Toronto Metropolitan University
Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

# Digital Systems

---

# General Purpose Processor

Lab 6 COE 328
Section 15
Faraz Sadrzadeh-Afsharazar
Dr. Arghavan Asad

Sheldon Cerejo
501173492

# Table of Contents

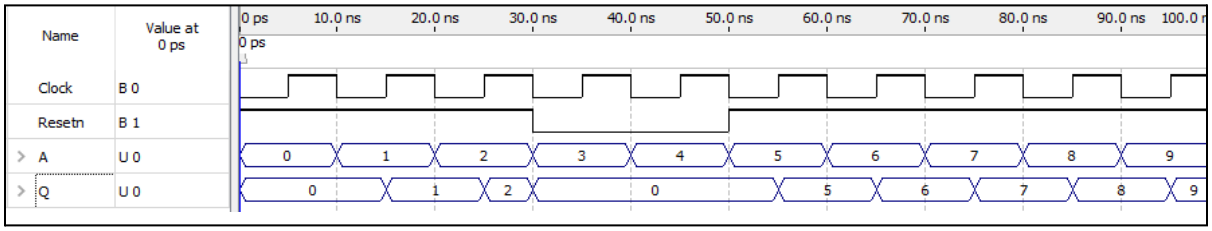
# Introduction

The objective of this lab was to be able to utilize the skills and knowledge obtained from previous labs to create a general purpose processor. This was done by combining components such as latches, a decoder, a finite-state machine (FSM) and an arithmetic logic unit (ALU) to create this machine. The information processed through this machine was then displayed digitally using several seven segment displays (SSEG) devices on the Altera FPGA boards. Alternatively, the data can be presented with their respective waveform diagrams that can be generated directly on the Quartus II software. The combination of all these different parts resulted in the creation of the processing unit which is present in computers worldwide, each part being responsible for specific functions. The latches being responsible for memory, the FSM used to control how each state shows up in sequence, the decoder to convert signals and the ALU is responsible for the various operations and calculations to be done with the data provided to the system. These components in tangent with the seven segment displays the data obtained from the various components which can be easily read through and LEDs which can output the data in 8-bit hexadecimal form.

# Components

## Latches

The latches in this case are exactly the same, the only difference is the inputs that the laches receive. One latch connects to input A which is the first two digits of the last four digits in the student number and B corresponds to the 8-bit binary of the last two digits in the student number. The point of latches is to provide temporary storage for the data that is inserted using the A and B input pins. This latch is positive edge-triggered, meaning that they are level sensitive, everytime there is a positive edge, when the level goes from low to high the latch determines whether the input is stored as memory or not.




Waveform

| Present State (y) | Next state | | Count (z) |
|---|---|---|---|
| | W = 0 | W = 1 | |

