Ryerson University

Department of Electrical and Computer Engineering

COE/BME 328 – Digital Systems

Lab 6

Design of a Simple General-Purpose Processor

35 Marks (2 **weeks**)
Due Date: Week 12

Objective:

This objective of this laboratory experiment is to design and construct an Arithmetic and Logic Unit (ALU) in VHDL environment and implement it on an FPGA board.

- Design and build all functions of the ALU.
- Design, simulate the ALU using VHDL (Based on Functional Simulation in Quartus Simulator).

Procedure:

This unit consists of different parts that would come together to create a functional ALU. A processing unit is usually divided to 4 distinct components. The *control* unit performs the fetching of instructions and signals. The bus controls access to data values throughout the processing unit - these *registers* act as temporary storage units. The ALU Core performs the arithmetic and logical operations on desired inputs and produces the required outputs. This project focuses on all four components of a typical ALU.

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 first utilized in simulation phase through waveform editor on Quartus II, Inputs **A** and **B** are procured using the last four digits of your student ID. For instance, if the student ID is 500864395, then $A = (\mathbf{43})_{\mathbf{16}}$ and $= (\mathbf{95})_{\mathbf{16}}$, translating to

$$\boldsymbol{A} = (\mathbf{0100\ 0011})_{\mathbf{2}}; \boldsymbol{B} = (\mathbf{1001\ 0101})_{\mathbf{2}}$$

Make sure that you ask for the last four digits of your lab partner's student ID and utilize those values for the rest of this project as inputs **A** and **B**. Ensure to report these values at the beginning of your lab session to your TA.

