Department of Electrical and Computer Engineering

Course Number	**COE328**
Course Title	**Digital System**
Semester/Year	**F/23**
Instructor	Reza Sedaghat
TA Name	**Menglu Li**

Lab/Tutorial Report No.	**#6**

Report Title	**COE328 Lab 6: Design of a Simple General-Purpose Processor**

Section No.	**6**
Submission Date	**2023-12-03**
Due Date	**2023-12-04**

Student Name	Student ID	Signature*
Fareez Mir	**59472**	

Table of Contents

Introduction

The objective of this lab experiment is to design and build an Arithmetic Logic Unit (ALU) using VHDL code, which is then represented through a block diagram with all the components working in tandem to create the unit shown in *Figure 1.0*, known as a General Purpose Processor (GPU). The ALU consists of two latches for memory storage, a control unit comprising a Finite State Machine (FSM), a 4-16 Decoder, and a seven-segment display for output. The ALU operates on two 8-bit inputs to perform arithmetic and logical functions. The latches store the 8-bit binary values, providing them as input to the ALU's core. The control unit, with the FSM and Decoder, directs the core to execute specific tasks, where commands are issued using predefined microcode values. The output of these operations is then displayed on a seven-segment display, converting the binary results into a readable format.

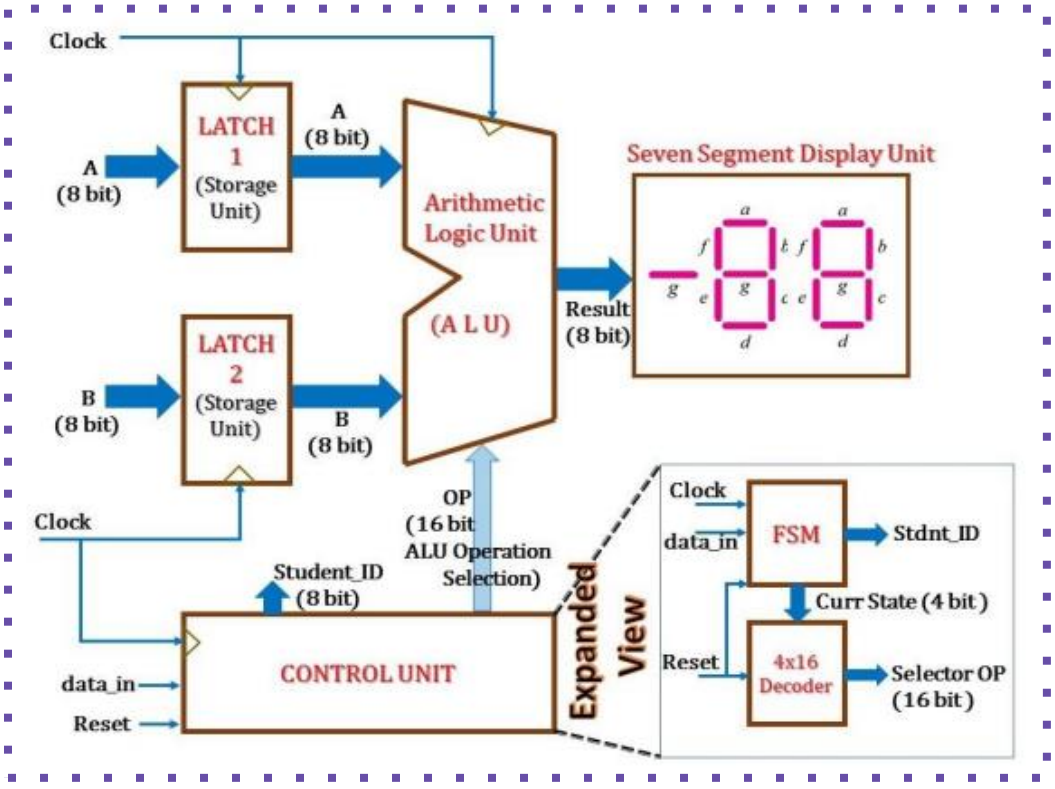


Figure 1.0: ALU Block Diagram of all Components in Place

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

This lab uses two inputs, A and B that are stored into two latch components for arithmetic purposes the ALU will use. Based on the student number of "*501159472*", $A = (94)_{16}$ and $B = (72)_{16}$. Converting both hexadecimal values gives the values of $A = (\mathbf{1}001\ 0\mathbf{1}00)_2$ and $B = (0\mathbf{111}\ 00\mathbf{1}0)_\mathbf{2}$ that will be used for the experiment.

1.1 - Latch

The first two components, Latch 1 and Latch 2, are designed as the Storage Units, known as Registers, to temporarily store inputs and pass them to the rest of the components of the system. These two bit registers are designed to store two 8-bit binary inputs, known as **A** for Latch 1 and **B** for Latch 2, where it reads these values on a rising clock edge trigger, and then passes these values to the output on the next rising edge of the clock. Finally, the latch includes an asynchronous reset, where a '0' clears the output to '00000000', while a '1' allows the input to pass through, independent of the clock state. The truth table below is true for both latches used for the final unit of the GPU.

Reset	Clock	A (Any 8-bit Input)	Q (Output)
0	X	X	00000000
1	0	A	Q (no change)
1	1	A	A

Figure 1.10: Truth Table of the Latch, using A as the example input. X is used as a "don't care" variable, as the value of clock or the input does not change the outcome of Reset = '0'.

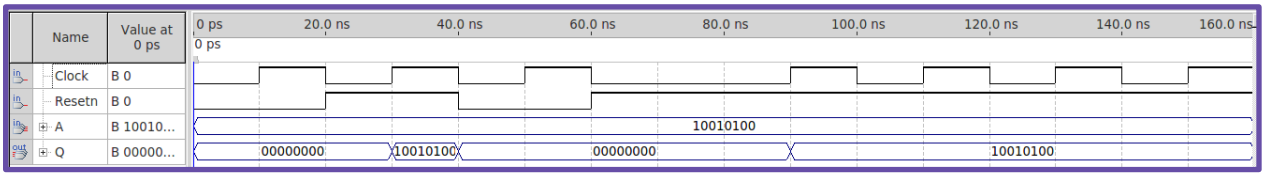


Figure 1.11: Waveform of the Latch, using A as the example input, where A = 10010100. Variance has been added to the waveform to add different cases.

As per the truth table of the Latch in *Figure 1.1*, the outcome of the waveform is as expected. The initial state of *Q* is "00000000", which is maintained during the first **`Clock`** cycle since **`Resetn`** is inactive (0). On the subsequent rising edge of the clock, the **`Resetn`** signal becomes active (1), allowing the output *Q* to take on the value of *A*, which is "10010100". This behavior is consistent with a latch that captures the input data on a rising clock edge when **`Resetn`** is active.

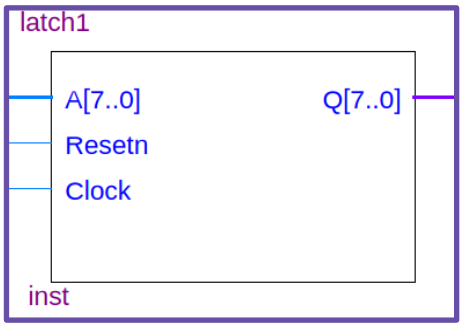


Figure 1.12: Block schematic of the latch component.

```
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); --8 bit A input
6              Resetn, Clock : IN STD_LOGIC; --1 bit clock input and 1 bit reset input bit
7              Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ); -- 8 bit output
8   END latch1;
9
10  ARCHITECTURE Behavior OF latch1 IS
11  BEGIN
12      PROCESS (Resetn, Clock) --Process takes reset and clock as inputs
13      BEGIN
14          IF Resetn = '0' THEN -- when reset input is '0' the latches does not operate
15              Q <= "00000000";
16          ELSIF Clock'EVENT AND Clock = '1' THEN -- level sensitive based on clock
17              Q <= A;
18          END IF;
19      END PROCESS;
20  END Behavior;
```

Figure 1.13: VHDL code for both latch 1 and latch 2 as they function identically apart from the value that they each store.

1.2 - FSM: Moore Machine

The first component used for the ALU core's control unit is the Finite State Machine, which in more specifics, is using a Moore implementation. The FSM acts as an up-counter, cycling from states 0 to 8 and then returning to state 0. This FSM is driven by a clock signal and outputs a 4-bit signal indicating the current state. This output, known as **`current_state`**, is forwarded to a decoder component within the system. The design also includes the **`student_id`**, which is displayed on the 7-segment display.

Present State $y_3y_2y_1y_0$		Next State W = 0 $Y_3Y_2Y_1Y_0$	Next State W = 1 $Y_3Y_2Y_1Y_0$	Output $z_3z_2z_1z_0$	
S_0	**0000**	0000	**0001**	**0101**	d_0
S_1	**0001**	0001	**0010**	0000	d_1
S_2	**0010**	0010	**0011**	0001	d_2
S_3	**0011**	0011	**0100**	0001	d_3
S_4	**0100**	0100	**0101**	0101	d_4
S_5	**0101**	0101	**0110**	**1001**	d_5
S_6	**0110**	0110	**0111**	0100	d_6
S_7	**0111**	0111	**1000**	0111	d_7
S_8	**1000**	**1000**	0000	0010	d_8

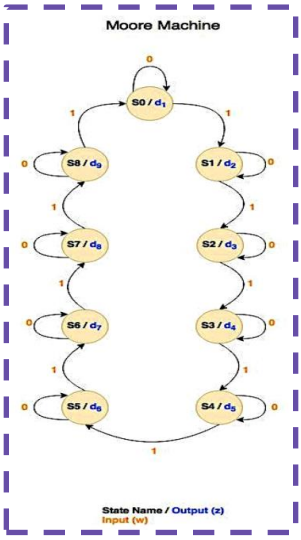


Figure 1.20 & Figure 1.21: State assigned table and state diagram of the Moore machine implementation of the FSM. S0-S8 represents the state binary values which are sent to the decoder, and D1-D9 represents the student ID values 501159472, which are displayed on the sseg.

Reset	Data-In	Clock	Student-ID	Current-State	State Number
0	**1**	**1**	0101	0000	**0**
0	**1**	**1**	0000	0001	**1**
0	**1**	**1**	0001	0010	**2**
0	**1**	**1**	0001	0011	**3**
0	**1**	**1**	0101	0100	**4**
0	**1**	**1**	**1001**	0101	**5**
0	**1**	**1**	0100	0110	**6**
0	**1**	**1**	0111	0111	**7**
0	**1**	**1**	0010	**1000**	**8**
1	**X**	**X**	0101	0000	**0**
0	**0**	**1**	**ID (from Prev State)**	**CS (from Prev State)**	**SN (from Prev State)**

Figure 1.22: Truth table for the Moore FSM, where positive edge triggered from the Clock allow in the transition of states when Reset = '0' and Data-In = '1'. X is used as a don't care variable, as the value of Clock or Data-In does not change the outcome of Reset = '0'.