| | X | Y | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |

State table for the 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
	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;
```

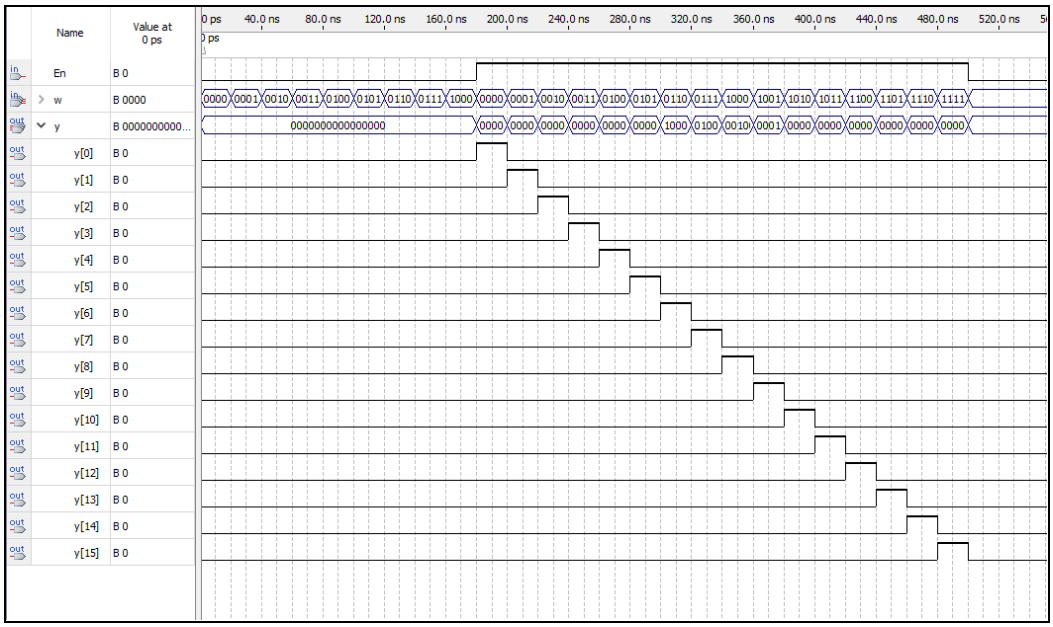
VHDL Code for latches

# 4-16 Decoder

The decoder takes in the inputs as 4-bits and converts them into a 16-bit signal. This unit helps break down the incoming signal into more bits that can be processed as digestible information through the ALU. This then allows the ALU to process that information to perform mathematical equations with.

| Inputs | | | | Outputs | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w_0$ | $w_1$ | $w_2$ | $w_3$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ | $y_7$ | $y_8$ | $y_9$ | $y_{10}$ | $y_{11}$ | $y_{12}$ | $y_{13}$ | $y_{14}$ | $y_{15}$ |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

4-16 Decoder Truth Table



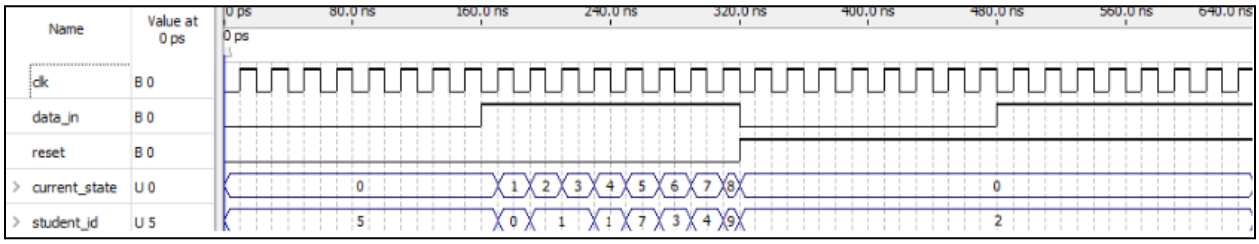

4-16 Decoder Waveform

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

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

END decod;

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

VHDL Code for 4-16 Decoder

# Finite-State Machine (FSM)

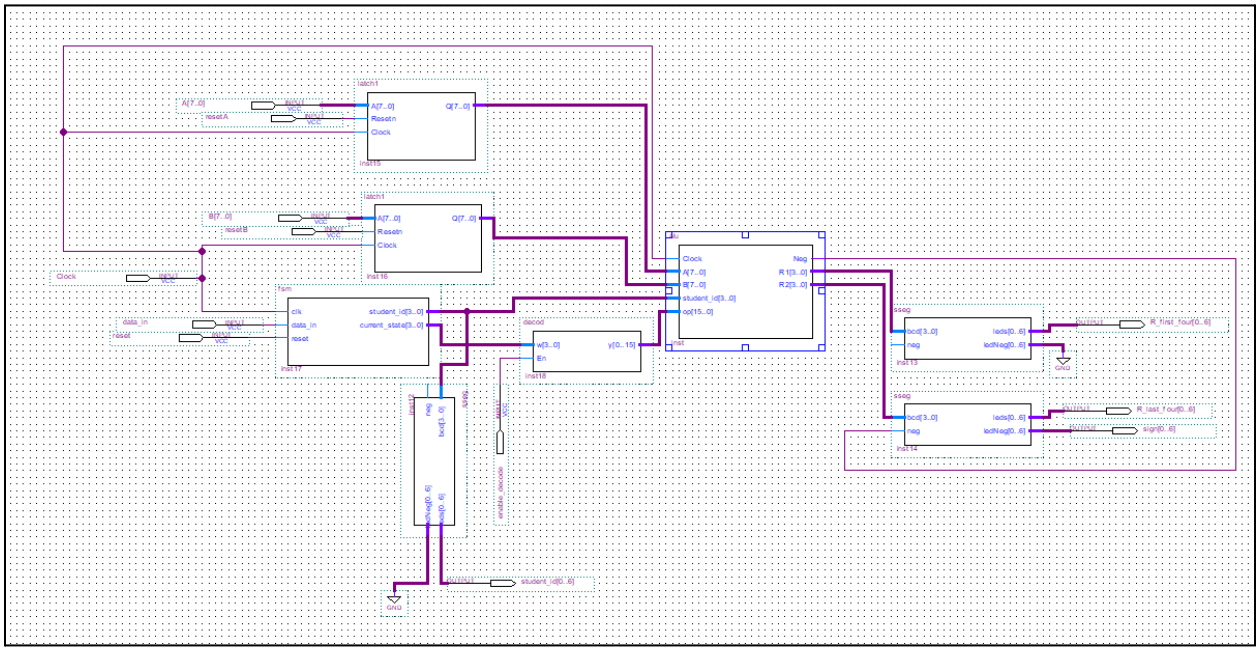
The FSM as mentioned earlier is the component that depicts each state sequence in this lab. This unit takes in the student ID and the data in and outputs a present state which is forwarded to the decoder. Typically there are two types of FSM's, the Mealy machine and the Moore machine but in the case of this lab, the Moore machine is used. The reason the Moore machine is used is due to the fact that using this alternative only requires the current state to generate an output. The Mealy machine requires both current state and input state to generate an output.




FSM Waveform

| Present State | Next State | | Count |
|---|---|---|---|
