Toronto Metropolitan University

Department of Electrical, Computer and Biomedical Engineering

COE 328 – Digital Systems

Lab 6 - Design of a Simple Central Processing Unit (2 weeks)

Objectives

To design a simple central processing unit (CPU) in a VHDL environment and implement it on an FPGA board. This includes:

- Designing and building all functions of the arithmetic and logic unit (ALU).
- Functional simulation of the CPU using the Quartus software.

Procedure

Any computer consists of the following components: central processing unit (CPU), memory, and input/output unit (see Fig. 1). The CPU, in turn, comprises an ALU and a control unit. The ALU performs data processing operations. The control unit ensures the operation of the whole system and controls all of its units (including the ALU). The CPU fetches instructions form the memory and executes them. The address of an instruction is supplied by a program counter (PC) that resides in the CPU. In our design, the 3-bit PC is implemented by a finite state machine (FSM), whereas the memory is simulated by a 3-to-8 decoder. The decoder outputs a 1-out-of-8 code that is interpreted as an operation code (opcode) of an instruction. The operands for an operation are supplied by the switches of the Altera board and student ID digits coming from the FSM that implements the PC. This project focuses on all components of a typical ALU.

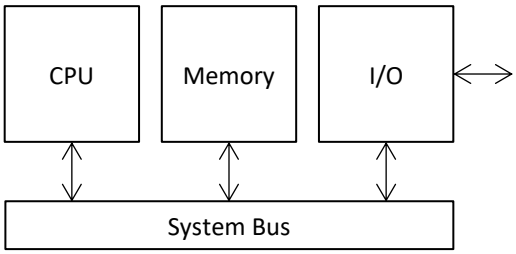

Fig. 1: Computer architecture

Part I: Procuring input data

The ALU is to perform a set of arithmetical and logical functions on two **8-bit inputs** A and B. These inputs are produced using the last four digits of your student ID. For instance, if the student ID is 500864395, then **A =** 43 and **B** = 95, which translates to binary **A =** 0100 0011 and **B** = 1001 0101.

This inputs **A** and **B** are used in Part I and Part II of the lab as 8-bit constants. In part III, in addition to **A** and **B**, the BCD digits of the lab partner's student ID are used as a 4-bit input variable **student_id**. For instance, if the lab partner's student ID is $\{d_1\ d_2\ d_3\ d_4\ d_5\ d_6\ d_7\ d_8\ d_9\}$ = 500435429, the FSM must cycle trough the states $\{s_0\ s_1\ s_2\ s_3\ s_4\ s_5\ s_6\ s_7\}$ and display the digits $\{d_2\ d_3\ d_4\ d_5\ d_6\ d_7\ d_8\ d_9\}$ in a BCD form synchronously with the rising edge of the clock signal, i.e. **student_id** = {0, … , 2, 9} = {0000, … , 0010, 1001}. Ensure to report these values at the beginning of your lab session to your TA.

The general structure of the system is presented in Fig. 2. The digits displayed correspond to the code {r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0} = 10011110 in the Reg. 3 (each of the two nibbles is now interpreted as a signed 4-bit number).