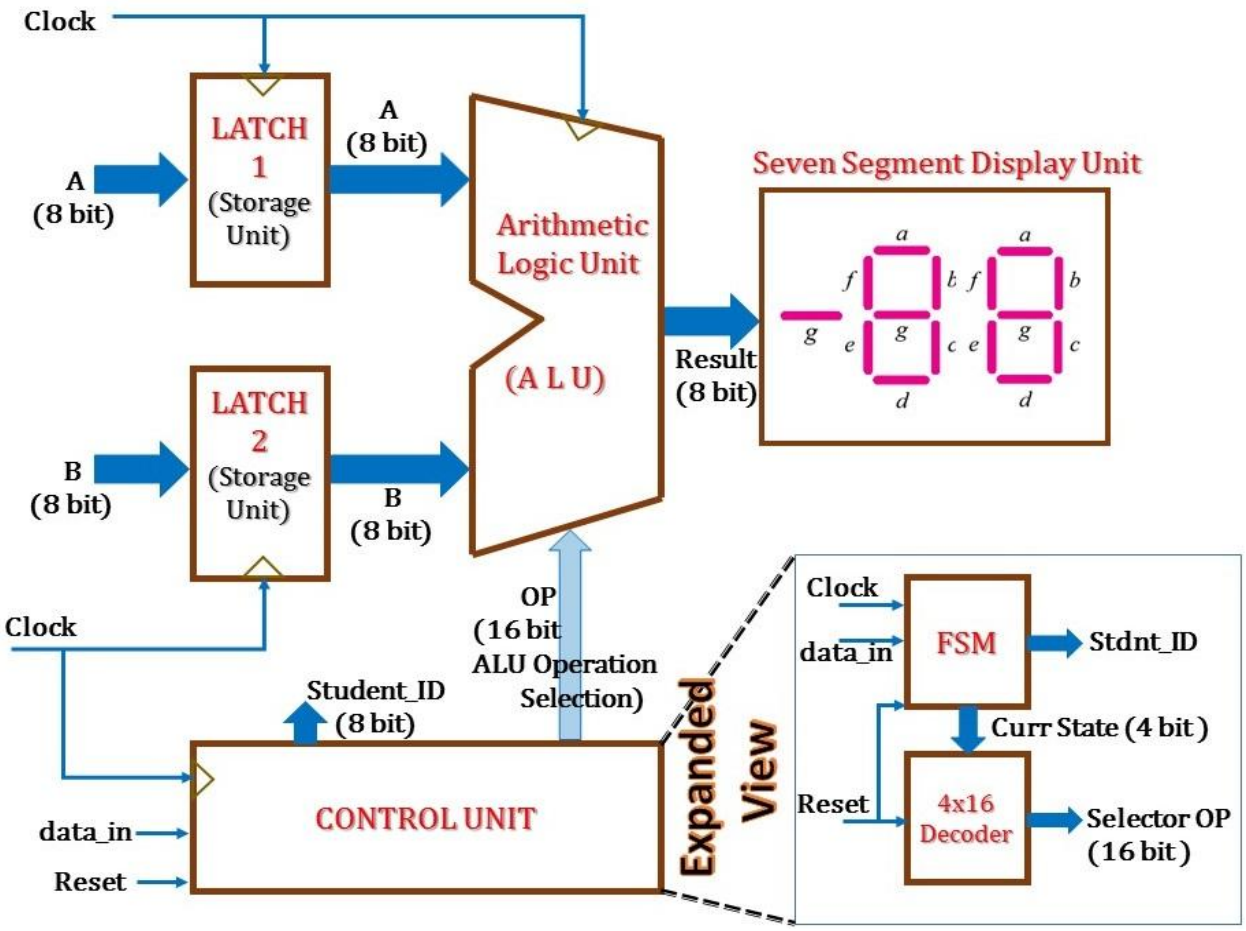

Figure 1. The Block Diagram of the GPU

Part II: Storage Unit (Register)

The storage units which are sometimes called Registers are utilized to temporary store the input values and then pass them to the following components in the system. As portrayed in Figure 1, 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 on the next rising edge of the clock signal.

Write the VHDL code using Figure 2 for a register unit and confirm its functionality with the help of a truth table. Thereafter, create a symbol of your design to be utilized in the final circuit design. Next, import the symbol to the GPU project. Import the same symbol twice as the system needs one register unit for each respective 8-bit input.

