Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Digital Systems
Course Number:	COE328 - 13
Semester/Year (e.g.F2016)	F2022

Instructor:	MD Shazzat Hossain
T.A/G.A	Luzalen Marcos

Assignment/Lab Number:	Lab 6
Assignment/Lab Title:	Design of a Simple General-Purpose Processor

Submission Date:	Saturday, December 3, 2022
Due Date:	Saturday, December 3, 2022

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Sooriyapperuma	Lathika	500904706	13	

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Contents

Introduction

The “Simple General-Purpose Processor” created for lab 6 is a digital circuit which can be used to calculate and output the result of various functions. Inputted are two 8-bit Booleans, and the 8-bit result of the calculation is displayed in hexadecimal on a 7-segment LED. To get from the inputs to the output, the unit can be thought to be comprised of 4 distinct parts.

The first is the storage, which is comprised of 2 registers, one for each of the inputs. Each register has the responsibility of storing an 8-bit value temporarily, before passing it on to the ALU. The 2nd part is the control unit, which decides which of the 9 functions the ALU is capable of should be done. It is made up of two parts: the FSM, which cycles through 9 states and outputs a corresponding 4-bit representation of the state number, and it has the decoder, which takes the 4-bit output and translates it into 16-bit microcode which the ALU can interpret for operation selection.

Third is the ALU itself, which perform whichever of it’s 9 functions it is meant to, according to the inputted microcode from the control unit. Its output is an 8-bit value and a negative bit, which is fed to the fourth and final part, namely the “Seven Segment Display unit”, which translates the 8-bit Boolean value into two 7-bit Booleans corresponding to a visual representation of a 2-hextet hexadecimal value on 7-segment LED, along with a negative sign for when it is applicable.

Components

Latch1 & Latch2

Latch1 and Latch2 are identical and act as registers which store the 8-bit Boolean value read from its input on a rising edge, and outputs the same value on the following rising edge. When the reset input is high, the value stored in the register is cleared and the output will be 0 for all its bits. When the reset becomes low again, it will be able to store the value and output it again.

The latch was made using VHDL code and is equivalent to 8 gated SR-Latches (1 for each bit), that all have the same Clock and Reset signal. It has the following truth table:

Clock	**Reset**	**Set**	**Q_n**	**Q_{n+1} (Output)**
0	X	X	X	No Change
1	0	0	X	No Change
1	0	1	X	1
1	1	0	X	0
1	1	1	X	X (Invalid)

Table 1: Truth Table for Gated SR Latch.

A gated SR-Latch has the following circuit diagram:

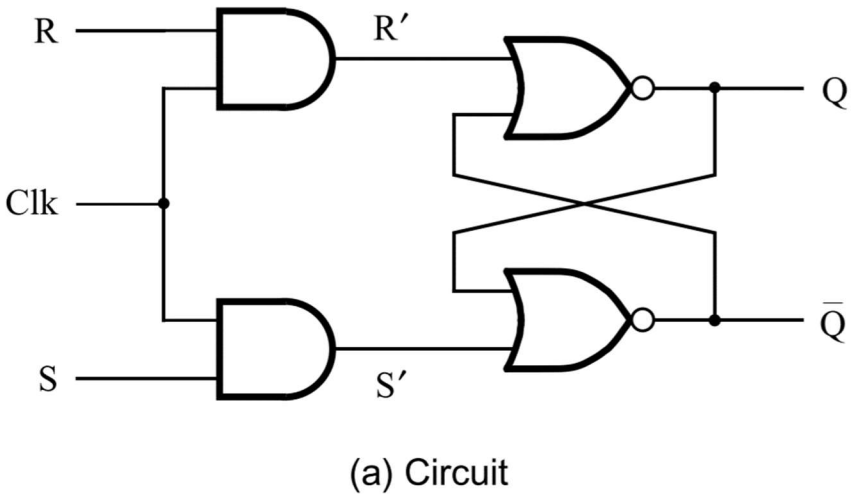


Figure 1: Gated SR-Latch Logic Diagram [1].