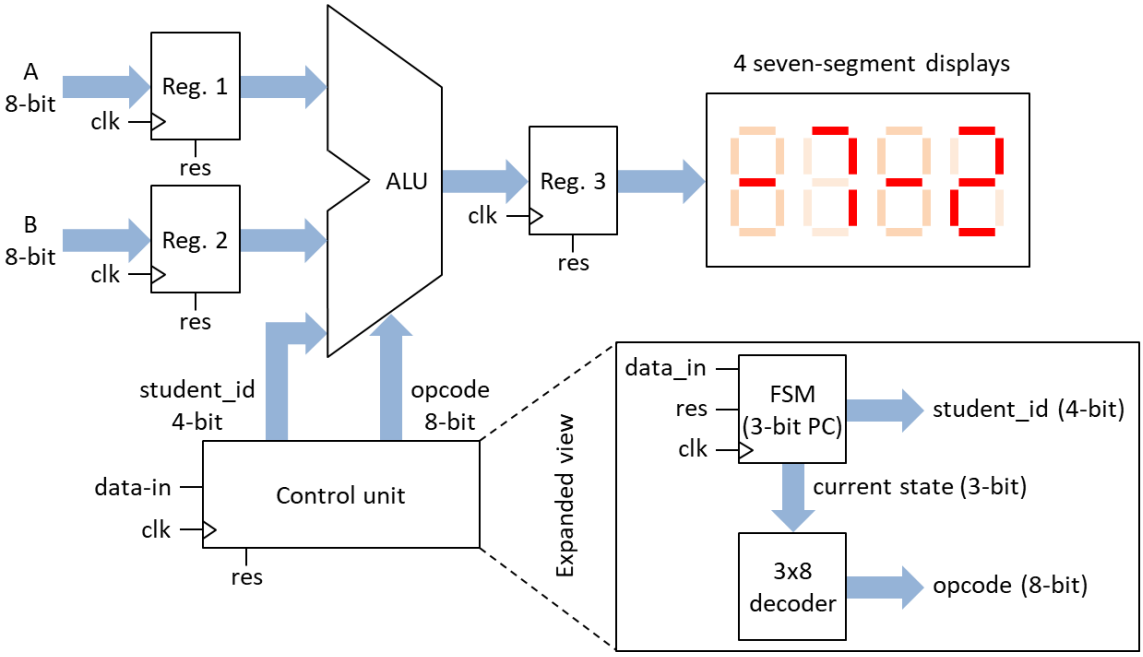

Fig. 2: The block diagram of the CPU

Part II: Storage Unit (Register)

The storage units (in this case registers) are utilized to temporary store the input values and then pass them to the following components in the system. As portrayed in Fig. 2, two 8-bit register units are utilized in the ALU to store inputs **A** and **B**. The register reads the bit values on its input on the rising edge of the clock signal and passes those bit values to the output port.

Write the VHDL code for a register unit using a sample code in Fig. 3. Thereafter, create a symbol of your design to be utilized in the final circuit design. Next, import the symbol to the CPU project. Import the same symbol twice as the system needs one register unit for each respective 8-bit input.

Part III: Control Unit

The Control unit supplies the opcode to the ALU; this opcode defines the operation. This component consists of two sub-components – the FSM and a 3×8 decoder.

Part III (a): Finite State Machine (FSM)

The FSM component of the Control unit defines the pattern of the controller sequence. The student has the option to utilize the FSM design from one of the previous labs in this section while some modification is still required. The FSM designed in the previous lab has 8 different states which are cycled through using the clock signal. The same design is to be implemented in this section.

The FSM takes the clock signal as the input, and produces the 3-bit output **current_state**. The FSM acts as an *up-counter*, cycling through the **states 0 to 7** consecutively (modification required for consecutive state

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

entity register IS
        port (      A  :  in   std_logic_vector(7 downto 0) ; -- 8-bit A input
              res, clk :  in   std_logic ;
                   Q   :  out  std_logic_vector(7 downto 0)) ; -- 8-bit output
end register;