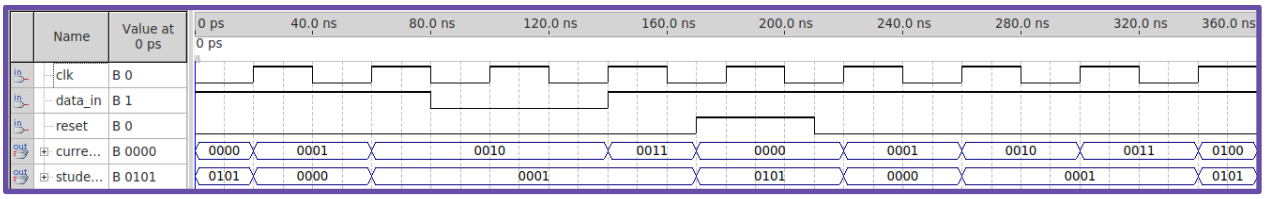


Figure 1.23: Waveform of the Moore FSM. Variance has been added to the waveform to add different cases.

As per the truth table of the Moore FSM in *Figure 1.6*, the outcome of the waveform is as expected. The first two positive edge **`Clock`** cycles change **`current_state`** along with its corresponding **`student_id`** since **`data_in`** is high (1) and **`reset`** is low (0). However, once the third **`Clock`** cycle is reached, **`current_state`** and its corresponding **`student_id`** remains unchanged. This is because **`data_in`** is low, preventing any new data to be taken in. Another case experimented upon was at the fifth **`Clock`** cycle, where although a positive edge of the **`Clock`** is reached and **`data_in`** is high, the **`current_state`** resets back to its initial state, along with its corresponding **`student_id`**. This waveform is the expected outcome for the Moore FSM.

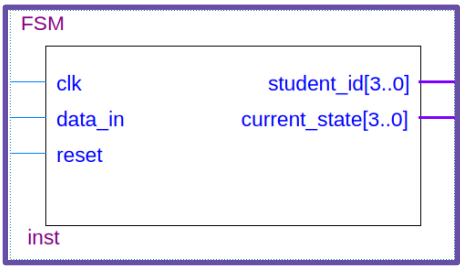


Figure 1.24: Block schematic of the Moore FSM.

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3
4   ENTITY FSM IS
5       PORT
6       (
7           clk : IN std_logic;
8           data_in : IN std_logic;
9           reset : IN std_logic;
10          student_id : OUT std_logic_vector(3 DOWNTO 0);
11          current_state : OUT std_logic_vector(3 DOWNTO 0)
12      );
13  END ENTITY;
14
15  ARCHITECTURE fsm OF FSM IS
16      TYPE state_type IS (s0, s1, s2, s3, s4, s5, s6, s7, s8);
```

\

Figure 1.25: VHDL Code of the Moore FSM.

1.3 - 4:16 Decoder

The final component used for the ALU's control unit is a 4:16 decoder, crafted using two 3:8 decoders as shown in *Figure 1.32*. The 4:16 decoder is used in the control unit of the ALU core, to interpret signals from the FSM, converting them into microcode that directs the operation of the ALU. This process involves translating the current state of the FSM into specific operation codes, which the ALU uses to select and perform the required functions. The "OP-Reversed" column in *Figure 1.30* is shown due to the microcode table being set in the opposite direction, thus the VHDL code and waveform reflects this as well for visual clarity as both act the same.

En	S (4-Bit input)	OP (16-Bit output)	OP-Reversed
0	X	0000000000000000	0000000000000000
1	0000	1000000000000000	0000000000000001
1	0001	0100000000000000	0000000000000010
1	0010	0010000000000000	0000000000000100
1	0011	0001000000000000	0000000000001000
1	0100	0000100000000000	0000000000010000
1	0101	0000010000000000	0000000000100000
1	0110	0000001000000000	0000000001000000
1	0111	0000000100000000	0000000010000000
1	1000	0000000010000000	0000000100000000
1	1001	0000000001000000	0000001000000000
1	1010	0000000000100000	0000010000000000
1	1011	0000000000010000	0000100000000000
1	1100	0000000000001000	0001000000000000
1	1101	0000000000000100	0010000000000000
1	1110	0000000000000010	0100000000000000
1	1111	0000000000000001	1000000000000000

Figure 1.30: Truth Table of the 4:16 Decoder. X is used as a "don't care" variable, as the value of the input does not change the outcome of En = '0'. There are extra outputs not utilized in the microtable since there are 9 functions, anything else from these decoder microcode outputs are not assigned to anything, hence they will not be used.

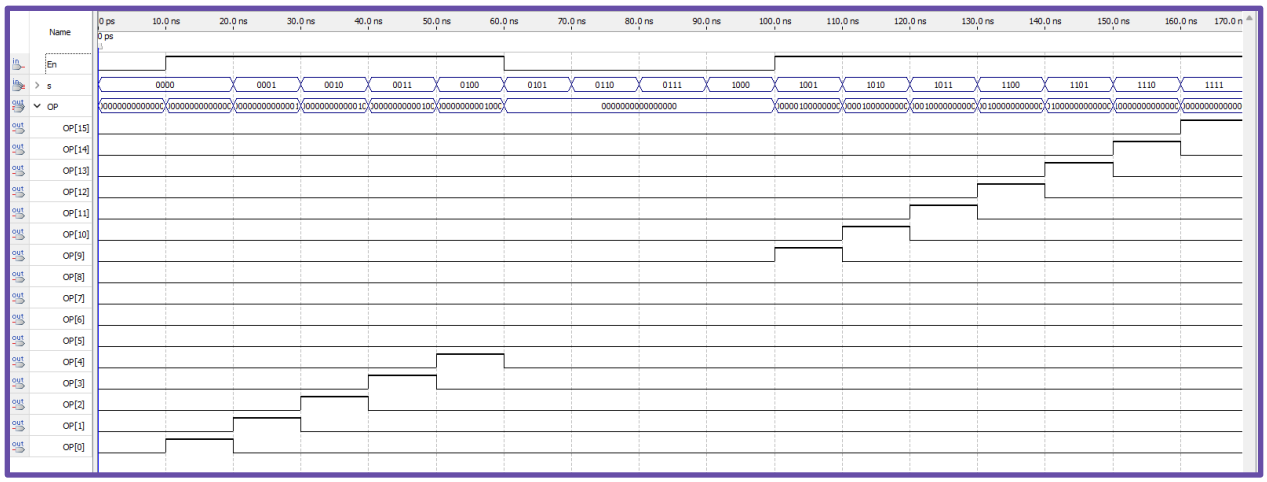


Figure 1.31: Waveform of the 4:16 decoder. Variance has been added to the waveform to add different cases.

As per the truth table of the 4:16 decoder in *Figure 1.9*, the outcome of the waveform is as expected. When **`En`** is set to high (1), the decoder is active, and the outputs respond to the input signals. When the **`En`** is low (0), all outputs are low, indicating that the decoder is disabled (an example seen from 60 ns to 100 ns). Furthermore, the input signal, **`S`**, changes over time. It can be seen that for any given binary input, only one **`OP`** line is high at a time, corresponding to the binary value of **`S`**, which is shown in the waveform.

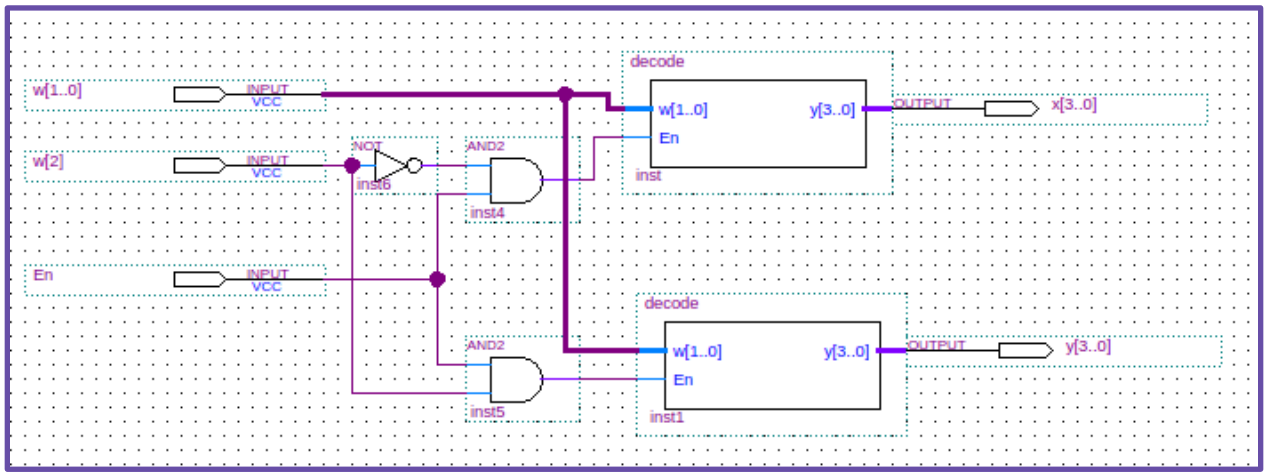


Figure 1.32: Block Diagram of the 4:16 decoder component, created using logic gates and two 3:8 decoders.

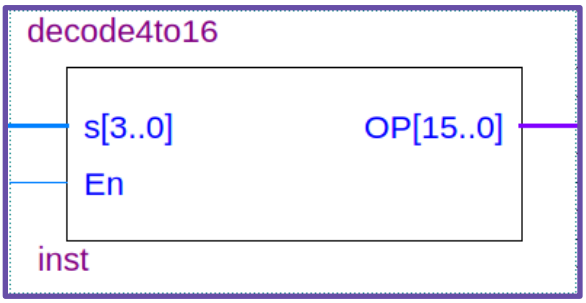


Figure 1.33: Block Schematic of the 4:16 decoder component, used in the final design of the GPU.

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3
4   ENTITY decode IS
5       PORT ( w : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
6              En : IN STD_LOGIC;
7              y : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
8            );
9   END decode;
10
11  ARCHITECTURE Behavior OF decode IS
12      SIGNAL Enw : STD_LOGIC_VECTOR(3 DOWNTO 0);
13  BEGIN
14      Enw <= En & w;
15
16      WITH Enw SELECT
17          y <= "10000000" WHEN "1000", -- Corresponding to input 000
18               "01000000" WHEN "1001", -- Corresponding to input 001
19               "00100000" WHEN "1010", -- Corresponding to input 010
20               "00010000" WHEN "1011", -- Corresponding to input 011
21               "00001000" WHEN "1100", -- Corresponding to input 100
22               "00000100" WHEN "1101", -- Corresponding to input 101
23               "00000010" WHEN "1110", -- Corresponding to input 110
24               "00000001" WHEN "1111", -- Corresponding to input 111
25               "00000000" WHEN OTHERS; -- Default case
26  END Behavior;
27
```

Figure 1.34: VHDL Code of the 4:16 decoder used for the final block diagram of the GPU to allow the block of Figure 1.33 to function correctly.