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

ENTITY latch1 IS
    PORT ( A : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ; --8 bit A input
           Resetn, Clock : IN STD_LOGIC ;--1 bit clock input and 1 bit reset input bit
           Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;-- 8 bit output
END latch1 ;
ARCHITECTURE Behavior OF latch1 IS
BEGIN
    PROCESS ( Resetn, Clock )--Process takes reset and clock as inputs
    BEGIN
        IF Resetn = '0' THEN -- when reset input is '0' the latches does not operate
            Q <= "00000000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN -- level sensitive based on clock
            Q <= A;
        END IF;
    END PROCESS;
END Behavior;
```

Figure 2. Code template for implementing Register Storage Unit.

Part III: Control Unit

The Control unit decides the microcode that is to be delivered to the ALU core and will act as the operations-selector for the ALU core. The Control Unit produces an output **OP**, which is passed to ALU core as the operations selector. This component consists of two sub-components – the FSM and a 4 to 16 decoder.

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

The FSM component of the Control unit decides 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 may still be required. The FSM designed in the previous lab has 9 different states which are cycled through using the clock signal. The same design is to be implemented in this section.

In the initial design of the Control unit, the current state of the FSM is to be passed to the decoder unit. In other words, the FSM sub-component in the initial Control unit design will only act as an *up-counter*, cycling through **states 0 to 8** consecutively (modification required for consecutive state transition) and back to **state 0** while the **student_id** will be displayed on a 7-segment. The FSM takes the clock signal as the input, and produces the 4-bit output **current_state** and passes it to the decoder sub-component. Upon completing the FSM design, create a symbol representing the FSM sub-component which is to be used in the final design.

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

The decoder is tasked with receiving the signal **current_state** from FSM and decoding it to the operation-selector microcode. In the initial design of the Control unit, the Decoder unit passes the signal **OP** to the ALU core, which will then translate to operations selector for the ALU core and follow the functions enlisted in Table 1.

A 3x8 decoder design has been designed and utilized in the previous labs. You can import the same design and extend it to the 4x16 decoder following the schematics portrayed in Figure 2. When the decoder design is completed, create a symbol of the sub-component to be utilized in the final circuit design.

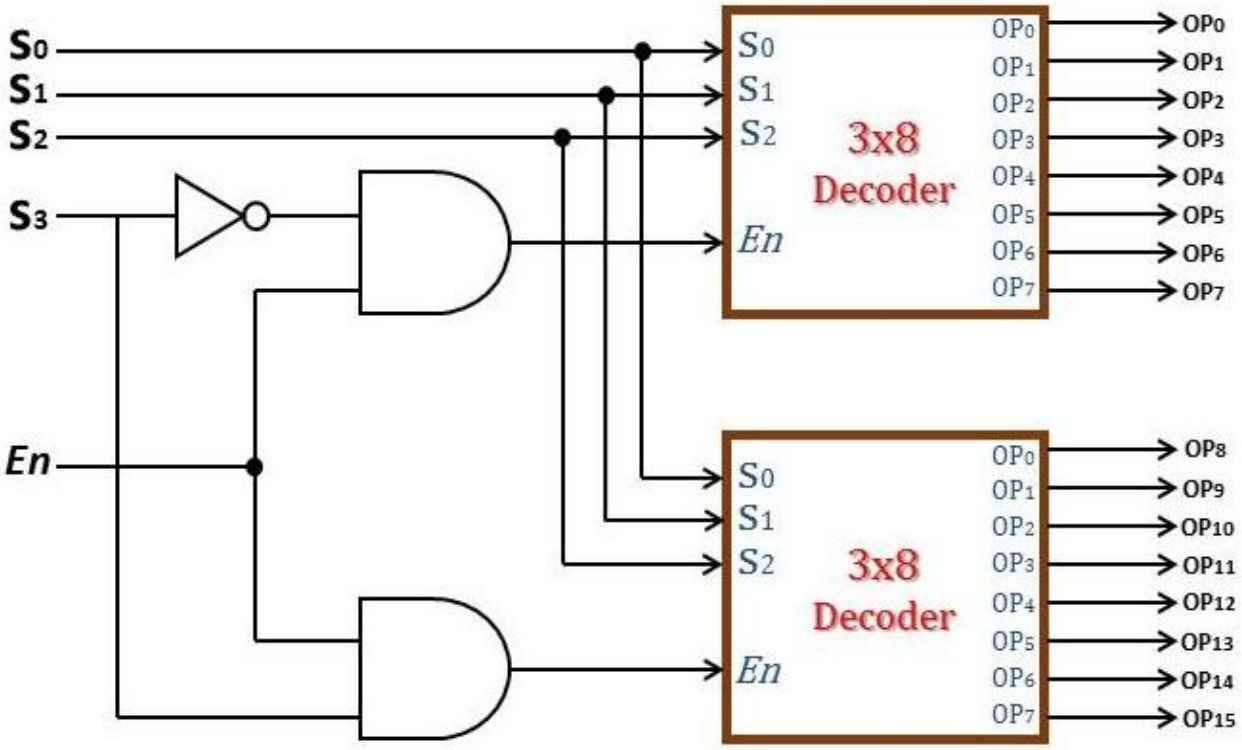

Figure 3. Implementation of 4x16 Decoder using 3x8 decoders

Part IV: Description of the ALU core