architecture behavior of register is
begin
    process (res, clk)
    begin
        if reset = '1' then
            Q <= "00000000" ;
        elsif (clk'EVENT AND clk = '1') then
            Q <= A ;
        end if ;
    end process ;
end behavior ;
```

Fig. 3: Code template for implementing a Register Storage Unit

transition - see Fig. 4) and back to **state 0**, while the eight digits {d_2, … , d_9} of a **student_id** are displayed on a 7- segment display. The current state of the FSM is passed to the decoder unit. Upon completing the FSM design, create a symbol representing the FSM sub-component which is to be used in the final design.

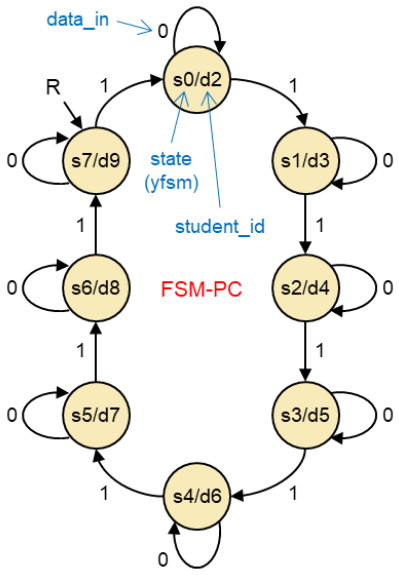

Fig. 4: A state diagram of a program counter with enable input (FSM)

Part III (b): 4 to 16 Decoder (4x16 Dec)

The decoder receives the **current_state** signal from the FSM and converts it to the 1-out-of-8 code that is used as an **opcode** for the ALU. The ALU will then perform one of the functions enlisted in Table 1.

A 2×4 decoder has already been designed and utilized in the previous labs. You can import the same design and extend it to the 3×8 decoder. When the decoder design is completed, create a symbol of the sub-component to be utilized in the final circuit design.

Part IV: Description of the ALU core

A part of every CPU is the ALU core where all arithmetic and logical operations are to be implemented. In this part students are required to implement all functionalities and operations using VHDL syntax compatible with Altera FPGA boards. The ALU core will take two 8-bit data inputs (**A** and **B**) and an 8-bit opcode input from the Control unit. The opcode is the operation-selector signal, deciding the operation that is to be applied on the inputs **A** and **B**. The functions of the ALU core and their corresponding opcodes are listed in Table 1.

Function #	Opcode	Function
1	00000001	$sum(\boldsymbol{A}, \boldsymbol{B})$
2	00000010	$dif(\boldsymbol{A}, \boldsymbol{B})$
3	00000100	$\overline{\boldsymbol{A}}$
4	00001000	$\overline{\boldsymbol{A} \cdot \boldsymbol{B}}$
5	00010000	$\overline{\boldsymbol{A} + \boldsymbol{B}}$
6	00100000	$\boldsymbol{A} \cdot \boldsymbol{B}$
7	01000000	$\boldsymbol{A} \oplus \boldsymbol{B}$
8	10000000	$\boldsymbol{A} + \boldsymbol{B}$

Table 1: ALU Core Operations for Problem 1

The operations in Table 1 are to be implemented by writing the proper VHDL code for the ALU core. A code template for the ALU is presented in Fig. 5. The 8-bit output (**Result** in Reg. 3) is to be displayed on two 7- segment displays. When the ALU Core design is completed, create a symbol to represent this component in the final design.

Part V: Displaying the Output

The ALU core produces an 8-bit output called **Result**, which is the result of the operations applied on A and B.

In the simulation phase of the design, the output **Result** is to be displayed in bit-value format in the waveform editor window. In the implementation phase, where the design is programed on the FPGA board, this output is split into two 4-bit numbers which are to be displayed on two 7-segment displays as signed BCD numbers. The signs of these numbers are displayed on another two 7-segment displays. If the number is negative, the middle segment (*g-segment*) must be turned on; otherwise, all segments are off.

For example, is the **Result** = 1100 0110, then the 7-segment displays will display the numbers: -4 and +6, as illustrated in Fig. 6. The sign of a number is controlled by the signal **Neg**. If the number is negative **Neg** = 1 (*g-segment* "on"), otherwise **Neg** = 0 (*g-segment* "off").

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

entity ALU_unit is -- ALU unit includes Reg. 3
      port (      clk, res : in    std_logic ;
            Reg1, Reg2 : in    std_logic_vector(7 downto 0);  -- 8-bit inputs A & B from Reg. 1 & Reg. 2
                 opcode : in    std_logic_vector(7 downto 0);  -- 8-bit opcode from Decoder
                  Result : out  std_logic_vector(7 downto 0)); -- 8-bit Result
end ALU_unit ;

architecture calculation of ALU_unit is
begin
      process ( clk, res )
      begin
          if reset = '1' then
               Result <= "00000000" ;
          elsif (clk'EVENT AND clk = '1') then
               case opcode is
                    when "00000001" =>
                         -- Do addition for Reg1 and Reg2
                    when "00000010" =>
                         -- Do subtraction for Reg1 and Reg2
                    when "00000100" =>
                         -- Do inverse
                    when "00001000" =>
                         -- Do Boolean NAND
                    when "00010000" =>
                         -- Do Boolean NOR
                    when "00100000" =>
                         -- Do Boolean AND
                    when "01000000" =>
                         -- Do Boolean XOR
                    when "10000000" =>
                         -- Do Boolean OR
                    when others =>
                         -- Don't care, do nothing
               end case ;
          end if ;
      end process ;
end calculation ;
```