Chapter 2 - ALU Problem Set #1

The Arithmetic and Logic Unit (ALU), a central component of the general-purpose processor designed in this lab, is tasked with performing a variety of arithmetic and logical operations. These operations are done using its respective input and output groups.

The input group of the ALU consists of four main components:

- *Clk*
- *OP*
- *A* and *B* (8-bit inputs)

First, the *Clk* signal is used to determine when the ALU will read the instructions from the 4:16 decoder. When the clock signal hits a rising edge, the ALU will read the input task specified from the decoder, which is represented in a 16-bit binary value. This 16-bit binary value is called *OP,* and tells the ALU core what arithmetic operation to do based on its microcode value. For example, in this problem set, if *OP = 0000000000000001*, the ALU core will perform addition on the two input values, *A* and *B. A and B* inputs hold two 8-bit binary values, which are stored by the two latches. These inputs are used in tandem with one another to perform the specified arithmetic operations determined from the ALU core.

Finally, output group of the ALU consists of two main components:

- *Neg*
- *R1* and *R2*

First, *Neg* is used to determine whether or not the output contains a negative result, which is then displayed using the connected Seven Segment Display (sseg). Finally, *R1* and *R2* consist of two 4-bit binary values which represent the two halves of the 8-bit result from the ALU operations. *R1* shows the first four bits, and *R2* the second four bits, which together, display the hexadecimal representation of the operation's binary result on the sseg.

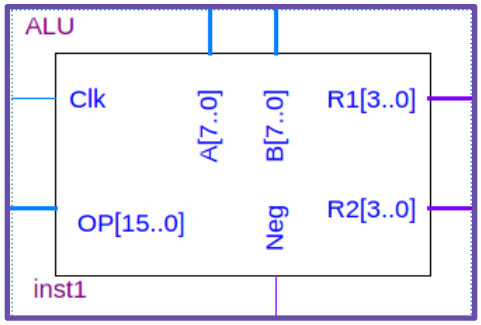


Figure 2.10: Block Diagram of the ALU.

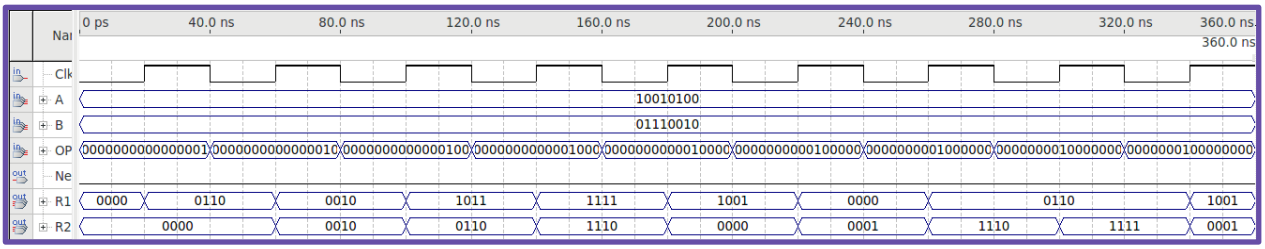


Figure 2.11: Waveform of the ALU, with A = 10010100, and B = 01110010.

It can be seen that the outputs of the ALU result as expected. When a positive edge trigger of the **`Clock`** occurs, the two resultant combination of, **`R1`** and **`R2`** change based on the micro table operations in *Figure 2.11*, as the microcode from the 4:16 decoder is sent to the ALU, and depending on its code, the ALU will do a certain function. For example, in the first clock cycle, the microcode sent from the decoder to the ALU is *"0000000000000001"*, which corresponds to the function of the summation between **`A`** and **`B`**. From simple binary addition, it can be seen that A + B does in fact equal to *"0000 0110"*, and since **`R1`** represents the first four *"0110"* and **`R2`** represents the last four *"0000"*, the output is what is expected since the concatenation between **`R2`** and **`R1`** is equal to *"0000 0110"*. This cycle is then repeated for each subsequent function before restarting back to function #1.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY ALU IS
    PORT (
        Clk : IN STD_LOGIC;
        A, B : IN UNSIGNED(7 DOWNTO 0);
        OP : IN UNSIGNED(15 DOWNTO 0);
        Neg : OUT STD_LOGIC;
        R1, R2 : OUT UNSIGNED(3 DOWNTO 0)
    );
END ALU;

ARCHITECTURE Calculation OF ALU 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
                WHEN "0000000000000001" =>
                    Result <= Reg1 + Reg2;
                    Neg <= '0';
                WHEN "0000000000000010" =>
                    IF Reg1 > Reg2 THEN
                        Result <= Reg1 - Reg2;
                        Neg <= '0';
                    ELSIF Reg2 > Reg1 THEN
                        Result <= Reg1 + (NOT Reg2 + 1);
                        Neg <= '1';
                    END IF;
                WHEN "0000000000000100" =>
                    Result <= NOT Reg1;
                    Neg <= '0';
                WHEN "0000000000001000" =>
                    Result <= Reg1 NAND Reg2;
                    Neg <= '0';
                WHEN "0000000000010000" =>
                    Result <= Reg1 NOR Reg2;
                    Neg <= '0';
                WHEN "0000000000100000" =>
                    Result <= Reg1 AND Reg2;
                    Neg <= '0';
                WHEN "0000000001000000" =>
                    Result <= Reg1 XOR Reg2;
                    Neg <= '0';
                WHEN "0000000010000000" =>
                    Result <= Reg1 OR Reg2;
                    Neg <= '0';
                WHEN "0000000100000000" =>
                    Result <= Reg1 XNOR Reg2;
                    Neg <= '0';
                WHEN OTHERS =>
                    -- Don't care, do nothing
            END CASE;
        END IF;
    END PROCESS;