The heart of every GPU unit is the ALU core where all arithmetic and logical operations are to be implemented and applied as required. 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 inputs (**A** and **B**) and a 16-bit input from Control unit. The microcode input from controller unit is the operation-selector signal, deciding the operation that is to be applied on the inputs **A** and **B**. Although the microcode received from the Control unit is 16 bits, only 9 distinct operations are to be implemented. The functionalities of the ALU core and their corresponding microcode are listed in Table 1. The microcode is delivered from the Control unit and will decide what operation the inputs would undergo.

Operations listed in Table 1 are to be implemented in the ALU core by writing the proper VHDL code for the ALU core. The 8-bit output (**Result**) of the ALU Core is to be displayed on two 7-segment display or LEDs. When the ALU Core design is completed, create a symbol to represent this component in the final design.

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

Table 1. ALU Core Operations for Problem 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
entity ALU is
port( Clock : in std_logic; --input clock signal
      A,B : in unsigned(7 downto 0); --8-bit inputs from latches A and B
      student_id : in unsigned(3 downto 0); --4 bit student id from FSM
      OP : in unsigned(15 downto 0); --16-bit selector for Operation from Decoder
      Neg: out std_logic; --is the result negative ? Set-ve bit output
      R1 : out unsigned(3 downto 0); -- lower 4-bits of 8-bit Result Output
      R2 : out unsigned(3 downto 0));   -- higher 4-bits of 8-bit Result Output
end ALU;
architecture calculation of ALU is --temporary signal declarations.
signal Reg1,Reg2,Result : unsigned(7 downto 0) := (others => '0');
signal Reg4 : unsigned (0 to 7);
begin
Reg1 <= A; --temporarily store A in Reg1 local variable
Reg2 <= B; --temporarily store B in Reg2 local variable
process(Clk, OP)
begin
    if(rising_edge(Clock)) THEN --Do the calculation @ positive edge of clock cycle.
        case OP is
          WHEN "0000000000000001" =>
                    --Do Addition for Reg1 and Reg2
               WHEN "0000000000000010" =>
                    --Do Subtraction
                    --"Neg" bit set if required.
               WHEN "0000000000000100" =>
                    --Do Inverse
               WHEN "0000000000001000" =>
                    --Do Boolean NAND
               WHEN "0000000000010000" =>
                    --Do Boolean NOR
               WHEN "0000000000100000" =>
                    --Do Boolean AND
               WHEN "0000000001000000" =>
                    --Do Boolean OR
               WHEN "0000000010000000" =>
                    --Do Boolean XOR
               WHEN "0000000100000000" =>
                    --Do Boolean XNOR
               WHEN OTHERS =>
                    --Don't care, do nothing
        end case;
    end if;
end process;
R1 <= Result(3 downto 0);   --Since the output seven segments can
R2 <= Result(7 downto 4);   -- only 4-bits, split the 8-bit to two 4-bits.
end calculation;
```