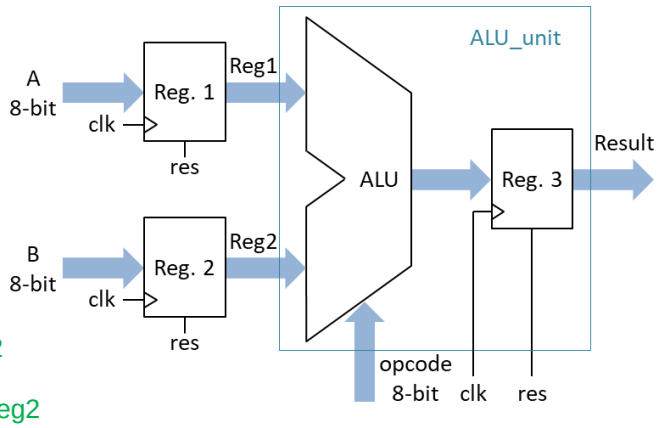

Fig. 5: A code template for the ALU_unit

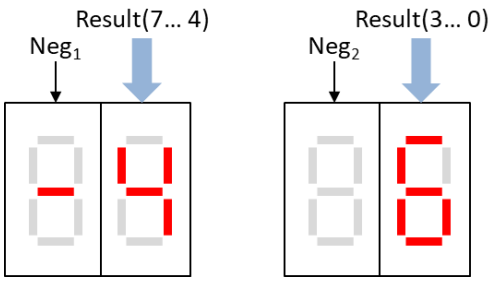

Fig. 6: Four 7-sgment displays representing the numbers -4 and +6

Part VI: Final Design

As the designs for different components (Register, ALU Core, FSM and Decoder units) are completed, they should all be ported to one final circuit design.

1. Open a new schematic design and import all of the required units. When importing is completed, you should have the following components: **two** Registers (for inputs **A** and **B**), **one** ALU Core, **one** FSM, **one** Decoder, **three** pairs of 7-segment displays: **one** for **student_id** (unsigned number) and **two** for the output **Result** from the ALU (signed numbers).
2. Create the 8-bit input variables **A** and **B**, 8-bit output variable **Result**, and the single-bit input variable **clk**.
3. Connect all of the components using single and multi-bit data buses. Follow the schematics portrayed in Fig. 7.
4. Name the data buses to represent the proper signals, for e.g. **opcode** and **A**.

When your final circuit looks similar to the one in Fig. 7, synthesize and simulate your design. Verify the functionality of your ALU and present the results to the TA. The waveforms must be included as part of the final report submission.

Part VIII: Problem Sets

In this section, you should address the following *three* problems and showcase the results to your TA.

Problem 1: Initial Design

Implement the initial design of the CPU. In this design, the FSM output signal, **current_state** follows an up-counting pattern. This signal feeds a decoder that is then connected to the ALU inputs that select the required function. As the value of the **current_state** changes, so does the ALU function. The output of the ALU, **Result**, represents the result of the current operation (function). The set of available functions is given in Table 1; the ALU is presented in Fig. 2 (**note:** in this problem, there are only two data inputs to the ALU, **Reg1** and **Reg2**; ignore **student-id** input).

Problem 2: Modified ALU Core 1

In this problem, the student is required to modify the functions of the ALU from Problem 1. The TA will assign each student one of the following modified functions for the ALU.

