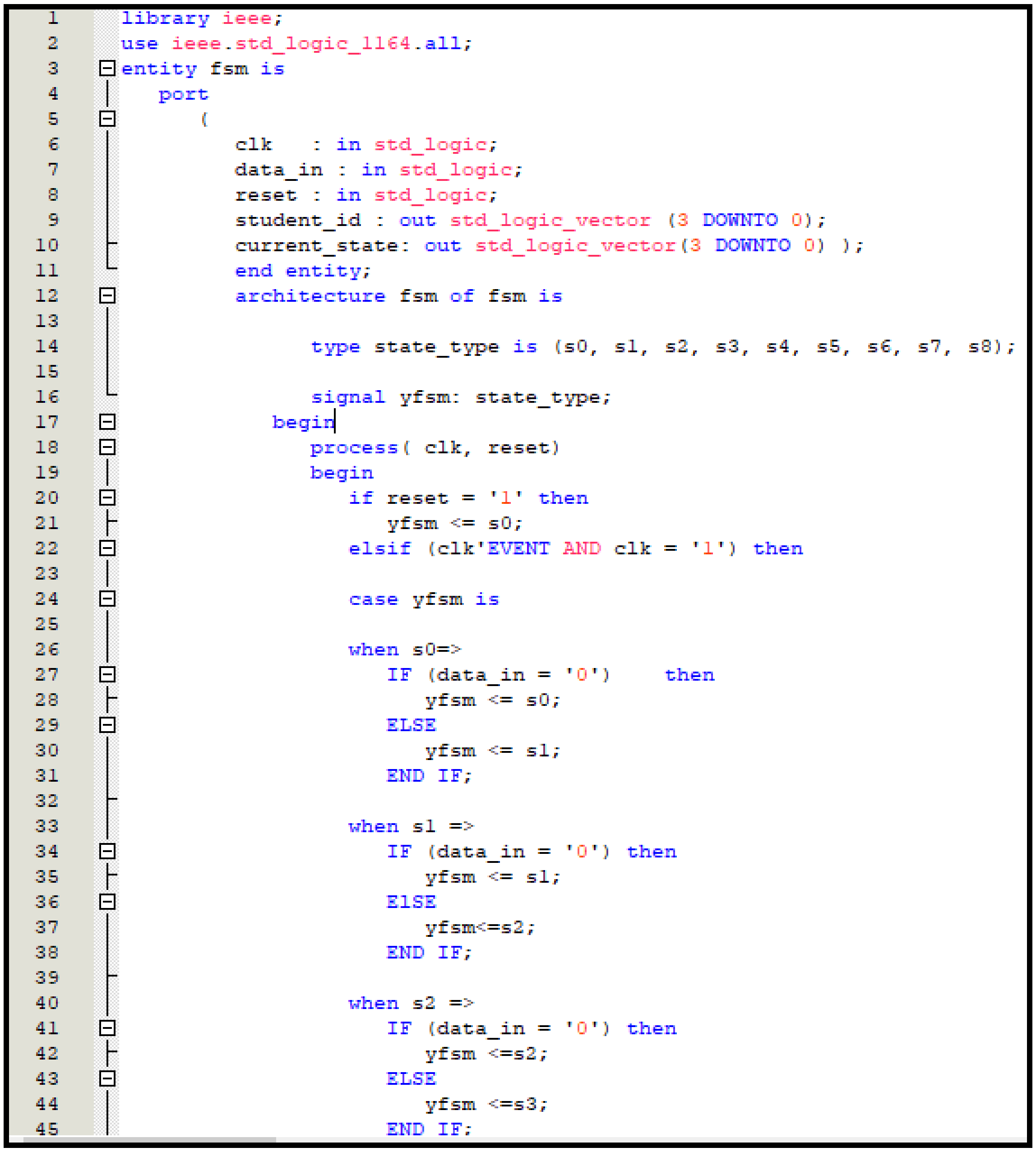
Figure 9: 4:16 Decoder Waveform

Finite State Machine (FSM)

The Finite State Machine (FSM) is the one of the two control units needed for operations to work in order for it to be executed by the ALU. The FSM cycles through 9 states, starting from state 0 and ending at state 8. The FSM takes the clock, reset and data in as inputs, but its outputs include the student number and current state. Each specific state corresponds to each digit in the student number through the clock cycle. The current state of the FSM is outputted as a 4-bit number.