Figure 4. Code Template for ALU Core.

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 implementation phase where the design is programed on the FPGA board, this output is to be displayed on two 7-segment displays in hexadecimal format.

For example, if then the 7-segment combined unit displays **C6** as illustrated in Figure 4.

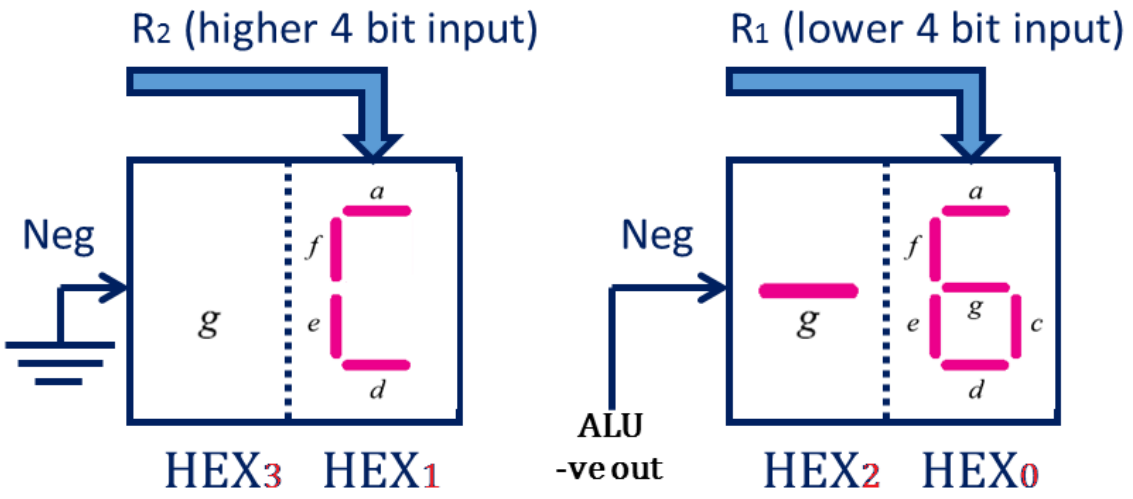

Figure 5. Typical 8-bit Seven Segment Display

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 the required units. When importing is completed, you should have the following components: **two** Registers (one per input), **one** ALU Core, **one** FSM, Decoder, **three** 7-segment displays (one for **student_id** and **two** for the output ***Result*** from the ALU).
2. Create the 8-bit input ports **A** and **B**, 8-bit output port **Result** and the single-bit input **Clock** port.
3. Connect all the components using single and multi-bit data buses. Follow the schematics portrayed in Appendix A. Figure 6.
4. Name the data buses to represent the proper signals, for e.g. **OP** and **A**.

When your final circuit looks similar to Appendix A. Figure 6, 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 problems and showcase the results to your TA.

Problem 1: Initial Design

Implement the initial design of the General Processor Unit (GPU). In this design, the FSM output **current_state** follows an up-counting pattern, which will then dictate the operation selector signal to the ALU core. Thus, the output of the ALU core, **Result**, will represent the outputs of operations shown in Table 1 and following the same order.

Problem 2: Modified ALU Core

In this problem, the student is tasked to modify the ALU core and its functionalities. The TA will assign each student with one of the following modifications to the ALU core.

a)

Function #	Operation / Function
1	Increment **A** by 2
2	Shift **B** to right by two bits, input bit = 0 (SHR)
3	Shift **A** to right by four bits, input bit = 1 (SHR)
4	Find the smaller value of **A** and **B** and produce the results (Min(**A**,**B**))
5	Rotate **A** to right by two bits (ROR)
6	Invert the bit-significance order of **B**
7	Produce the result of XORing **A** and **B**
8	Produce the summation of **A** and **B**, then decrease it by 4
9	Produce all high bits on the output

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**
9	Rotate **B** to left by three bits (ROL)

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
9	Invert all bits of **B**

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)
9	Show null on the output

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**
9	Rotate **B** to right by 2 bits (ROR)

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**
9	Show **A** on the output

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**
9	Rotate **B** to right by 3 bits (ROR)

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)
9	Invert lower four bits of **A**

Problem 3: Modified Control Unit (FSM)

In this problem, students are assigned the task to utilize the **student_id** output from the FSM sub-component of the Control Unit. Use the ALU and the microcode from the initial design section as described in Figure 1 and modify it so that it can process 4 inputs as per the diagram in Appendix A. Figure 7. Implement the functionalities described below which are assigned by

your TA. Some modifications to the 7 segment may also be needed to display “y” or “n”. The TA shall assign one of the following problems for each student.

a) For each microcode instruction, display 'y' if the FSM output (**student_id**) is odd and 'n' otherwise

b) For each microcode instruction, display 'y' if the FSM output (**student_id**) is even and 'n' otherwise

c) For each microcode instruction, display 'y' if the FSM output (**student_id**) had an odd parity and 'n' otherwise

d) For each microcode instruction, display 'y' if the FSM output (**student_id**) had an even parity and 'n' otherwise

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

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

g) For each microcode instruction, 'y' if one of the 2 digits of A are equal to FSM output (**student_id**) and 'n' otherwise. Use the microcode instruction from part 1 of the lab.

h) 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.