| | W = 0 | W = 1 | |
| $y_3y_2y_1y_0$ | $Y_3Y_2Y_1Y_0$ | $Y_3Y_2Y_1Y_0$ | $z_3z_2z_1z_0$ |
| 0000 | 0000 | 0001 | 0000 |
| 0001 | 0001 | 0010 | 0001 |
| 0010 | 0010 | 0011 | 0010 |
| 0011 | 0011 | 0100 | 0011 |
| 0100 | 0100 | 0101 | 0100 |
| 0101 | 0101 | 0110 | 0101 |
| 0110 | 0110 | 0111 | 0110 |
| 0111 | 0111 | 1000 | 0111 |
| 1000 | 1000 | 1001 | 1000 |
| 0000 | 0000 | 0001 | 0000 |
| 0001 | 0001 | 0010 | 0001 |
| 0010 | 0010 | 0011 | 0010 |
| 0011 | 0011 | 0100 | 0011 |
| 0100 | 0100 | 0101 | 0100 |
| 0101 | 0101 | 0110 | 0101 |
| 0110 | 0110 | 0111 | 0110 |

# ALU Problem I: Initial Design



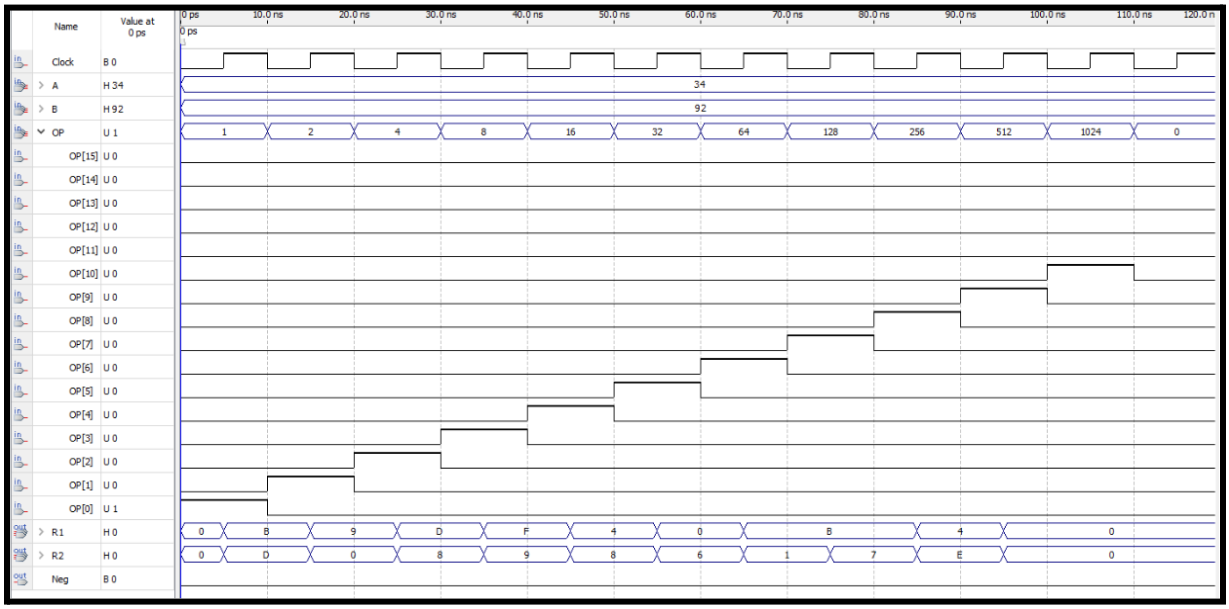
Block Diagram Schematic

| Function # | Microcode | Boolean Operation/Function | Answer |
|---|---|---|---|
| 1 | 0000000000000001 | sum(**A,B**) | C6 |
| 2 | 0000000000000010 | diff(**A,B**) | -A2 |
| 3 | 0000000000000100 | not(**A**) | -CB |
| 4 | 0000000000001000 | nand(**A,B**) | -EF |
| 5 | 0000000000010000 | nor(**A,B**) | -49 |
| 6 | 0000000000100000 | and(**A,B**) | 10 |
| 7 | 0000000001000000 | xor(**A,B**) | A6 |
| 8 | 0000000010000000 | or(**A,B**) | 7E |
| 9 | 0000000100000000 | xnor(**A,B**) | 6E |

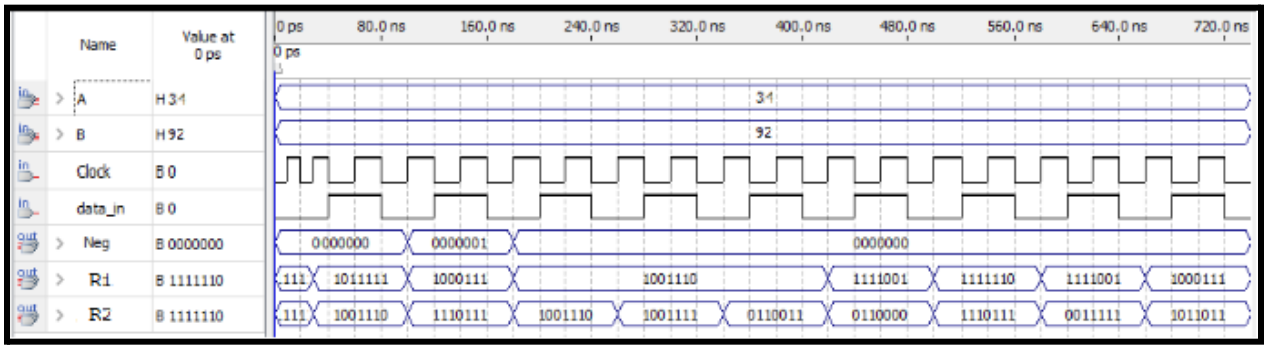
Table 1

For the first problem the objective was to make sure that the initial design of the ALU and GPP as a whole worked properly. This was done by making the ALU perform specific functions which were given in a table of core operations within the lab manual. The idea is that

every clock the ALU should be able to perform various calculations, which is possible by enabling the decoder first. This will dictate what signal is sent through the decoder and forwarded to the ALU determining what operation should be performed, then the results are outputted and displayed on the FPGA using the seven segment displays.




Waveform for ALU for Problem I




Completed Waveform Problem I