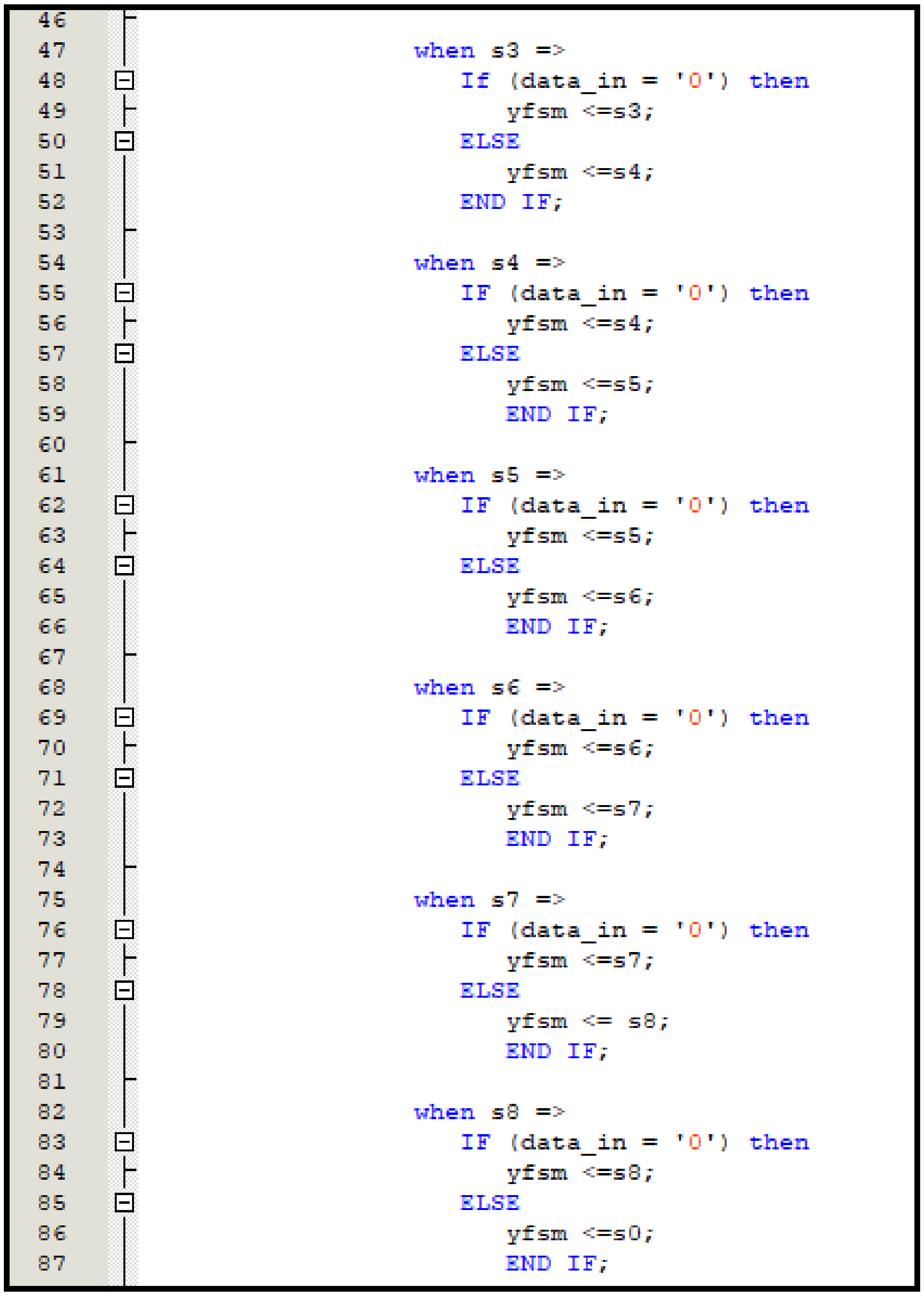
		Current State				Next State			
Student ID	State	Q_3	Q_2	Q_1	Q_0	Q_3	Q_3	Q_3	Q_3
5	0	0	0	0	0	0	0	0	1
0	1	0	0	0	1	0	0	1	0
1	2	0	0	1	0	0	0	1	1
1	3	0	0	1	1	0	1	0	0
6	4	0	1	0	0	0	1	0	1
5	5	0	1	0	1	0	1	1	0
8	6	0	1	1	0	0	1	1	1
8	7	0	1	1	1	1	0	0	0
7	8	1	0	0	0	0	0	0	0

Table 2: Truth Table for FSM



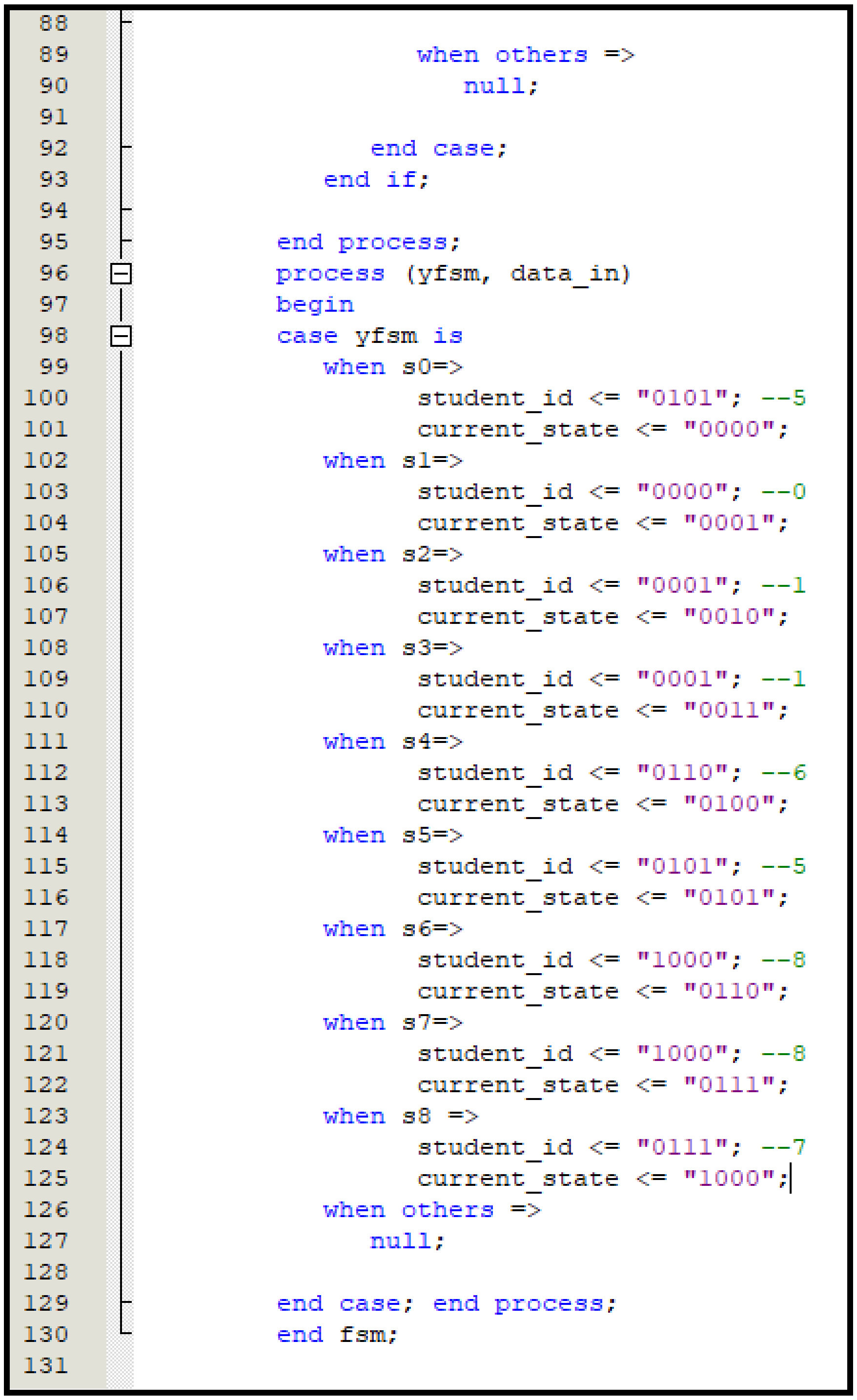
```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity fsm is
   port
      (
         clk   : in std_logic;
         data_in : in std_logic;
         reset : in std_logic;
         student_id : out std_logic_vector (3 DOWNTO 0);
         current_state: out std_logic_vector(3 DOWNTO 0) );
         end entity;
         architecture fsm of fsm is

               type state_type is (s0, s1, s2, s3, s4, s5, s6, s7, s8);

               signal yfsm: state_type;
            begin
               process( clk, reset)
               begin
                  if reset = '1' then
                     yfsm <= s0;
                  elsif (clk'EVENT AND clk = '1') then

                  case yfsm is

                  when s0=>
                     IF (data_in = '0')     then
                        yfsm <= s0;
                     ELSE
                        yfsm <= s1;
                     END IF;

                  when s1 =>
                     IF (data_in = '0') then
                        yfsm <= s1;
                     E1SE
                        yfsm<=s2;
                     END IF;

                  when s2 =>
                     IF (data_in = '0') then
                        yfsm <=s2;
                     ELSE
                        yfsm <=s3;
                     END IF;
```