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

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

- Appendix A

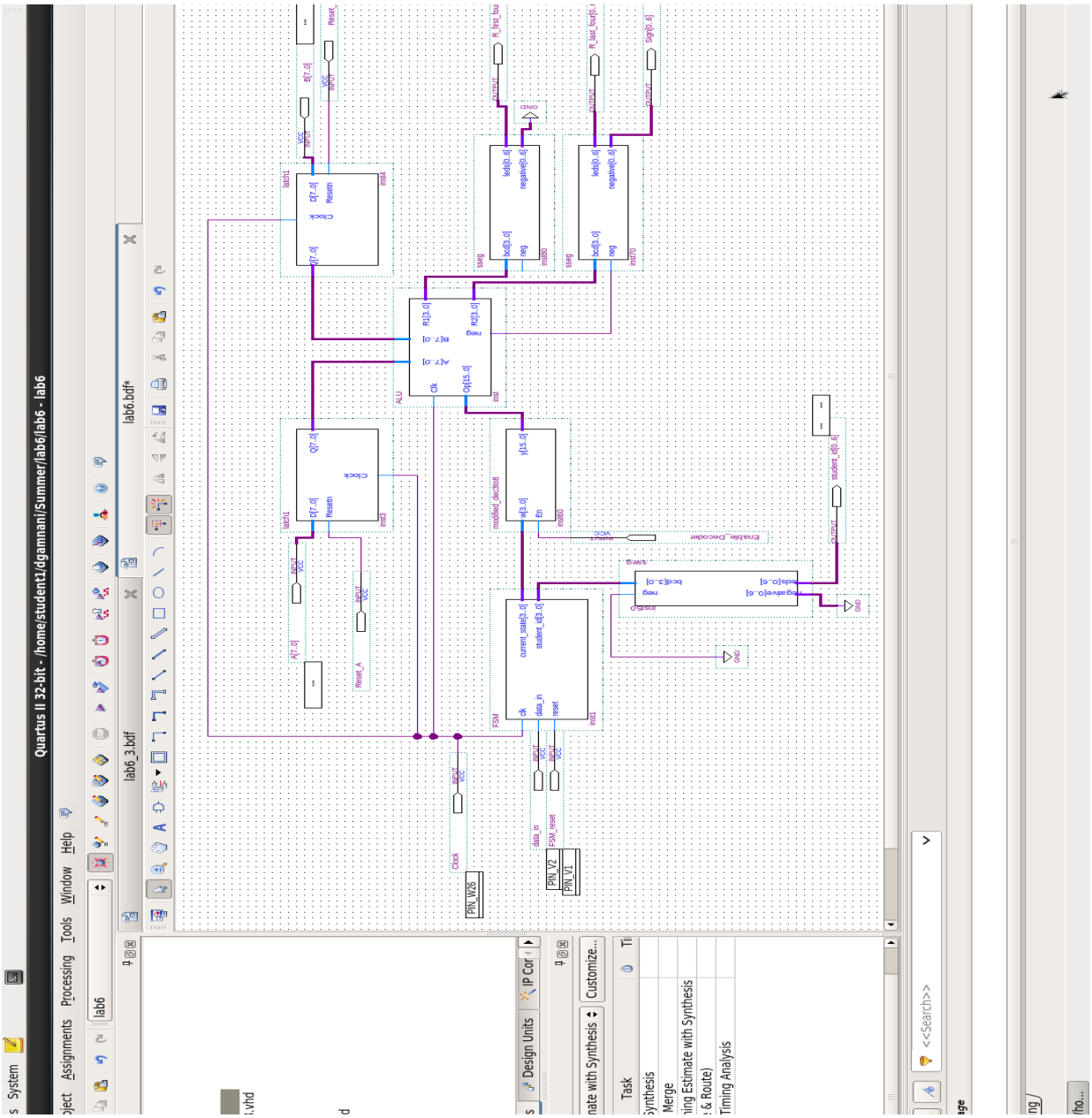