    R1 <= Result(3 DOWNTO 0);
    R2 <= Result(7 DOWNTO 4);
END Calculation;

```

Figure 2.12: ALU VHDL code.

2.1: General Processing Unit (ALU_1)

Once all the components of the ALU core are built together, along with the Seven Segment Display, it becomes a unit as shown in *Figure 2.12*, known as the General Purpose Processor (GPU). The table of microcode's generated by decoder for the ALU along with its waveform using FSM's student ID are shown as follows:

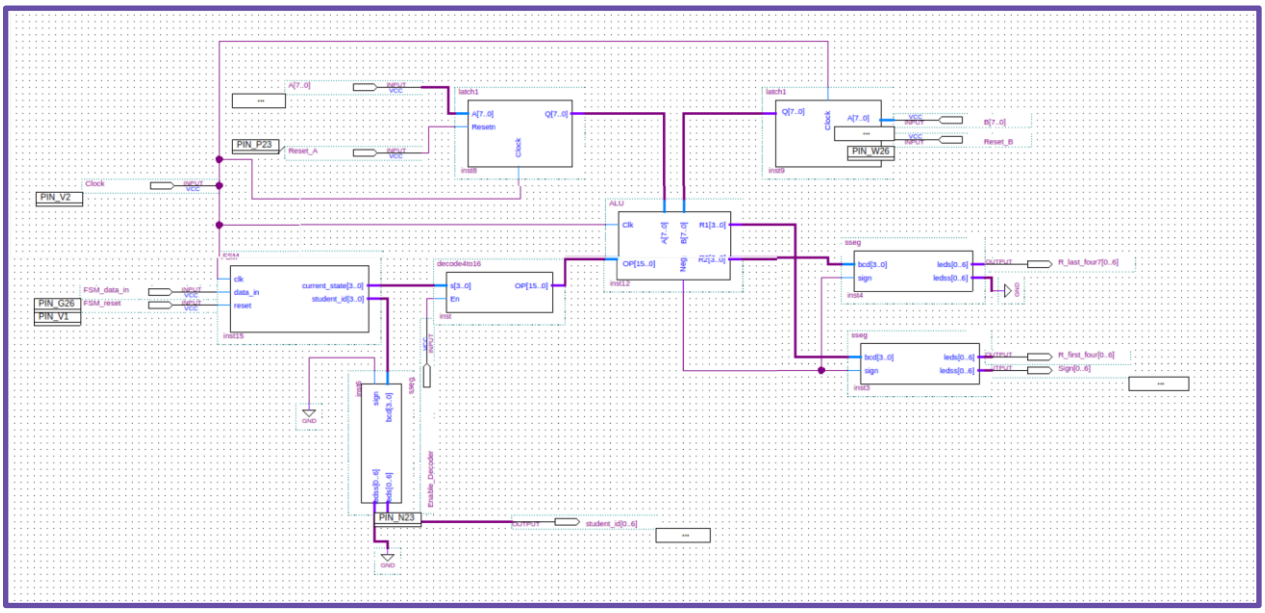


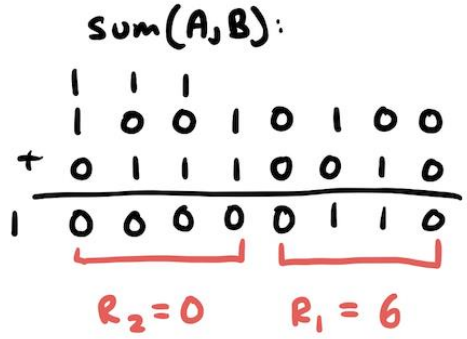
Figure 2.13: Block Diagram of the completed GPU, denoted as ALU_1.

Function #	OP (Microcode)	Boolean Function/Operation
1	0000000000000001	$sum(A, B)$
2	0000000000000010	$diff(A, B)$
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	$\overline{A \oplus B}$

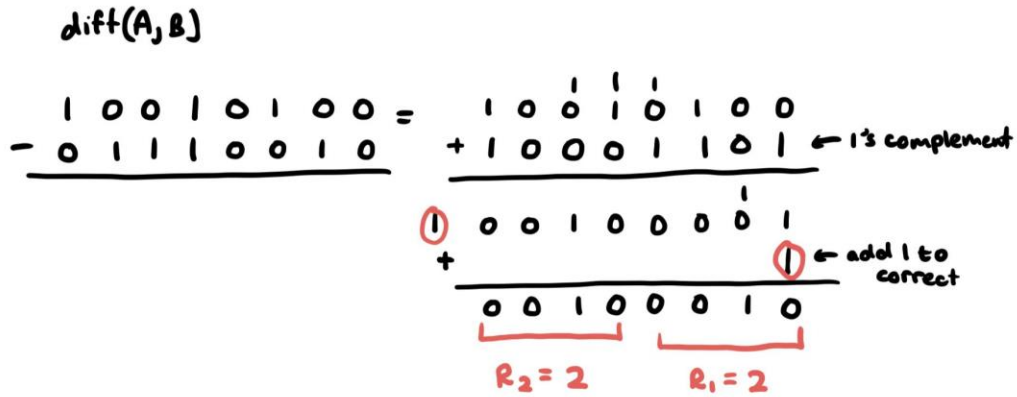
Figure 2.14: ALU core operations for problem set 1, determined from the 4:16 decoder's OP signal.

To ensure that the waveform outputs as expected, the following calculations were done for each individual function in the microtable (**R2** and **R1** resultants is what is displayed on the SSEG Display):

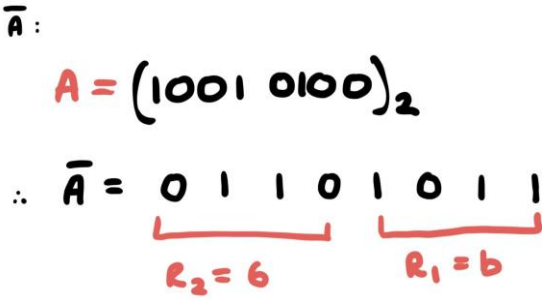
1. Function #1 - sum(A,B):



2. Function #2 - diff(A,B):



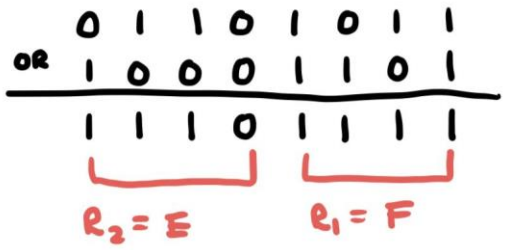
3. Function #3 - NOT(A):



4. Function #4 - NOT(A AND B)

$\overline{A \cdot B}$:

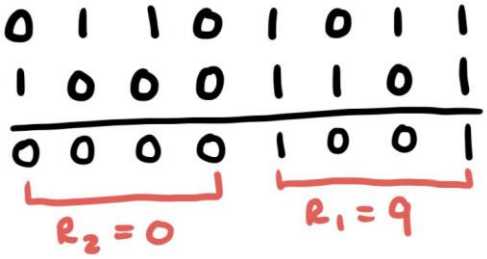
$\overline{A \cdot B} = \bar{A} + \bar{B} \Rightarrow$



5. Function #5 - NOT(A OR B)

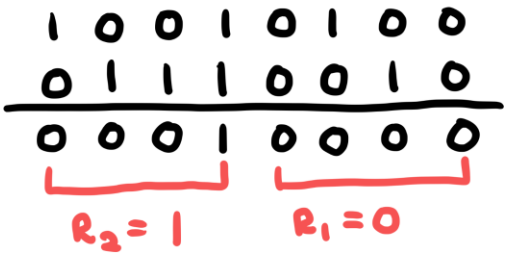
$\overline{A + B}$:

$\overline{A + B} = \bar{A} \cdot \bar{B} \Rightarrow$

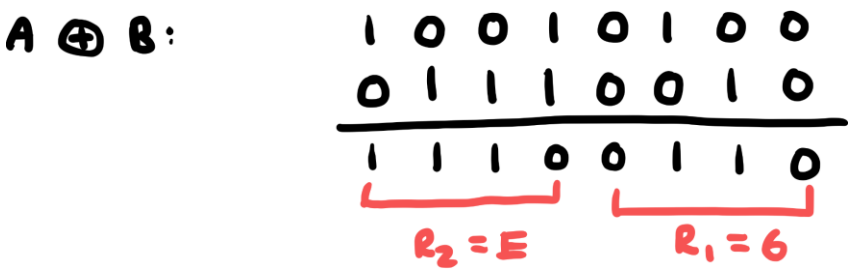


6. Function #6 - A AND B

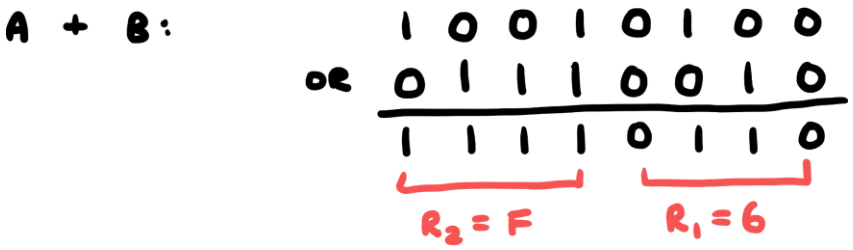
$A \cdot B$



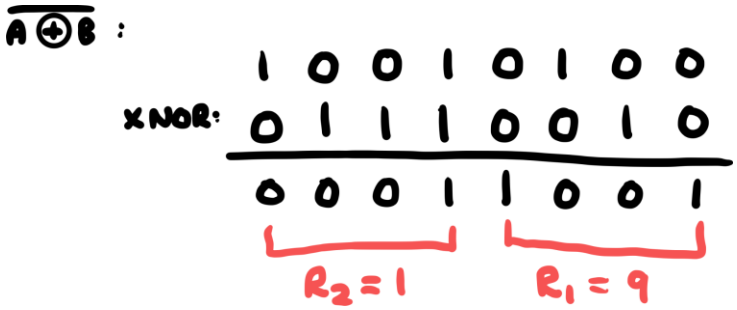
7. Function #7 - A XOR B



8. Function #8 - A OR B



9. Function #9 - A XNOR B



These calculations will be further used in the verification of the waveform in *Figure 2.15* but the outputs of the SSEG Display will be outputted instead of the raw values of **`R2`** and **`R1`**, but the conversion is merely hexadecimal, hence they are both equivalent. The values calculated match with the ALU waveform, so the waveform is properly verified, thus the first GPU is then proceeded.

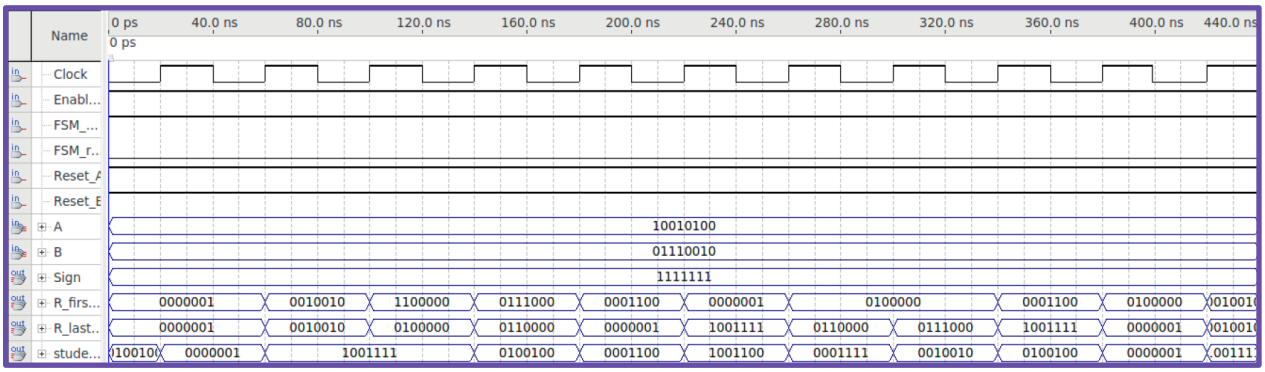


Figure 2.15: ALU_1 (General Processing Unit) Waveform.

It can be seen that the outputs of the GPU result as expected, noting the fact that the two initial results of **`R1 (R_lastfour)`** and **`R2 (R_firstfour)`** both display "0" on the SSEG Display. The first output being "0" is shown since **`A`** and **`B`** are not set in the previous state, thus they are essentially "don't care" values and the SSEG will display 0 since these two values are not defined yet. Once the first edge of the clock occurs, the latch components which store **`A`** and **`B`** load both of these values in, but read from the previous state which are also "don't care" values, thus showing "0" on the SSEG Display as well. Once the first clock cycle is complete, the next positive edge in the **`Clock`** cycle will present the first function in the ALU, as depending on the initial state of the FSM, the value fed into the decoder results in a specific microcode value that results in the specific operation as shown in *Figure 2.14*. Since the second clock cycle now feeds both **`A`** and **`B`**, this causes a delay between the FSM and the ALU unit by one clock cycle due to the aforementioned occurrence. Once the first round of outputs are complete, the first student ID value of "5" will now be displayed alongside the *XNOR* function since the ALU is obtaining the microcode values from the previous state. This is shown in the last clock cycle of the ALU, as the last positive edge now displays "5" (the first state of the FSM which contains the **`student_id`** of "5") with the *XNOR* resultant being "19" (in the format of **`R2R1`**) verified with the previous calculations. This means that the second state will now be the function associated with *sum(A,B)*, and the rest of the functions are shifted as well.

This means that past the first state $\square_0$ which as mentioned before displays the last function, *XNOR* between **`A`** and **`B`**, the **`student_id`** that is associated with $\square_1$, which is "0", now corresponds with the first ALU function of *sum(A,B)*. The two resultants, ***`R1`*** and ***`R2`***, will output the sum of **`A`** and **`B`** but split into 4 bits each, which is then taken in by the SSEG to then associate the bit value that corresponds with it. As calculated previously, ***`R2`*** = *0000* and ***`R1`*** = *0110*. Thus, on the waveform, ***`R2`*** = *0000001* and ***`R1`*** = *0100000*, which will display *"06"* (***`R2R1`***) on the SSEG Display. This repeats for the rest of the functions, corresponding to the microtable of *Figure 2.14* per rising **`Clock`** edge, assuming that the **`FSM_datain`**, **`Enable`**, and **`Reset_A`** and **`Reset_B`** are high (1).

Chapter 3 - ALU Problem Set #2: Function H

The second set of problem sets for the ALU was to modify the preexisting microcode table to now do following:

- ➔ Rotate A to right by 4 bits (ROR)
- ➔ Produce the result of XORing A and B
- ➔ Invert the bit-significance order of B
- ➔ Calculate the summation of A and B and decrease it by 2
- ➔ Rotate B to left by 2 bits (ROL)
- ➔ Invert the even bits of B
- ➔ Swap the lower 4 bits of B with lower 4 bits of A
- ➔ Shift B to right 2 bits, input bit = 0 (SHR)
- ➔ Invert lower four bits of A

Since only the ALU's functionality regarding the operations have changed, the inputs/outputs groups compared to the first problem set are identical. For the outcome to be achieved, it contains a number of input and output groups.

The input group of the ALU consists of four main components:

- ➢ *Clk*
- ➢ *OP*
- ➢ *A* and *B* (8-bit inputs)

First, the *Clk* signal is used to determine when the ALU will read the instructions from the 4:16 decoder. When the clock signal hits a rising edge, the ALU will read the input task specified from the decoder, which is represented in a 16-bit binary value. This 16-bit binary value is called *OP,* and tells the ALU core what arithmetic operation to do based on its microcode value. For example, in this problem set, if *OP* = 0000000000000001, the ALU core will perform addition on the two input values, *A* and *B. A and B* inputs hold two 8-bit binary values, which are stored by the two latches. These inputs are used in tandem with one another to perform the specified arithmetic operations determined from the ALU core.

Finally, output group of the ALU consists of two main components:

- ➢ *Neg*
- ➢ *R1* and *R2*

First, *Neg* is used to determine whether or not the output contains a negative result, which is then displayed using the connected Seven Segment Display (sseg). Finally, *R1* and *R2* consist of two 4-bit binary values which represent the two halves of the 8-bit result from the ALU operations. *R1* shows the first four bits, and *R2* the second four bits, which together, display the hexadecimal

representation of the operation's binary result on the sseg.

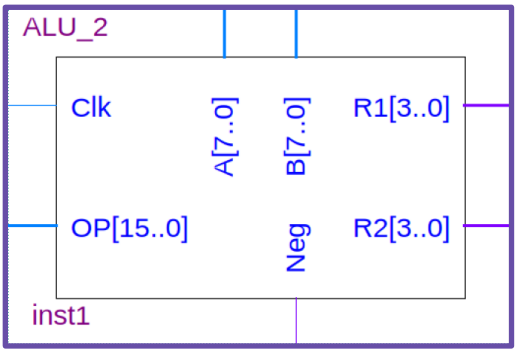


Figure 3.10: Block diagram of ALU_2.

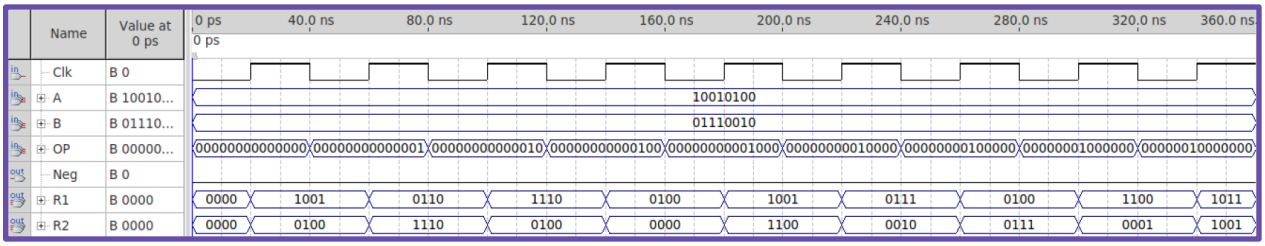


Figure 3.11: Generated waveform of ALU_2.

It can be seen that the outputs of the ALU result as expected. When a positive edge trigger of the **`Clk`** occurs, the two resultant combination of, **`R1`** and **`R2`** change based on the micro table operations in *Figure 3.13*, as the microcode from the 4:16 decoder is sent to the ALU, and depending on its code, the ALU will do a certain function. For example, in the first **`Clk`** cycle, the microcode sent from the decoder to the ALU is "*0000000000000001*", which corresponds to the function of the rotation of **`A`** to the right by 4 bits . To check if the waveform is indeed correct, rotating A by 4 bits can be done from basic boolean manipulation. Since **`A`** = *1001 0100,* to rotate **`A`** to the right by 4 bits, two halves, X and Y, are rotated clockwise since it is to the right. Let X = 1001 and Y = 0100. Thus, **`A`** = YX = 0100 1001 which is displayed on the waveform correctly as **`R1`** corresponds to *1001* and **`R2`** corresponds to *0100* as shown in the waveform of *Figure 3.11.*

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY ALU_2 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 ALU_2;

ARCHITECTURE calculation OF ALU_2 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
                WHEN "0000000000000001" =>
                    Result <= Reg1 ROR 4; --1
                WHEN "0000000000000010" =>
                    Result <= Reg1 XOR Reg2; --2
                WHEN "0000000000000100" =>
                    FOR i IN 0 TO 7 LOOP --3
                        Result(i) <= Reg2(7 - i);
                    END LOOP;
                WHEN "0000000000001000" =>
                    Result <= Reg1 + Reg2 - "00000010"; --4 Calculate the summation of A and B and decrease it by 2
                WHEN "0000000000010000" =>
                    Result <= Reg2 ROL 2; --5
                WHEN "0000000000100000" =>
                    Result <= Reg2 XOR "01010101";
                WHEN "0000000001000000" =>
                    Result(3 DOWNTO 0) <= Reg1(3 DOWNTO 0); --7 Swap the lower 4 bits of B with lower 4 bits of A
                    Result(7 DOWNTO 4) <= Reg2(7 DOWNTO 4);
                WHEN "0000000010000000" =>
                    Result <= Reg2 SRL 2; --8 Shift B to right by 2 bits, input bit = 0 (SHR)
                WHEN "0000000100000000" => '
                    Result(3 DOWNTO 0) <= NOT Reg1(3 DOWNTO 0); --9 Invert lower four bits of A
                    Result(7 DOWNTO 4) <= Reg1(7 DOWNTO 4);
                WHEN OTHERS =>
                    Result <= (OTHERS => '0');
            END CASE;
        END IF;
    END PROCESS;

    R1 <= Result(3 DOWNTO 0);
    R2 <= Result(7 DOWNTO 4);
END calculation;
```