Figure 10: VHDL Code for FSM (Part 1)



```
                    when s3 =>
                       If (data_in = '0') then
                          yfsm <=s3;
                       ELSE
                          yfsm <=s4;
                       END IF;

                    when s4 =>
                       IF (data_in = '0') then
                          yfsm <=s4;
                       ELSE
                          yfsm <=s5;
                          END IF;

                    when s5 =>
                       IF (data_in = '0') then
                          yfsm <=s5;
                       ELSE
                          yfsm <=s6;
                          END IF;

                    when s6 =>
                       IF (data_in = '0') then
                          yfsm <=s6;
                       ELSE
                          yfsm <=s7;
                          END IF;

                    when s7 =>
                       IF (data_in = '0') then
                          yfsm <=s7;
                       ELSE
                          yfsm <= s8;
                          END IF;

                    when s8 =>
                       IF (data_in = '0') then
                          yfsm <=s8;
                       ELSE
                          yfsm <=s0;
                          END IF;
```

Figure 11: VHDL Code for FSM (Part 2)



```

                    when others =>
                       null;

                 end case;
              end if;

           end process;
           process (yfsm, data_in)
           begin
           case yfsm is
              when s0=>
                    student_id <= "0101"; --5
                    current_state <= "0000";
              when s1=>
                    student_id <= "0000"; --0
                    current_state <= "0001";
              when s2=>
                    student_id <= "0001"; --1
                    current_state <= "0010";
              when s3=>
                    student_id <= "0001"; --1
                    current_state <= "0011";
              when s4=>
                    student_id <= "0110"; --6
                    current_state <= "0100";
              when s5=>
                    student_id <= "0101"; --5
                    current_state <= "0101";
              when s6=>
                    student_id <= "1000"; --8
                    current_state <= "0110";
              when s7=>
                    student_id <= "1000"; --8
                    current_state <= "0111";
              when s8 =>
                    student_id <= "0111"; --7
                    current_state <= "1000";
              when others =>
                 null;

           end case; end process;
           end fsm;

```

Figure 12: VHDL Code for FSM (Part 3)