```
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(Clock, OP)
begin
        if(rising_edge(Clock)) THEN -- do the calculation @ positive edge of clock cycle
                case OP is
                        WHEN "1000000000000000" =>
                        -- do addition for reg1 and reg2
                                Result <= Reg1 + Reg2;
                                Neg <= '0';
                        WHEN "0100000000000000" =>
                        -- do subtraction
                                IF (Reg2(7 DOWNTO 0) > Reg1(7 DOWNTO 0)) THEN
                                        Result <=  (Reg1+NOT Reg2+1);
                                        Neg <= '1';
                                ELSE
                                        Result <= (Reg1 - Reg2);
                                        Neg <= '0';
                                END IF;
                        -- neg bit set if req
                        WHEN "0010000000000000" =>
                        -- do inverse
                                Result <= NOT A;
                                Neg<='0';
                        WHEN "0001000000000000" =>
                        -- do boolean NAND
                                Result <= A NAND B;
                                Neg<='0';
                        WHEN "0000100000000000" =>
                        -- do boolean NOR
                                Result <= A NOR B;
                                Neg<='0';
                        WHEN "0000010000000000" =>
                        -- do boolean AND
                                Result <= A AND B;
                                Neg<='0';
                        WHEN "0000001000000000" =>
                        -- do boolean XOR
                                Result <= A XOR B;
                                Neg<='0';
                        WHEN "0000000100000000" =>
                        -- do boolean OR
                                Result <= A OR B;
                          Neg<='0';
                        WHEN "0000000010000000" =>
                        -- do boolean XNOR
                                Result <= A XNOR B;
```