Figure 3.12: VHDL code for ALU_2.

3.1: General Processing Unit (ALU_2)

Once all the components of the ALU_2 core are built together, along with the Seven Segment Display, it becomes a unit as shown in *Figure 3.14*, known as the General Purpose Processor (GPU). The overall look of the GPU is the same as *Figure 2.13*, but the overall functionality is different due to the different functions the ALU now performs. The table of microcode's generated by decoder for the ALU along with its waveform using FSM's student ID are shown as follows:

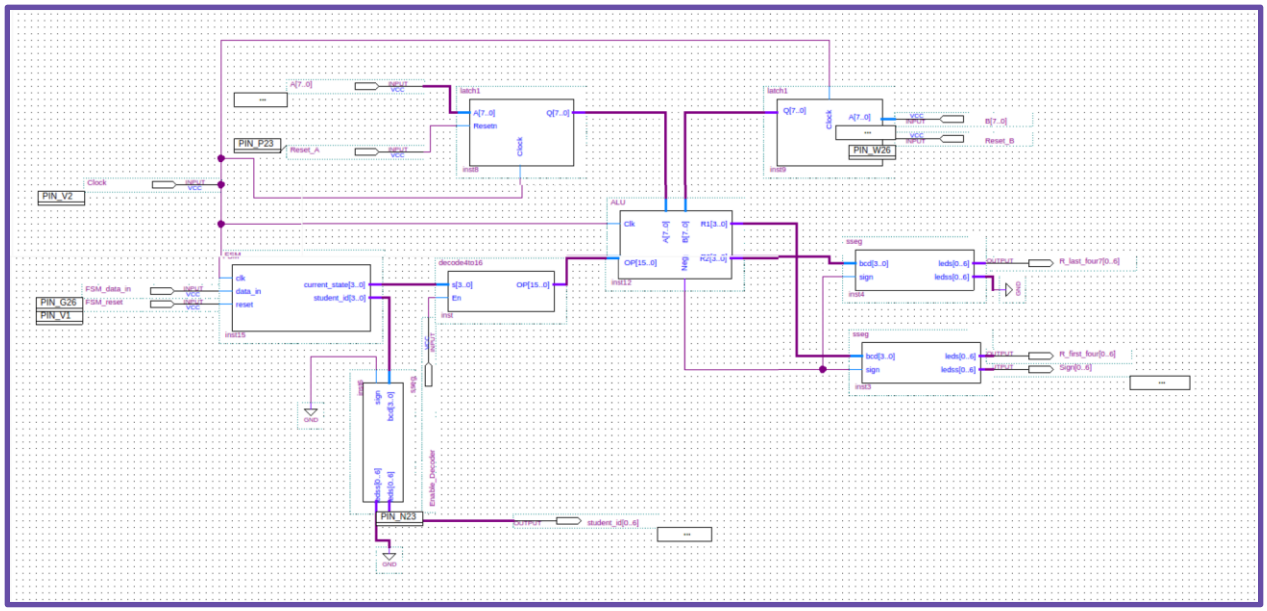


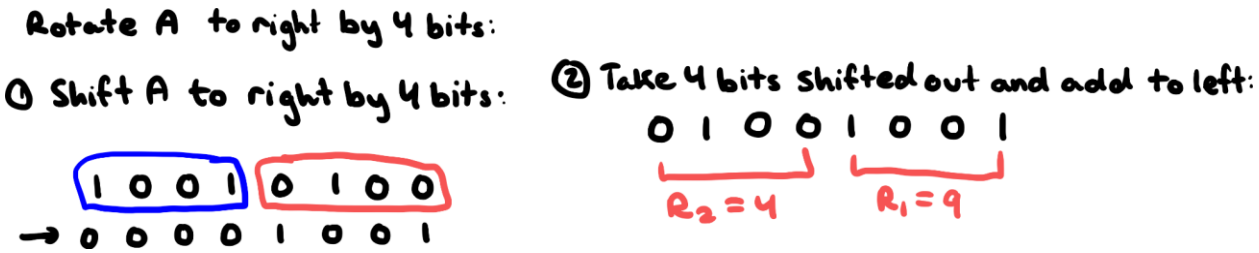
Figure 3.14: Block Diagram for ALU_2 - General Processing Unit.

Function #	OP (Microcode)	Boolean Operation /Function
1	000000000000000**1**	Rotate **A** to right by 4 bits (ROR)
2	000000000000000**1**0	Produce the result of XORing **A** and **B**
3	0000000000000**1**00	Invert the bit-significance order of **B**
4	0000000000000**1**000	Calculate the summation of **A** and **B** and decrease it by 2
5	0000000000**1**0000	Rotate **B** to left by 2 bits (ROL)
6	0000000000**1**00000	Invert the even bits of **B**
7	0000000000**1**000000	Swap the lower 4 bits of **B** with lower 4 bits of **A**
8	0000000**1**0000000	Shift **B** to right 2 bits, input bit = 0 (SHR)
9	0000000**1**00000000	Invert lower four bits of **A**

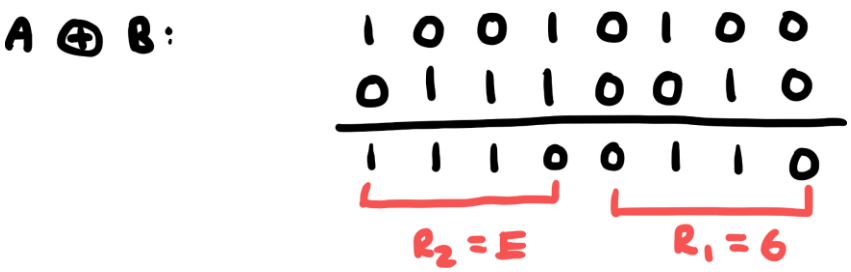
Figure 3.15: ALU core operations for problem set 2, determined from the 4:16 decoder's OP signal.