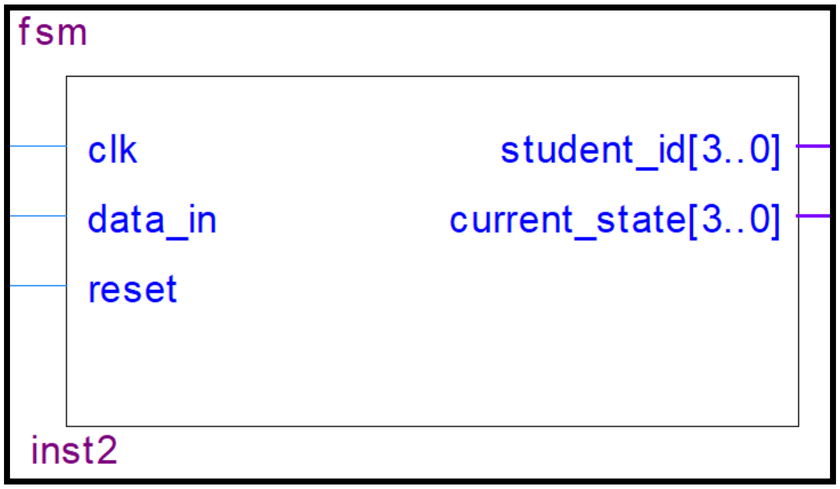


Figure 13: FSM BDF

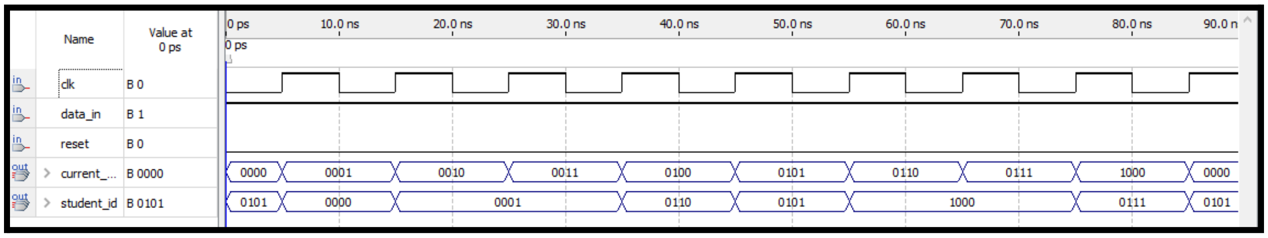


Figure 14: FSM Waveform

Problem 1 - ALU 1

The operations are carried out using the ALU which is the central component. There are five inputs which include the Student ID number, OP, the A values and B values, and the clock. The student ID comes from the output of the FSM, the output of the decoder gives the OP, and the A and B values are from the latches. In order for the functions to switch the clock is used. The OP has a 16-bit number which comes from the decoder and is used as a microcode which tells the ALU what the current state is. Each state, represented by the 16-bit microcode, has its own function.