```
                                        Neg<='0';
                                WHEN OTHERS =>
                                -- do nothing, dont care
                                        Result <= "11111111";
                        end case;
                end if;
end process;
R1 <= Result(7 downto 4); --since the output seven segment can
R2 <= Result(3 downto 0); -- only 4-bits, split the 8-bit to two 4-bits
end calculation;
```

VHDL code for ALU

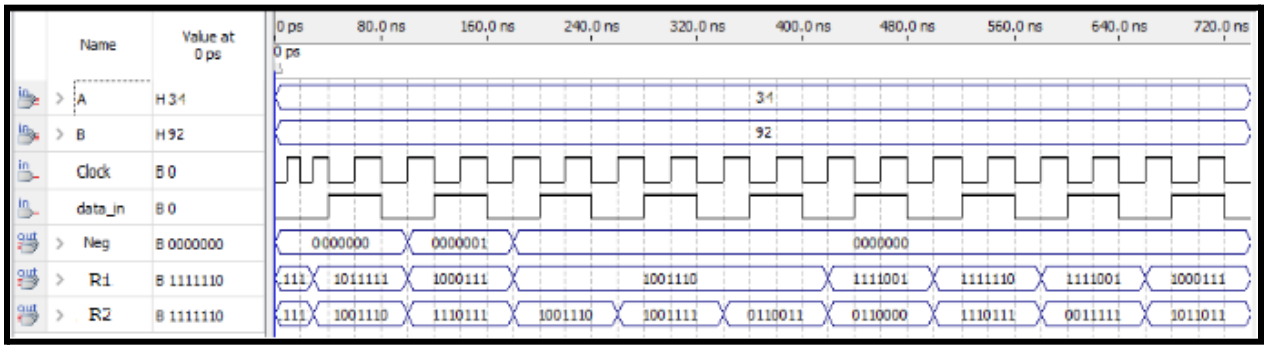
## ALU Problem II: Problem Set 2

For this problem, the circuit retains its original form from Problem I, the only change being made in the changes being made solely in the ALU itself since all that changes are the operations within the ALU. On top of the new functions, one of the challenges in this part of the lab was to implement a NULL operation. The operations are slightly more complex with more functions associated with them, such as shift right and max.

| **Function #** | **Microcode** | **Boolean Operation/Function** | **Answer** |
|---|---|---|---|
| 1 | 0000000000000001 | Shift A to right by two bits, input bit = 1 (**SHR**) | 82 |
| 2 | 0000000000000010 | sum(difference(**A, B**), **4**) | 3E |
| 3 | 0000000000000100 | max(**A, B**) | 5C |
| 4 | 0000000000001000 | Swap the upper 4 bits of **A** by the lower 4 bits of **B** | 42 |
| 5 | 0000000000010000 | sum(**A, 1**) | 35 |
| 6 | 0000000000100000 | and(**A,B**) | 31 |
| 7 | 0000000001000000 | Invert the upper four bits of **A** | D2 |

| | | | |
|---|---|---|---|
| 8 | 0000000010000000 | Rotate B to left by 3 bits (**ROL**) | E2 |
| 9 | 0000000100000000 | **NULL** | Nothing |

Table 2




Completed Waveform Problem II