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

b)

Function #	Operation / Function
1	Swap the lower and upper 4 bits of **A**
2	Produce the result of ORing **A** and **B**
3	Decrement **B** by 5
4	Invert all bits of **A**
5	Invert the bit-significance order of **A**
6	Find the greater value of **A** and **B** and produce the results (Max(**A**,**B**))
7	Produce the difference between **A** and **B**
8	Produce the result of XNORing **A** and **B**

c)

Function #	Operation / Function
1	Produce the difference between **A** and **B**
2	Produce the 2's complement of **B**
3	Swap the lower 4 bits of **A** with lower 4 bits of **B**
4	Produce null on the output
5	Decrement **B** by 5
6	Invert the bit-significance order of **A**
7	Shift **B** to left by three bits, input bit = 1 (SHL)
8	Increment **A** by 3

d)

Function #	Operation / Function
1	Shift **A** to right by two bits, input bit = 1 (SHR)
2	Produce the difference of **A** and **B** and then increment by 4
3	Find the greater value of **A** and **B** and produce the results (Max(**A**,**B**))
4	Swap the upper 4 bits of **A** by the lower 4 bits of **B**
5	Increment **A** by 1
6	Produce the result of ANDing **A** and **B**
7	Invert the upper four bits of **A**
8	Rotate **B** to left by 3 bits (ROL)

e)

Function #	Operation / Function
1	Replace the odd bits of **A** with odd bits of **B**
2	Produce the result of NANDing **A** and **B**
3	Calculate the summation of **A** and **B** and decrease it by 5
4	Produce the 2's complement of **B**
5	Invert the even bits of **B**
6	Shift **A** to left by 2 bits, input bit = 1 (SHL)
7	Produce null on the output
8	Produce 2's complement of **A**

f)

Function #	Operation / Function
1	Decrement **B** by 9
2	Swap the lower and upper 4 bits of **B**
3	Shift **A** to left by 2 bits, input bit = 0 (SHL)
4	Produce the result of NANDing **A** and **B**
5	Find the greater value of **A** and **B** and produce the results (Max(**A**,**B**))
6	Invert the even bits of **B**
7	Produce null on the output
8	Replace the upper four bits of **B** by upper four bits of **A**

g)

Function #	Operation / Function
1	Invert the bit-significance order of **A**
2	Shift **A** to left by 4 bits, input bit = 1 (SHL)
3	Invert upper four bits of **B**
4	Find the smaller value of **A** and **B** and produce the results (Min(**A**,**B**))
5	Calculate the summation of **A** and **B** and increase it by 4
6	Increment **A** by 3
7	Replace the even bits of **A** with even bits of **B**
8	Produce the result of XNORing **A** and **B**

h)

Function #	Operation / Function
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 by 2 bits, input bit = 0 (SHR)

Problem 3: Modified ALU Core 2

In this problem, students are assigned the task to utilize the **student_id** output from the FSM component of the Control Unit. Use the ALU from the Problem 1, but modify its functions so that it can process 3 data inputs, **Reg1**, **Reg2** and **student-id** (as shown in Fig. 2 and Fig. 8). Implement the functionalities described below which are assigned by your TA. Some modifications to the 7-segment display VHDL code may also be needed to display "y" or "n" symbols instead of numerical digits. The TA shall assign one of the following problems for each student.

Note: the inputs **A** and **B** for this problem come from your partner's student ID, whereas the **student-id** digits generated by the FSM must correspond to your student ID. Ensure to report your partner's student ID digits to your TA.