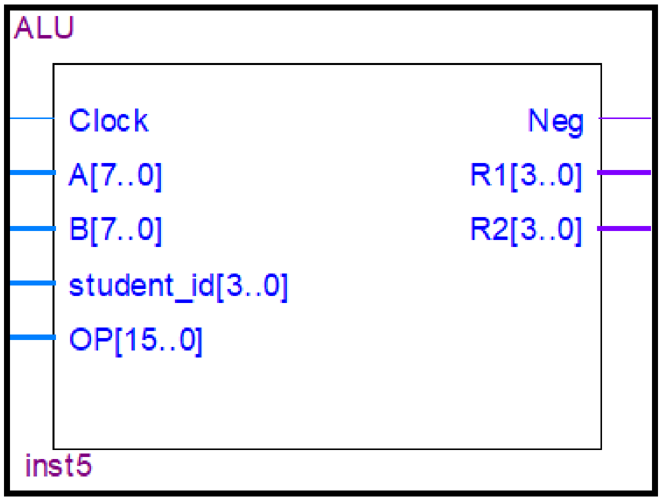


Figure 15: ALU BDF

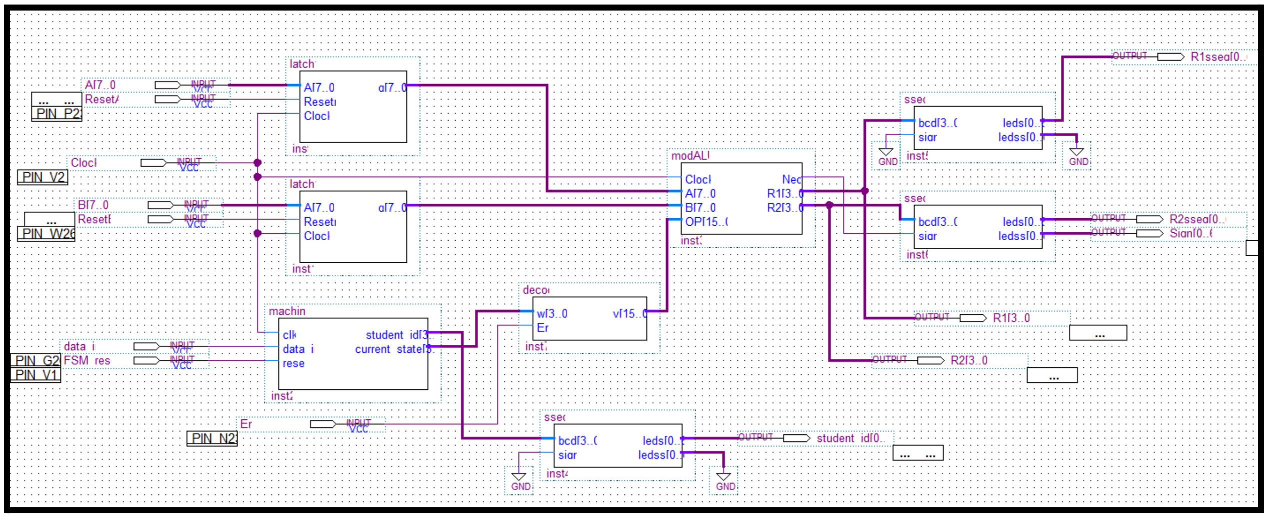


Figure 16: Problem 1 BDF

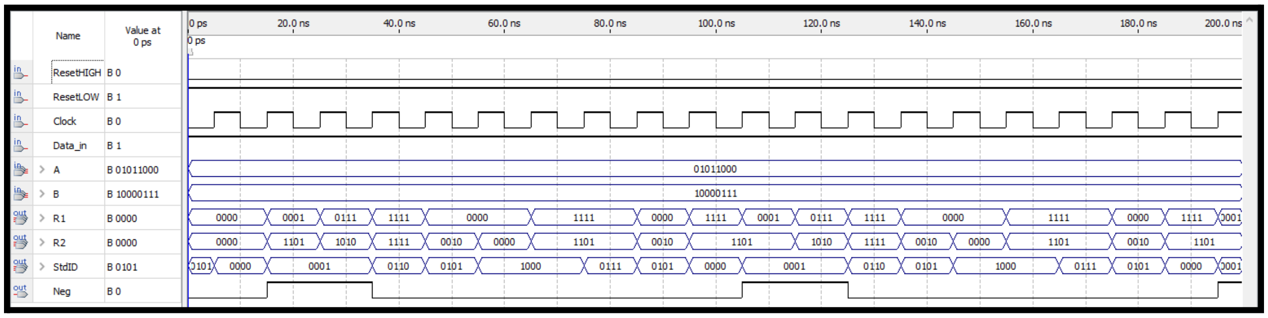
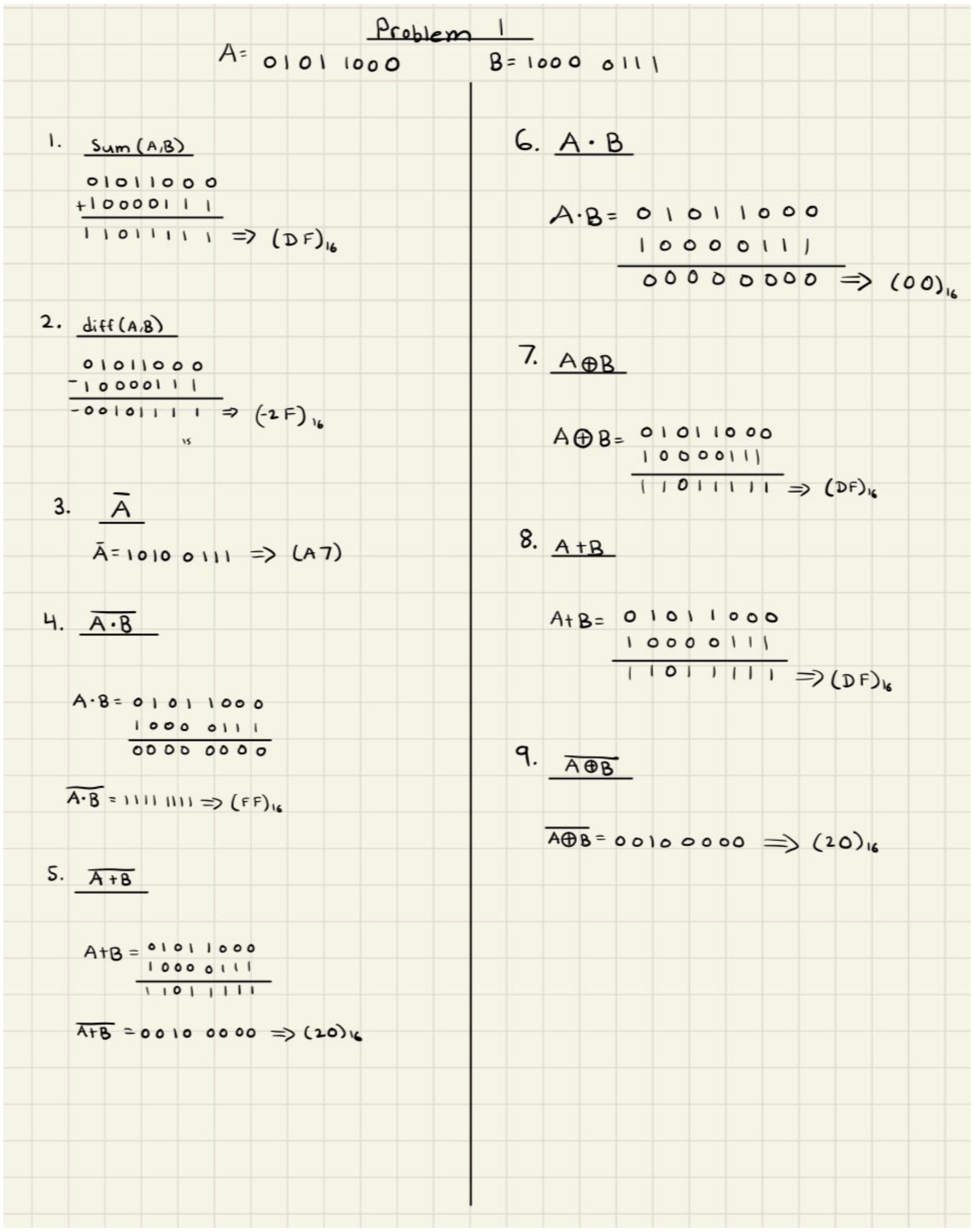


Figure 17: Problem 1 Waveform

Upon analyzing **Figure 17**, it is shown that the first output is “0000” which is not a result from the expected microcode according to the operations. This is due to the fact that there is a delay in the result between the decoder and the ALU. The ALU gets operations done upon the rising edge of the clock, and on the other hand the decoder does not. This occurs because the ALU does the operation based off of its previous state from the FSM. This is the explanation behind the output of “0000” in the beginning as the FSM has no state previous to 0 resulting in the output. This lag occurs in ALU 2 and ALU 3.