To ensure that the waveform outputs as expected, the following calculations were done for each individual function in the microtable (`R2` and `R1` resultants is what is displayed on the SSEG Display):

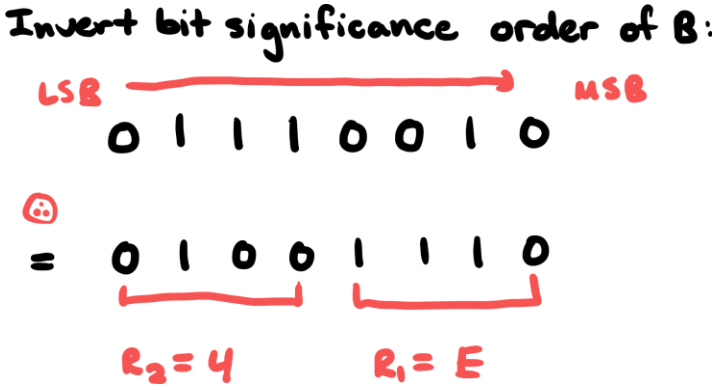
1. Rotate A to right by 4 bits (ROR)



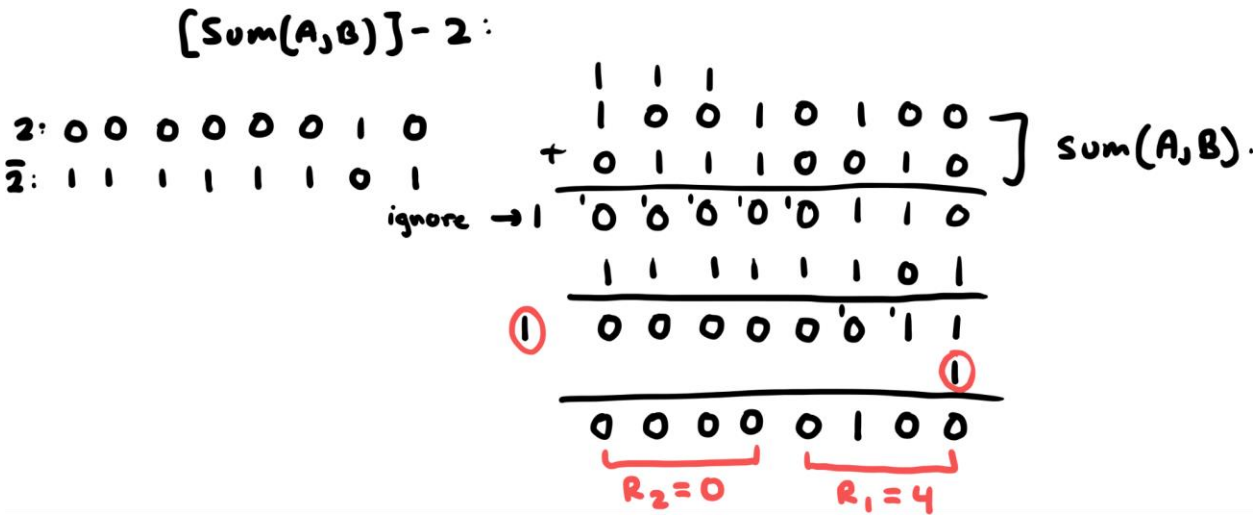
2. Produce the result of XORing A and B



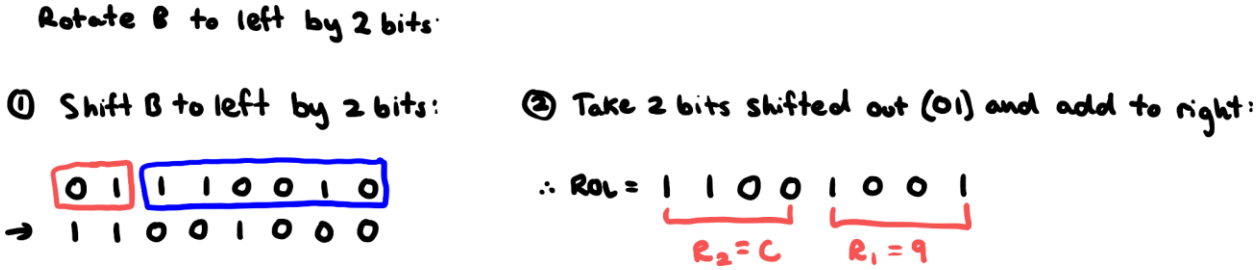
3. Invert the bit-significance order of B



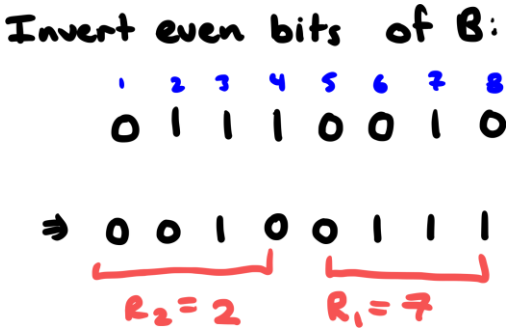
4. Calculate the summation of A and B and decrease it by 2



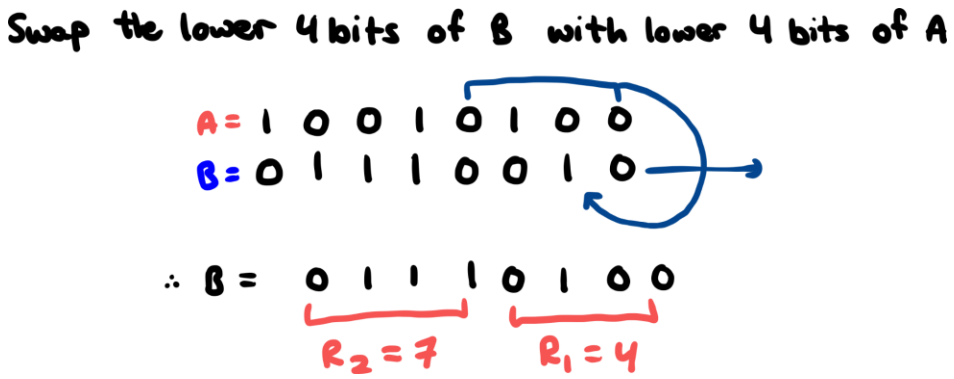
5. Rotate B to left by 2 bits (ROL)



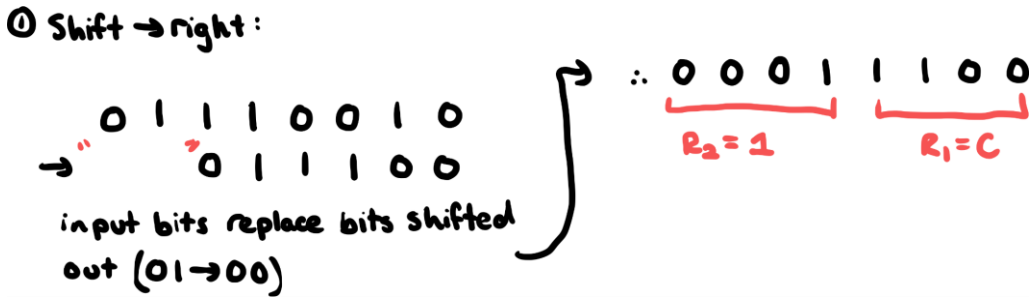
6. Invert the even bits of B



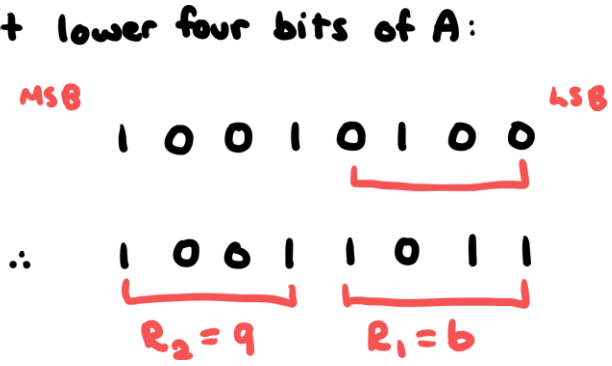
7. Swap the lower 4 bits of B with lower 4 bits of A



8. Shift B to right 2 bits, input bit = 0 (SHR)



9. Invert lower four bits of A



These calculations will be further used in the verification of the waveform in *Figure 3.16* but the outputs of the SSEG Display will be outputted instead of the raw values of **R2** and **R1**, but the conversion

is merely hexadecimal, hence they are both equivalent. The values calculated match with the ALU waveform, so the waveform is properly verified, thus the new GPU's waveform can be made.

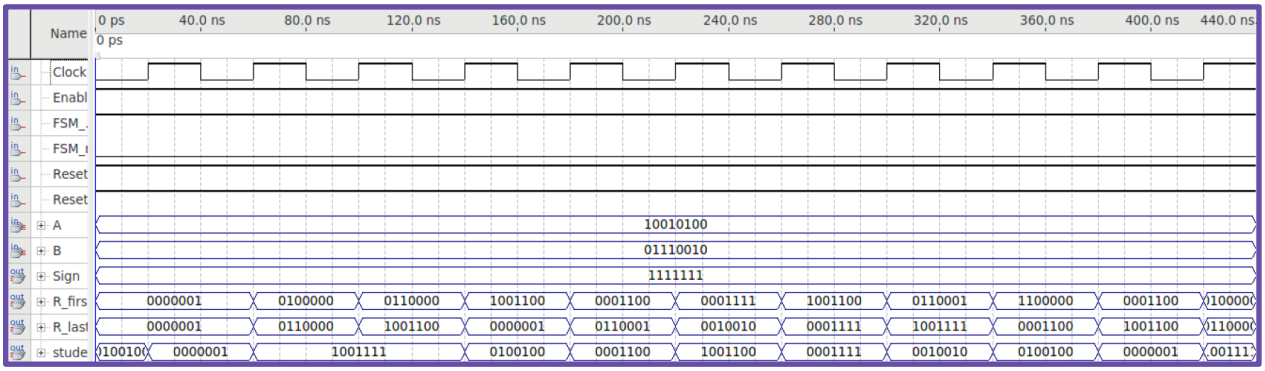


Figure 3.16: Waveform of the Block Diagram for ALU_2 - General Processing Unit.

It can be seen that the outputs of the GPU result as expected, noting the fact that the two initial results of **`R1 (R_lastfour)`** and **`R2 (R_firstfour)`** both display "0" on the SSEG Display. The first output being "0" is shown since **`A`** and **`B`** are not set in the previous state, thus they are essentially "don't care" values and the SSEG will display "00" since these two values are not defined yet. Once the first edge of the clock occurs, the latch components which store **`A`** and **`B`** load both of these values in, but read from the previous state which are also "don't care" values, thus showing "0" on the SSEG Display as well. Once the first clock cycle is complete, the next positive edge in the **`Clock`** cycle will present the first function in the ALU, as depending on the initial state of the FSM, the value fed into the decoder results in a specific microcode value that results in the specific operation as shown in *Figure 3.15*. Since the second clock cycle now feeds both **`A`** and **`B`**, this causes a delay between the FSM and the ALU unit by one clock cycle due to the aforementioned occurrence. Once the first round of outputs are complete, the first **`student_id`** value of "5" will now be displayed alongside the "Invert lower four bits of A" function since the ALU is obtaining the microcode values from the previous state. This is shown in the last clock cycle of the ALU, as the last positive edge now displays "5" (the first state of the FSM which contains the **`student_id`** of "5") with the "Invert lower four bits of A" resultant being "9b" (***`R2R1`***). This means that the second state will now be the function associated with *"Rotate A to right by 4 bits (ROR)*", and the rest of the functions are shifted as well.