Figure 6. Typical Block Schematic for Problem 1

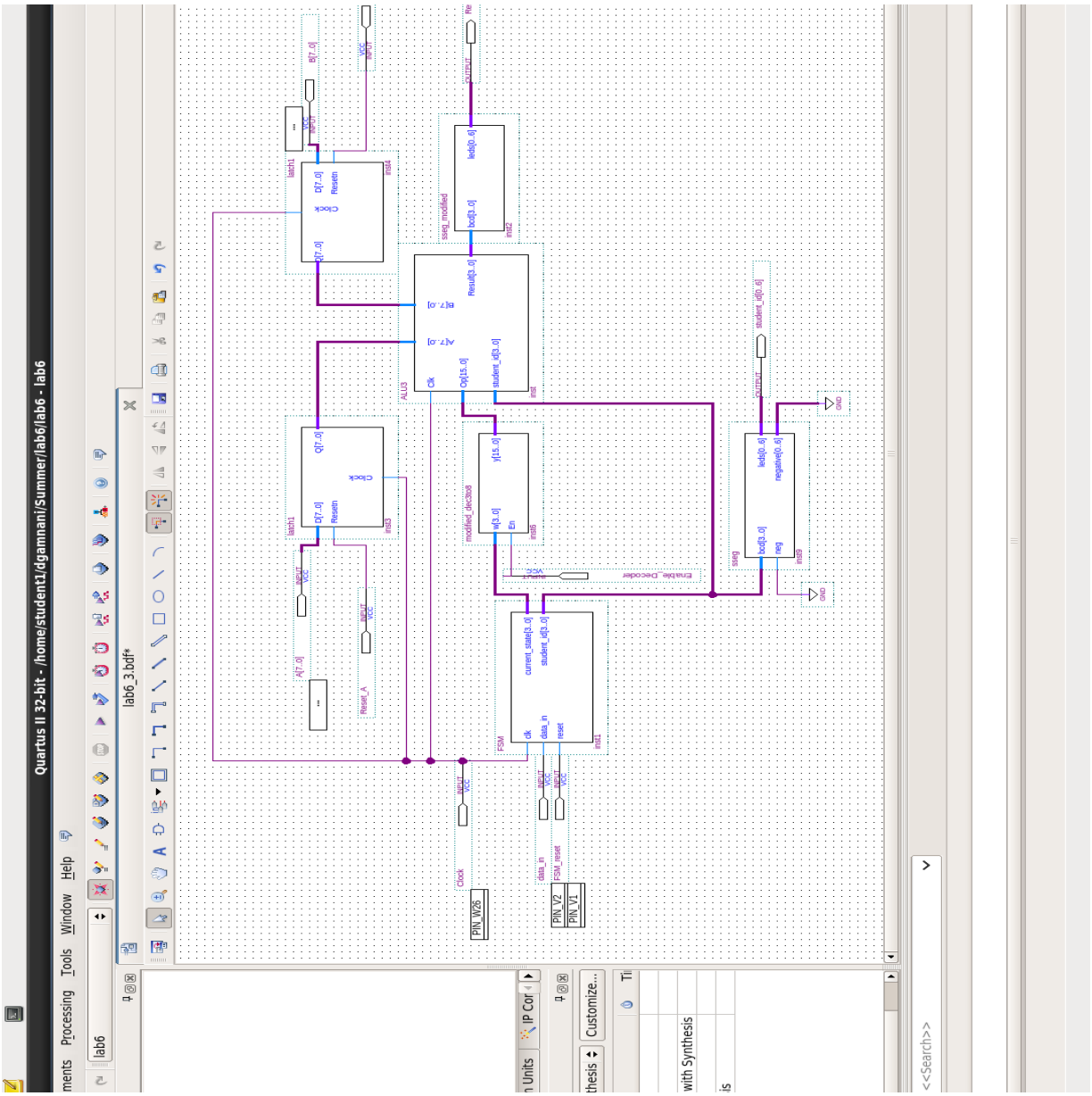

Figure 7. Typical Block Schematic for Problem 3