Function #	**Microcode**	**Operation / Function**	**Binary Result**	**Hexadecimal Result**
1	0000000000000001	Sum(**A**, **B**)	1101 1111	DF
2	0000000000000010	Diff(**A**, **B**)	-0010 1111	-2F
3	0000000000000100	$\overline{A}$	1010 0111	A7
4	0000000000001000	$\overline{A \cdot B}$	1111 1111	FF
5	0000000000010000	$\overline{A + B}$	0010 0000	20
6	0000000000100000	$A \cdot B$	0000 0000	00
7	0000000001000000	$A \oplus B$	1101 1111	DF
8	0000000010000000	$A + B$	1101 1111	DF
9	000000010000000	$\overline{A \oplus B}$	0010 0000	20

Table 3: Problem 1 ALU Operations



Problem 1

A= 0101 1000   B= 1000 0111

1. Sum(A,B)

```
 01011000
+10000111
 11011111  => (DF)16
```

2. diff(A,B)

```
 01011000
-10000111
-00101111  => (-2F)16
```

3. $\overline{A}$

$\bar{A}$= 1010 0111 => (A7)

4. $\overline{A\cdot B}$

```
A·B= 0101 1000
     1000 0111
     0000 0000
```

$\overline{A\cdot B}$ = 1111 1111 => $(FF)_{16}$

5. $\overline{A+B}$

```
A+B = 0101 1000
      1000 0111
      1101 1111
```

$\overline{A+B}$ = 0010 0000 => $(20)_{16}$

6. A · B

```
A·B= 01011000
     10000111
     00000000  => (00)16
```

7. A⊕B

```
A⊕B= 01011000
     10000111
     11011111  => (DF)16
```

8. A+B

```
A+B= 01011000
     10000111
     11011111  => (DF)16
```

9. $\overline{A\oplus B}$

$\overline{A\oplus B}$ = 00100000 => $(20)_{16}$

Figure 18: Hand Computations for Problem 1

Problem 2 - ALU 2 (Option C)

The ALU for problem 2 has the same structure as problem 1. The difference between the two problems is the operations that occur. In this case Option C was assigned and the code has to correspond according to the table. The functionality for the ALU of problem 2 follows **Table 4.**