This means that past the first state, $□_0$, which as mentioned before displays the "*Invert lower four bits of A*" function, the **`student_id`** that is associated with $□_1$, which is "0", now corresponds with the first ALU function of "Rotate A to right by 4 bits (ROR)". The two resultants, ***`R1`*** and ***`R2`***, will output the sum of **`A`** and **`B`** but split into 4 bits each, which is then taken in by the SSEG to then associate the bit value that corresponds with it. As calculated previously, ***`R1`*** = *1001* and **`R2`** = *0100*. Thus, on the waveform, **`R1`** = *0001100* and **`R2`** = *1001100*, which will display *"49"* (***`R2R1`***) on the SSEG Display. This repeats for the rest of the functions, corresponding to the microtable of *Figure 3.16* per rising **`Clock`** edge, assuming that the **`FSM_datain`**, **`Enable`**, and **`Reset_A`** and **`Reset_B`** are high(1).

Chapter 4 - ALU Problem Set #3

For the third and final problem set, the ALU was modified to handle the **`student_id`** output from the FSM sub-component of the Control Unit. This allowed the ALU to now handle four inputs as shown in *Figure 4.10*. Utilizing the **`student_id`** output, the task for this problem set is as follows:

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

Since the ALU was modified to take an additional input, the inputs/outputs groups compared to the first problem set are different in some aspects. These input and output groups are the following to achieve the desired results:

The input group of the ALU consists of four main components:

- *Clk*
- *OP*
- *A* and *B* (8-bit inputs)
- *student_id*

First, the *Clk* signal is used to determine when the ALU will read the instructions from the 4:16 decoder. When the clock signal hits a rising edge, the ALU will read the input task specified from the decoder, which is represented in a 16-bit binary value. This 16-bit binary value is called *OP,* and tells the ALU core what arithmetic operation to do based on its microcode value. For example, in this problem set, if *OP* = 0000000000000001, the ALU core will perform addition on the two input values, *A* and *B. A and B* inputs hold two 8-bit binary values, which are stored by the two latches. These inputs are used in tandem with one another to perform the specified arithmetic operations determined from the ALU core. Finally, the *student_id* is used to determine what is to be displayed on the SSEG Display based on whether or not one of the two digits of B are greater than the *student_id* output from the FSM.

Finally, output group of the ALU consists of two main components:

- *R1* (Result)

R1 is a 4-bit binary representation of the output that is displayed on the Seven Segment Display (sseg). *R1* will output either *(0001)* or *(0000)* to denote a true or false value for the state of the condition being checked. If the condition is true, then the SSEG Display will display a “y”. Conversely, if the condition is false, then the SSEG Display will display a “n”. This is depicted in *Figure 4.11*.

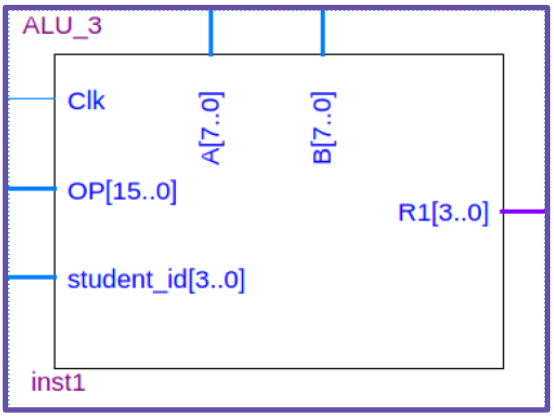


Figure 4.10: Block diagram of ALU_3.

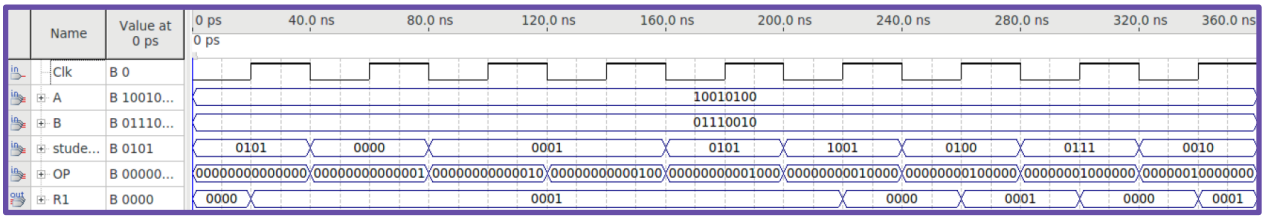


Figure 4.11: Generated waveform of ALU_3.

It can be seen that the outputs of the ALU result as expected. When a positive edge trigger of the **`Clk`** occurs, the ALU checks to see whether the two digits of B are greater than **`student_id`**. If it is, it displays "0001" indicating that it is bigger, else, it displays "0000" indicating that it is not bigger. This logic can be seen in the VHDL code of the ALU_3 in *Figure 4.12*. This repeats for each digit of the **`student_id`** since the OP microcode determines which **`student_id`** is next up to compare the **`B`** digits with. For example, the first **`Clock`** edge has a **`student_id`** of 5. The **`B`** digit in Hexadecimal is 72, where digit 1 is *"7"* and digit 2 is *"2"*. Since *"7"* is greater than *"2"*, this means that the result, **`R2,`** will indicate that it is bigger than **`student_id`**, thus denoting a *"0001"*. It should be noted that A is not used for this part since the logic operations are only dependent on B's two digit values, 7 and 2 respectively. A is merely shown for visuals.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY ALU_3 IS
    PORT(
        Clk: IN STD_LOGIC;
        A, B: IN UNSIGNED(7 DOWNTO 0);
        student_id: IN UNSIGNED(3 DOWNTO 0);
        OP: IN UNSIGNED(15 DOWNTO 0);
        R1: OUT UNSIGNED(3 DOWNTO 0)
    );
END ALU_3;

ARCHITECTURE Calculation OF ALU_3 IS
    SIGNAL Reg1, Result: UNSIGNED(7 DOWNTO 0) := (OTHERS => '0');
    SIGNAL Reg4, Reg5: UNSIGNED(0 TO 7);
BEGIN

    Reg1 <= B;
    Reg4 <= "0000" & B(7 DOWNTO 4); -- grab the high-order 4 bits (tens place digit in hex, 7)
    Reg5 <= "0000" & B(3 DOWNTO 0); -- grab the low-order 4 bits (ones place digit in hex, 2)

    PROCESS(Clk, OP)
    BEGIN
        IF (RISING_EDGE(Clk)) THEN
            CASE OP IS
                WHEN "0000000000000001" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000000000010" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000000000100" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000000001000" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000000010000" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000000100000" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000001000000" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000010000000" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN "0000000100000000" =>
                    IF (Reg4 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSIF (Reg5 > student_id) THEN
                        Result <= "00000001"; -- Y
                    ELSE
                        Result <= "00000000"; -- N
                    END IF;

                WHEN OTHERS =>
                    Result <= (OTHERS => '-'); -- Default case

            END CASE;
        END IF;
    END PROCESS;