a) For each opcode submitted to the ALU, display 'y' if the **student_id** signal value is odd and 'n' otherwise
b) For each opcode submitted to the ALU, display 'y' if the **student_id** signal value is even and 'n' otherwise
c) For each opcode submitted to the ALU, display 'y' if the **student_id** signal has an odd parity and 'n' otherwise
d) For each opcode submitted to the ALU, display 'y' if the **student_id** signal has an even parity and 'n' otherwise
e) For each opcode submitted to the ALU, display 'y' if one of the 2 digits of A is greater than the **student_id** signal value and 'n' otherwise
f) For each opcode submitted to the ALU, display 'y' if one of the 2 digits of A is less than the **student_id** signal value and 'n' otherwise
g) For each opcode submitted to the ALU, display 'y' if one of the 2 digits of A is equal to the **student_id** signal value and 'n' otherwise
h) For each opcode submitted to the ALU, display 'y' if one of the 2 digits of B is greater than the **student_id** signal value and 'n' otherwise
i) For each opcode submitted to the ALU, display 'y' if one of the 2 digits of B is less than the **student_id** signal value and 'n' otherwise
j) For each opcode submitted to the ALU, display 'y' if one of the 2 digits of B is equal to the **student_id** signal value and 'n' otherwise

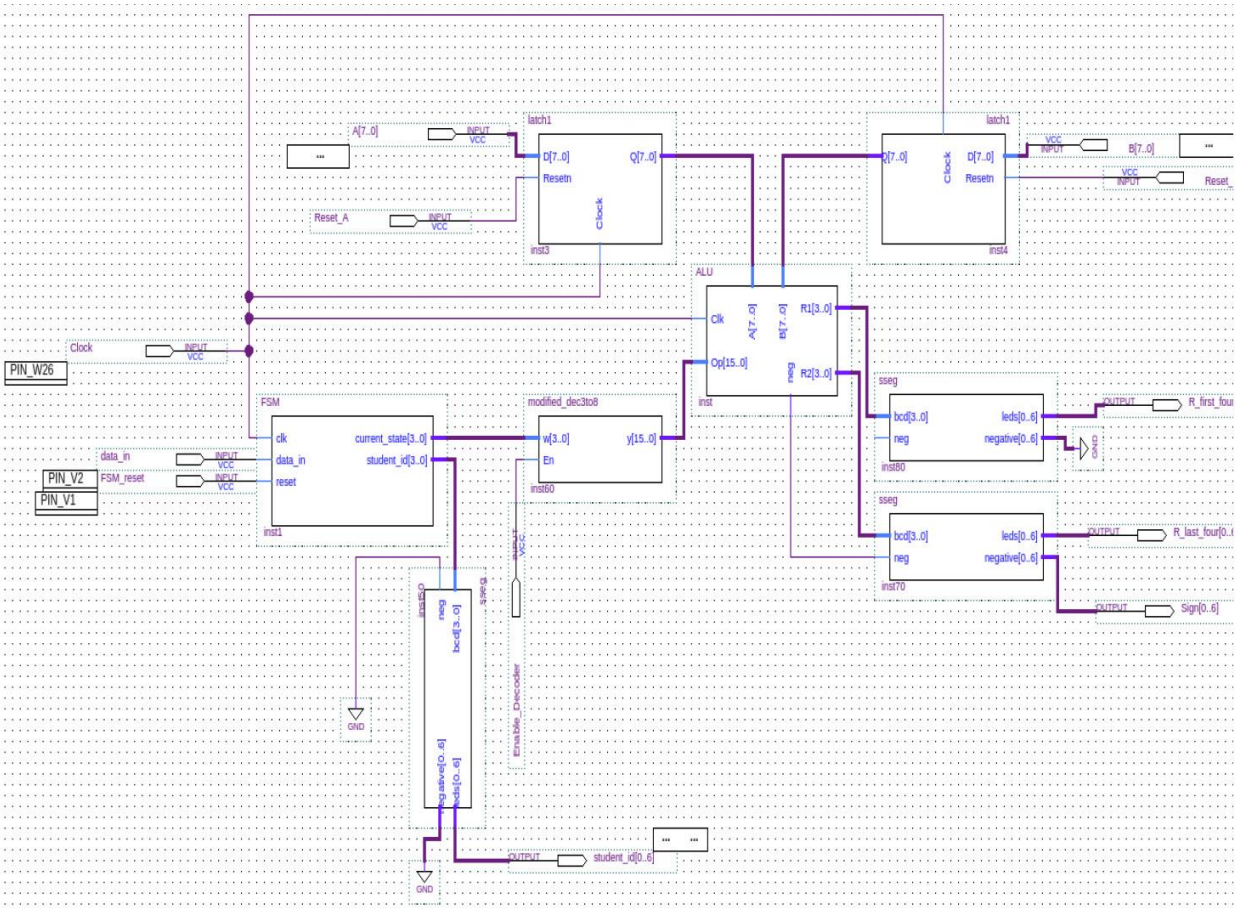

Fig. 7: Typical Block Schematic for Problem 1

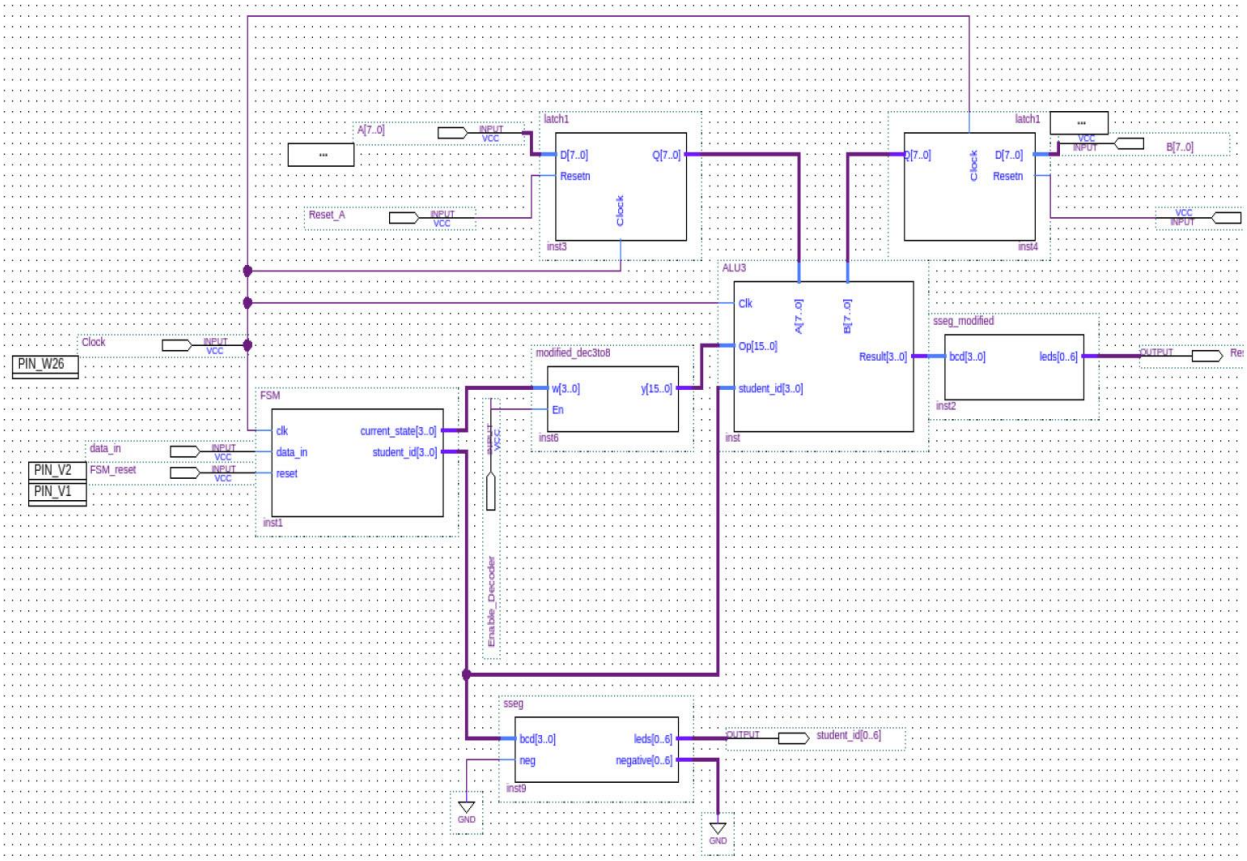

Fig. 8: Typical Block Schematic for Problem 3