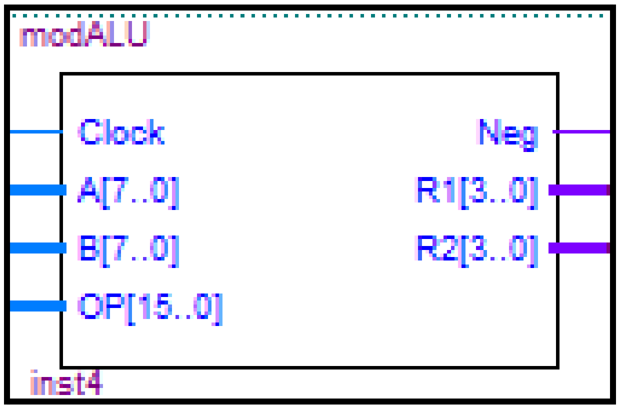


Figure 19: modALU BDF

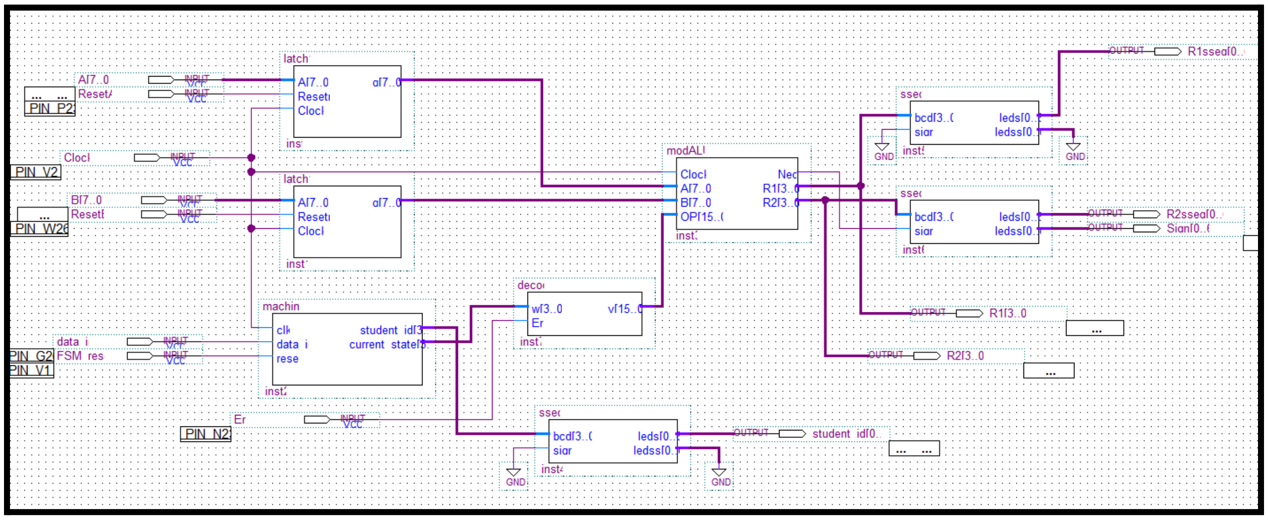


Figure 20: Problem 2 BDF

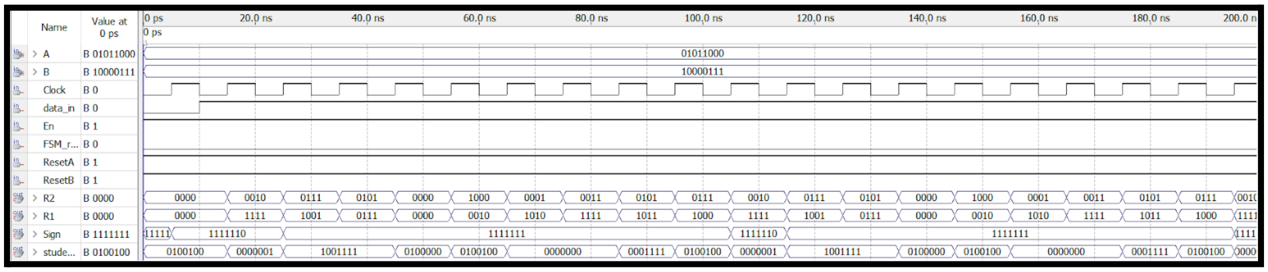
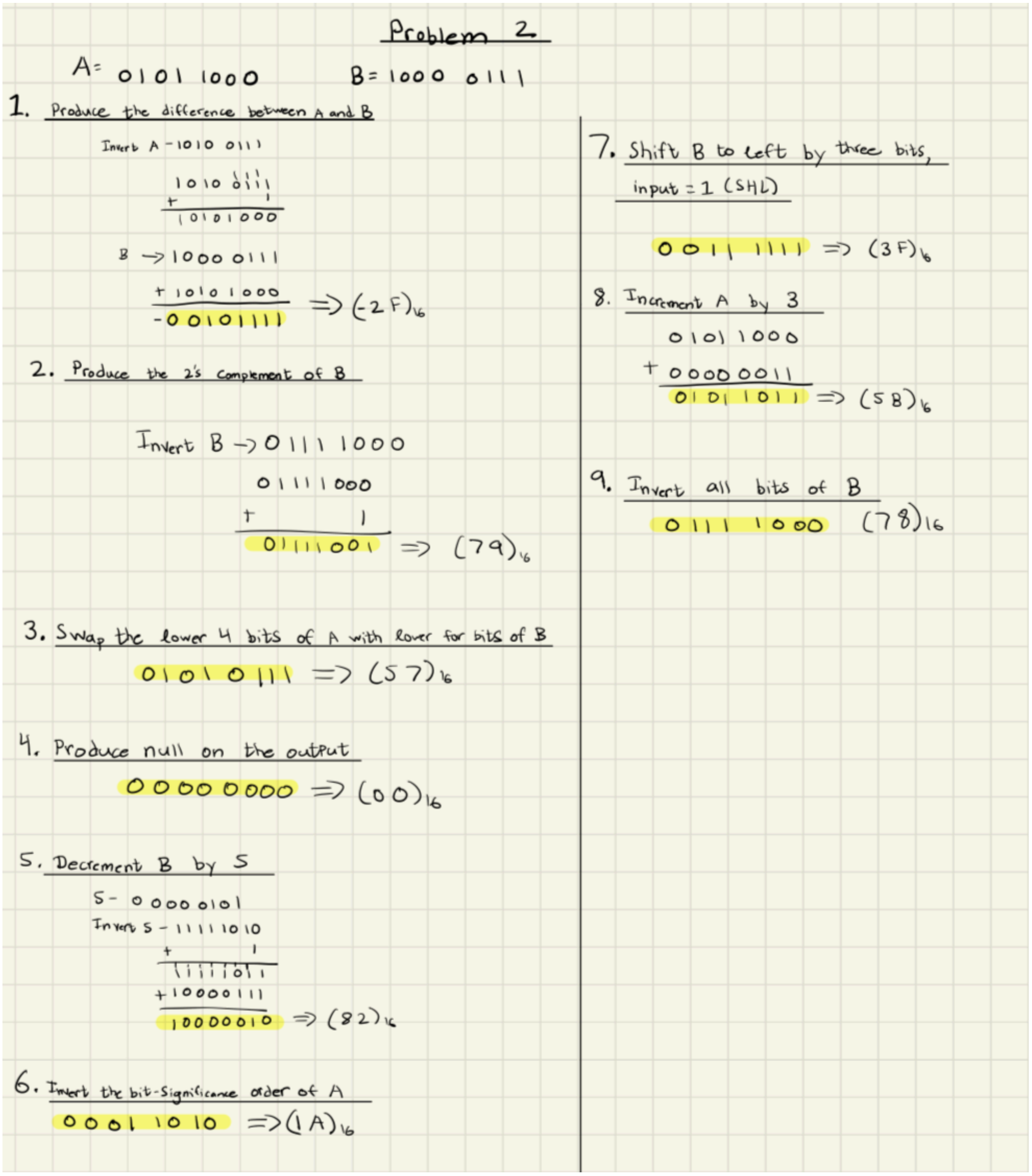


Figure 21: Problem 2 Waveform

Function #	**Microcode**	**Operation / Function**	**Binary Result**	**Hexadecimal Result**
1	0000000000000001	Produce the difference between **A** and **B**	-0010 1111	-2F
2	0000000000000010	Produce the 2's complement of **B**	0111 1001	79
3	0000000000000100	Swap the lower 4 bits of **A** with lower 4 bits of **B**	0101 0111	57
4	0000000000001000	Produce null on the output	0000 0000	00
5	0000000000010000	Decrement **B** by 5	1000 0010	82
6	0000000000100000	Invert the bit-significance order of **A**	0001 1010	1A
7	0000000001000000	Shift **B** to left by three bits, input bit = 1 (SHL)	0011 1111	3F
8	0000000010000000	Increment **A** by 3	0101 1011	5B
9	000000010000000	Invert all bits of **B**	0111 1000	78

Table 4: Problem 2 ALU Operations


Problem 2

A= 0101 1000   B= 1000 0111

1. Produce the difference between A and B

Invert A - 1010 0111

```
  1010 0111
+         1
  10101000
```

B -> 1000 0111

```
 + 1010 1000
 - 0 0101111   => $(-2F)_{16}$
```

2. Produce the 2's complement of B

Invert B -> 0111 1000

```
  01111000
+        1
  01111001   => $(79)_{16}$
```

3. Swap the lower 4 bits of A with lower for bits of B

0101 0111 => $(57)_{16}$

4. Produce null on the output

00000000 => $(00)_{16}$

5. Decrement B by 5

```
5 - 0000 0101
Invert 5 - 11111010
        +        1
          11111011
        + 10000111
          10000010   => $(82)_{16}$
```