    R1 <= Result(3 DOWNTO 0); -- grab only first 4 bits of the result since that is what is conv to SSEG
END Calculation;

```

Figure 4.12: VHDL code of ALU_3.

4.1: General Processing Unit (ALU_3)

Once all the components of the ALU core are built together, along with the Seven Segment Display and its modified version, it becomes a unit as shown in *Figure 2.12*, known as the General Purpose Processor (GPU) now modified to handle simple logic arithmetic. The table of microcode's generated by decoder for the ALU along with its waveform using FSM's student ID are shown as follows:

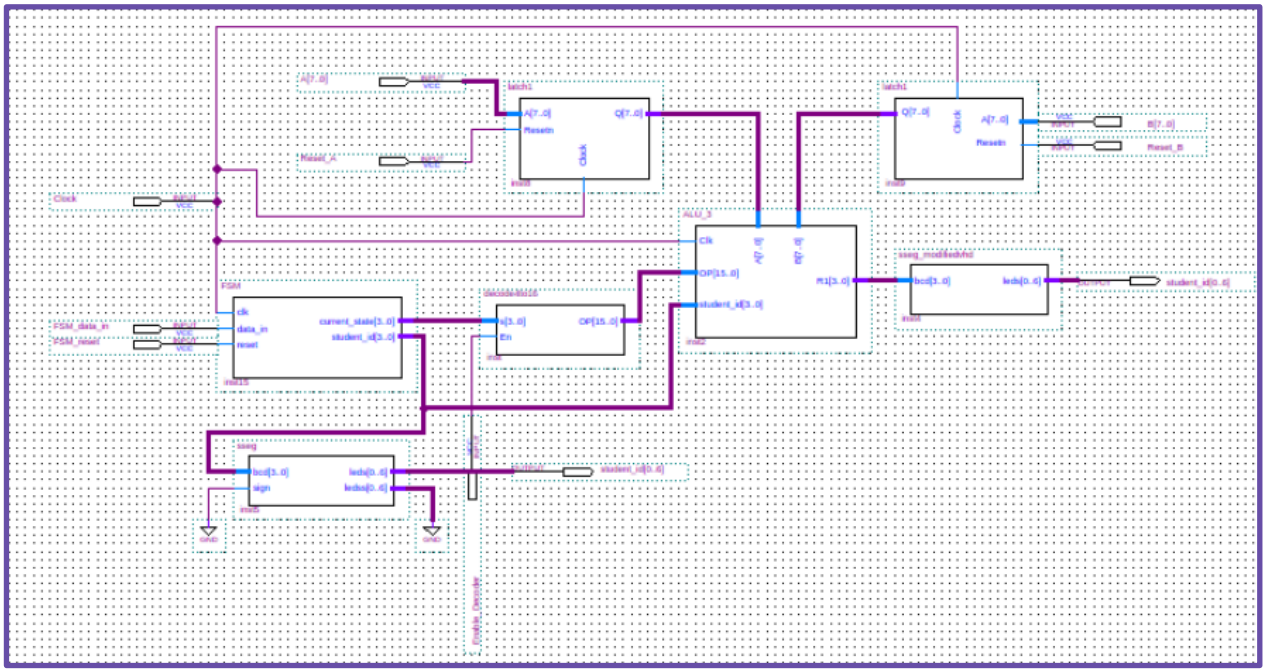
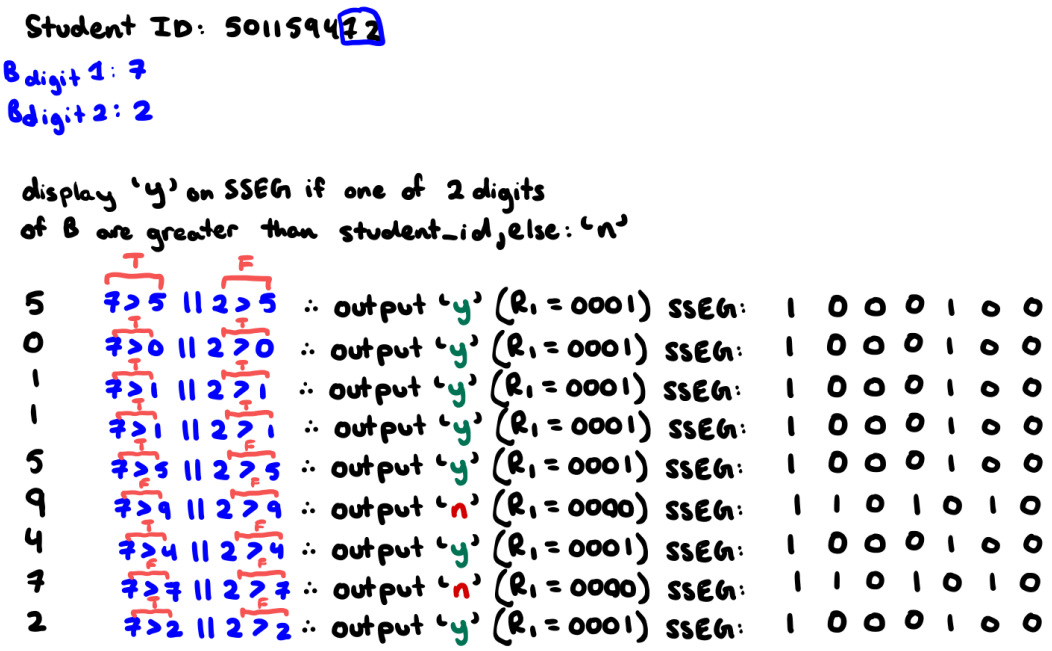


Figure 4.13: Block Diagram for ALU_3 General Processing Unit.

Function #	OP (Microcode)	Boolean Operation /Function
1	000000000000000**1**	Check if either Digit of B > 5: (D1>5)V(D2>5)
2	000000000000000**1**0	Check if either Digit of B > 0: (D1>0)V(D2>0)
3	0000000000000**1**00	Check if either Digit of B > 1: (D1>1)V(D2>1)
4	0000000000000**1**000	Check if either Digit of B > 1: (D1>1)V(D2>1)
5	0000000000**1**0000	Check if either Digit of B > 5: (D1>5)V(D2>5)
6	0000000000**1**00000	Check if either Digit of B > 9: (D1>9)V(D2>9)
7	0000000**1**1000000	Check if either Digit of B > 4: (D1>4)V(D2>4)
8	0000000**1**0000000	Check if either Digit of B > 7: (D1>7)V(D2>7)
9	0000000**1**00000000	Check if either Digit of B > 2: (D1>2)V(D2>2)

Figure 4.14: ALU core operations for problem set 3, determined from the 4:16 decoder's OP signal.

To ensure that the waveform outputs as expected, the following logic arithmetic were done for each individual function in the microtable (**R2** resultant is what is displayed on the SSEG Display):



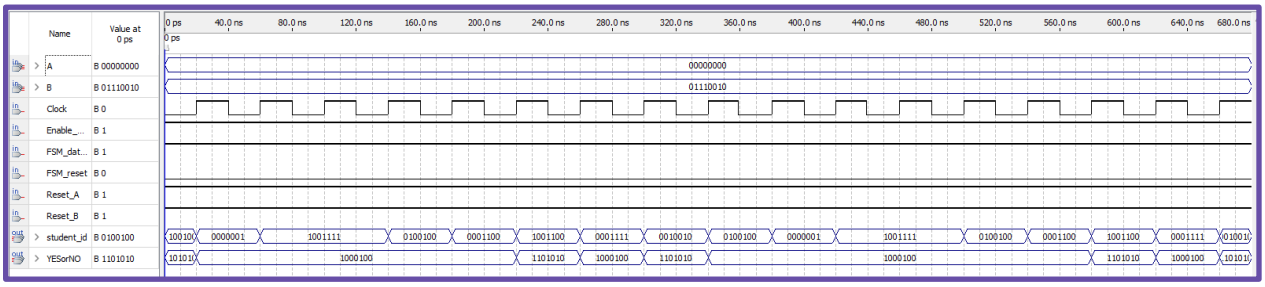


Figure 4.15: Waveform of the Block Diagram for ALU_3 - General Processing Unit, with 2 full iterations.

It can be seen that the outputs of the GPU result as expected, as depending on the state of the FSM, the value fed into the decoder results in a specific microcode value that results in the specific operation as shown in *Figure 4.14*, which in this case is merely a logical arithmetic. Like the ALU waveform shown in *Figure 4.11,* A is not used for this part since the logic operations are only dependent on B's two digit values. As like the other GPU units, the functions displayed are delayed by one full clock cycle due to the latch component not storing the value of B (A is not needed for this example since we are merely comparing the two digits of B) in the first clock cycle since it reads from the previous state which are "don't care" values. This is precisely why the first two student numbers shown in the beginning of the waveform output an 'n' on the SSEG Display. Once one full output of the arithmetic logic cycle is displayed, it goes back to the **`student_id`** of "5" with the logic function

output of the previous state. This is why the **student_id** "9" shows a 'y' rather than the expected 'n' output since it is displaying the output for the previous **student_id** of "5". The **student_id** after "9", "4", displays the expected 'n' since that output belongs to "9", all due to the aforementioned delay by one **Clk** cycle.

For example, after "5", the **student_id** that is associated with □$_1$ holds the **student_id** of "0" presenting *"00000001"* on the SSEG Display. The ALU proceeds to check whether or not "0" is greater than the two digits associated with **B**, *"7"* or *"2"*. Since both digits, although only one is required, is greater than "0", the ALU will output a 8 bit value of *"00000001"*, indicating that there is a value that is greater than the **student_id** mentioned. The first 4 bits of the value, *"0001"* noted as **R1** on the VHDL code shown in *Figure 4.12*, is then taken to the modified SSEG Display shown in *Figure 4.13*, where it converts this value to the SSEG representation of *"1000100"*, which displays the letter "y" on the FPGA board. This output is noted as **YESorNO**. This is then repeated for each subsequent **student_id** per rising **Clk** edge, assuming that the **FSM_datain**, **Enable_decoder**, and **Reset_A** and **Reset_B** are high (1), and with knowing that this output is displayed on the next **student_id** value due to the delay.

Conclusion

To wrap up the making of the Arithmetic and Logic Unit and its components to build and simulate the General Processing Unit, many conclusions can be drawn. The lab experiment was segmented into three distinct parts, each focusing on different tasks performed by the ALU. A critical learning outcome from this lab was the ability to analyze waveform simulations effectively. This skill was particularly highlighted by understanding how the outputs of certain components are significantly influenced by the inputs connected to other components and vice versa.

One certain component that significantly influenced the operation of the ALU would be the FSM, in its Moore representation. For instance, the state of the FSM determined which operation the ALU would perform next, whether it was an arithmetic addition or a logical shift. This reliance on the FSM's states introduced an additional layer of complexity in the operation of the ALU, as depending on its state, the function of the ALU's operations would be changed significantly.

In summarizing this experience, it's evident that a profound understanding was gained not only about the functional intricacies of a simple ALU and its constituent components but also about the step-by-step process from input reception to output display. This experience encapsulates the fundamental principles of digital logic design and the operational dynamics of an essential unit in computer architecture; a microprocessor.

References:

Toronto Metropolitan University - COE 328. (n.d.). Design of a Simple General-Purpose Processor

Appendix (SSEG Code):

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY sseg IS
    PORT (
        bcd  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
        sign : IN  STD_LOGIC;
        leds : OUT STD_LOGIC_VECTOR(0 TO 6);
        ledss: OUT STD_LOGIC_VECTOR(0 TO 6)
    );
END sseg;

ARCHITECTURE Behavior OF sseg IS
BEGIN
    PROCESS (bcd, sign)
    BEGIN
        CASE bcd IS
                                -- abcdefg
            WHEN "0000" => leds <= "0000001"; -- 0
            WHEN "0001" => leds <= "1001111"; -- 1
            WHEN "0010" => leds <= "0010010"; -- 2
            WHEN "0011" => leds <= "0000110"; -- 3
            WHEN "0100" => leds <= "1001100"; -- 4
            WHEN "0101" => leds <= "0100100"; -- 5
            WHEN "0110" => leds <= "0100000"; -- 6
            WHEN "0111" => leds <= "0001111"; -- 7
            WHEN "1000" => leds <= "0000000"; -- 8
            WHEN "1001" => leds <= "0001100"; -- 9
            WHEN "1010" => leds <= "0001000"; -- A
            WHEN "1011" => leds <= "1100000"; -- b
            WHEN "1100" => leds <= "0110001"; -- C
            WHEN "1101" => leds <= "1000010"; -- d
            WHEN "1110" => leds <= "0110000"; -- E
            WHEN "1111" => leds <= "0111000"; -- F
            WHEN OTHERS => leds <= "-------";
        END CASE;

        IF sign = '1' THEN
            ledss <= "1111110";
         ELSE
         ledss <= "1111111";
        END IF;
    END PROCESS;
END Behavior;

```

Figure A: SSEG For ALU Part 1, 2 & 3 to display student ID.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all; -- Import standard logic library

-- Define the entity sseg_modifiedvhd  with input and output ports
ENTITY sseg_modifiedvhd  IS
    PORT(
        bcd: IN STD_LOGIC_VECTOR (3 DOWNTO 0); -- 4-bit binary-coded decimal input
        leds: OUT STD_LOGIC_VECTOR(0 TO 6) -- 7-bit output for LED display
    );
END sseg_modifiedvhd  ;

-- Architecture definition for sseg_modifiedvhd
ARCHITECTURE Behaviour OF sseg_modifiedvhd   IS
BEGIN
    -- Process block sensitive to changes in bcd
    PROCESS(bcd)
    BEGIN
        -- Determine the LED pattern based on the bcd value
        CASE bcd IS
            WHEN "0000" => leds <= "1101010"; -- LED pattern for 'n'
            WHEN "0001" => leds <= "1000100"; -- LED pattern for 'y'
            WHEN OTHERS => leds <= "-------"; -- Default pattern for undefined bcd values
        END CASE;
    END PROCESS;
END Behaviour;
```

Figure B: SSEG Modified for ALU_Part3 to check yes/no conditions.