```
Result <= Reg1 + "00000010";
                CurrentOp_internal <= "0000000000000001";

            when "0000000000000010" => --Subtraction Operation
                Result <= ("00" & Reg2(7 downto 2));
                CurrentOp_internal <= "0000000000000010";

            when "0000000000000100" => --Inversion Operation
                Result <= ("1111" & Reg1(7 downto 4));
                CurrentOp_internal <= "0000000000000100";

            when "0000000000001000" => --NAND Operation
                if (Reg1 < Reg2)
then Result <= Reg1;
elsif (Reg2 < Reg1)
then Result <= Reg2;
end if;
                CurrentOp_internal <= "0000000000001000";

            when "0000000000010000" => --NOR Operation
                Result <= Reg1(1 downto 0) & Reg1(7 downto 2);
                CurrentOp_internal <= "0000000000010000";

            when "0000000000100000" => --AND Operation
                Result <= (Reg2(0) & Reg2(1) & Reg2(2) & Reg2(3) & Reg2(4) & Reg2(5) & Reg2(6) &
```

```
Reg2(7));
                CurrentOp_internal <= "0000000000100000";

            when "0000000001000000" => --OR Operation
                Result <= Reg1 XOR Reg2;
                CurrentOp_internal <= "0000000001000000";

            when "0000000010000000" => --XOR Operation
                Result <= (Reg1 + Reg2) - "00000100";
                CurrentOp_internal <= "0000000010000000";

            when "0000000100000000" => --XNOR Operation
                Result <= ("11111111");
                CurrentOp_internal <= "0000000100000000";

            when others => -- Rest is negative
                Result <= (others => '-');
                CurrentOp_internal <= (others => '-');
        end case;
    end if;
  end process;