6. Invert the bit-significance order of A

0001 1010 => $(1A)_{16}$

7. Shift B to left by three bits, input = 1 (SHL)

0011 1111 => $(3F)_{16}$

8. Increment A by 3

```
  0101 1000
+ 0000 0011
  0101 1011   => $(5B)_{16}$
```

9. Invert all bits of B

0111 1000 $(78)_{16}$

Figure 22: Hand Computations for Problem 2

Problem 3 - ALU 3 (Option C)

For problem 3 Option C was assigned and the code had to correspond to this assigned option. In this problem set the student number would determine if its binary value is an odd parity. If it is an odd parity the Sseg would output “y” otherwise “n”. The ALU will either output “0001” or “0000” and it would be forwarded to the modified Sseg to output the value onto the Sseg. An example is “0101” whose output would be “n” in the Sseg which is also “1000100”.

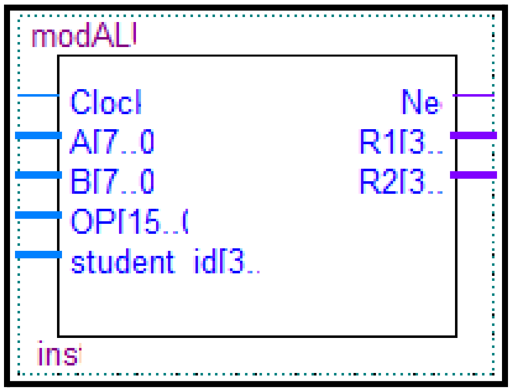


Figure 23: modALU BDF

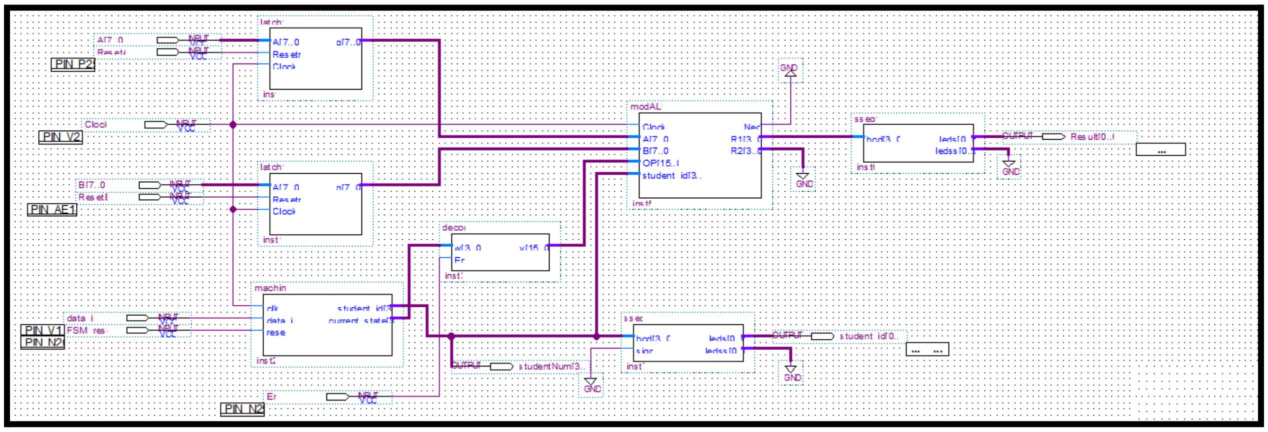


Figure 24: Problem 1 BDF

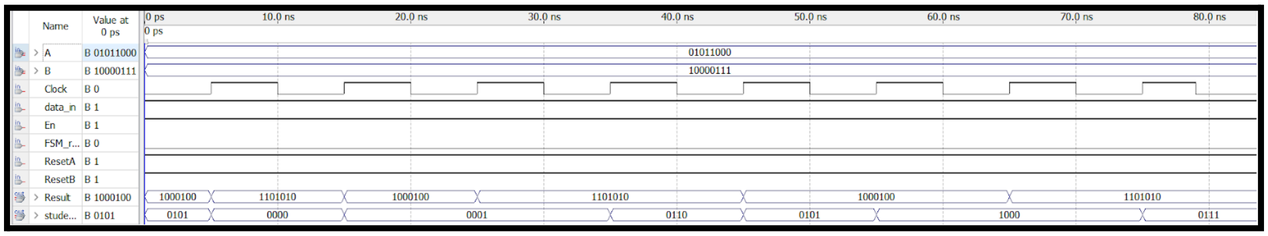


Figure 25: Problem 3 Waveform

Student Number	Student Number in binary	Current State	Microcode	Result (If odd parity then ‘y’ else ‘n’)	FPGA Result
5	0101	0	0000000000000001	1000100	n
0	0000	1	0000000000000010	1101010	y
1	0001	2	0000000000000100	1101010	y
1	0001	3	0000000000001000	1101010	y
6	0110	4	0000000000010000	1000100	n
5	0101	5	0000000000100000	1000100	n
8	1000	6	0000000001000000	1101010	y
8	1000	7	0000000010000000	1101010	y
7	0111	8	0000000100000000	1000100	y

Table 5: Problem 3 ALU Operations

Conclusion

The main focus of this lab was to implement and design an ALU component which executed different operations depending on the state, the inputs and the written code in the VHDL environment. This lab was completed successfully as 3 different microprocessors were created, compiled and tested. Knowledge from previous labs was used in order to create a functional General-Purpose Processor. In order for this to work multiple parts had to be put together such as the storage unit, control unit, the seven-segment display, and the arithmetic and logic unit. Upon putting these pieces together the implementation onto the FPGA board was successful.