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

VHDL Code for Problem 2's ALU

## ALU Problem III: Problem Set 3

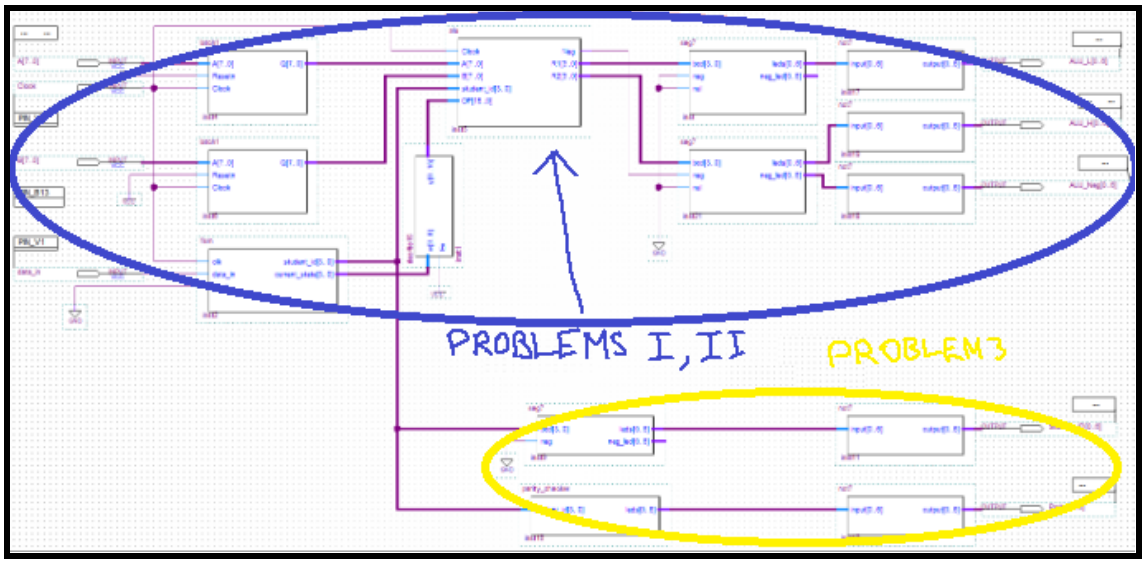
For this last problem set the circuit remains the same however the only change that was made within the seven segment. The problem requires the seven segments to display each microcode instruction, display 'y' if the FSM output (student_id) is odd and 'n' otherwise. This was completed by making cases for every 4-bit combination of the student number where if the number is odd.

| Student_ID in 4-bit binary | Even or Odd | 'y' or 'n' | 'y' or 'n' in Seven Segment Display |
|---|---|---|---|
| 0101 | Odd | y | 0111011 |
| 0000 | Even | n | 1110110 |
| 0001 | Odd | y | 0111011 |
| 0001 | Odd | y | 0111011 |
| 0111 | Odd | y | 0111011 |
| 0011 | Odd | y | 0111011 |
| 0100 | Even | n | 1110110 |
| 1001 | Odd | y | 1110110 |
| 0010 | Even | n | 1110110 |

Table 3

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

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

ARCHITECTURE Behavior OF ssegmodified IS
BEGIN
PROCESS (bcd)
BEGIN
if negative = '1' then
ledss <= "1111110";
else
ledss <= "1111111";

END if;

CASE bcd IS   -- abcdefg
WHEN "0000" => leds <= "1101010";
WHEN "0001" => leds <= "1000100";
WHEN "0010" => leds <= "1101010";
WHEN "0011" => leds <= "1000100";
WHEN "0100" => leds <= "1101010";
WHEN "0101" => leds <= "1000100";
WHEN "0110" => leds <= "1101010";
WHEN "0111" => leds <= "1000100";
WHEN "1000" => leds <= "1101010";
WHEN "1001" => leds <= "1000100";
WHEN "1010" => leds <= "1101010";
WHEN "1011" => leds <= "1000100";
WHEN "1100" => leds <= "1101010";
WHEN "1101" => leds <= "1000100";
WHEN "1110" => leds <= "1101010";
WHEN "1111" => leds <= "1000100";
WHEN OTHERS => leds <= "-------";
END CASE;
END PROCESS;
END Behavior;
```

VHDL Code for Problem 3 SSEG




## Conclusion

In conclusion, by using components formed by skills and knowledge obtained in previous labs was used to create a general purpose processor, a highly important component in a computer. In this lab the GPP was tested and had specific components altered and reprogrammed to fit the requirements of the user. This reflects its real world application as many processors are altered to fit the needs of various customers worldwide, whether it’s complex calculation or simple math the GPP is an essential component in today's technology.