The code used to create the symbol file for the latches is shown below along with an image of the block schematic itself:

SimpleGPP.bdf latch1.vhd

```
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY latch1 IS
5     PORT (
6           A : IN STD_LOGIC_VECTOR(7 DOWNTO 0); --8 bit A input
7           Resetn,Clock : IN STD_LOGIC;  --1 bit clock input, 1 bit reset input bit
8           Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
9           );
10 END latch1;
11
12 ARCHITECTURE Behavior OF latch1 IS
13 BEGIN
14    PROCESS(Resetn,Clock) --Process takes reset and clock as inputs
15    BEGIN
16       IF Resetn = '1' THEN --when reset input is '1' the latches do not operate
17          Q<= "00000000";
18       ELSIF Clock'EVENT AND Clock = '1' THEN --level sensitive based on clock
19          Q<= A;
20       END IF;
21    END PROCESS;
22
23 END Behavior;
```

0% 00:00:00

Search ENG US 5:50 PM 2022-12-03

Figure 2: VHDL Code Screenshot for the Latch device.

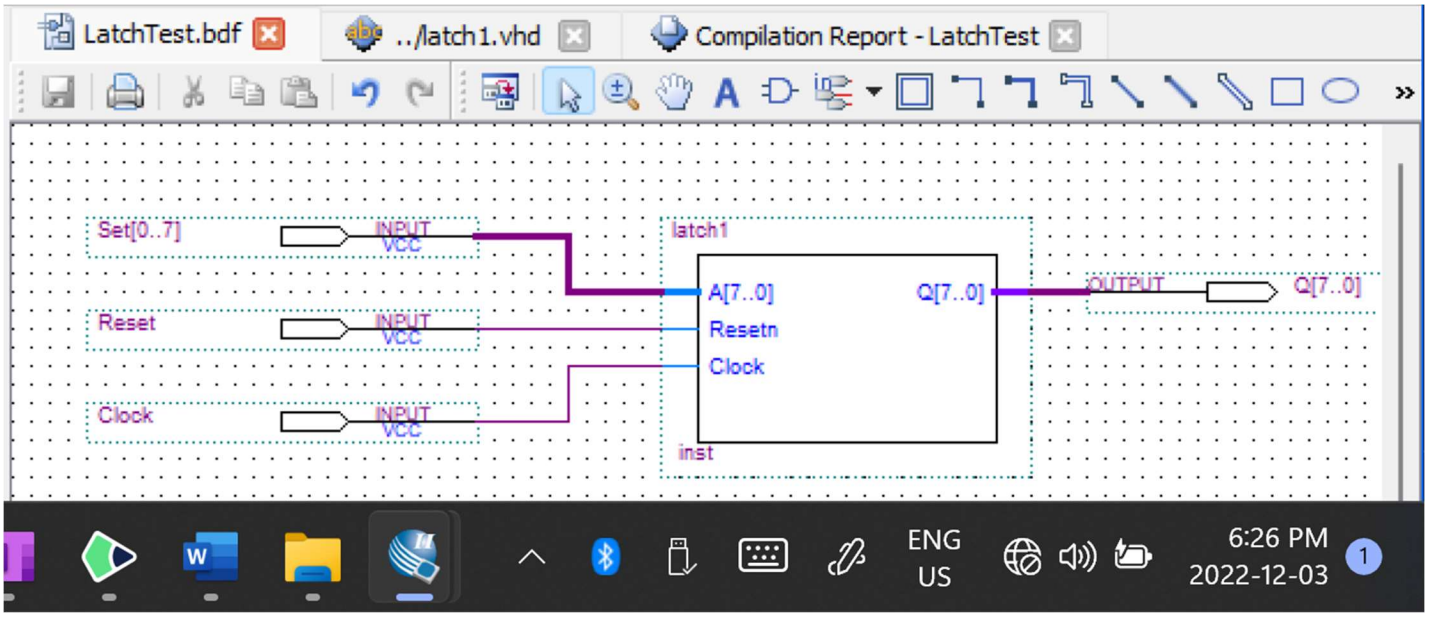


Figure 3: Screenshot of Block Diagram of Latch.

Using the block diagram, the latch is tested and is proven to work using the following waveform:

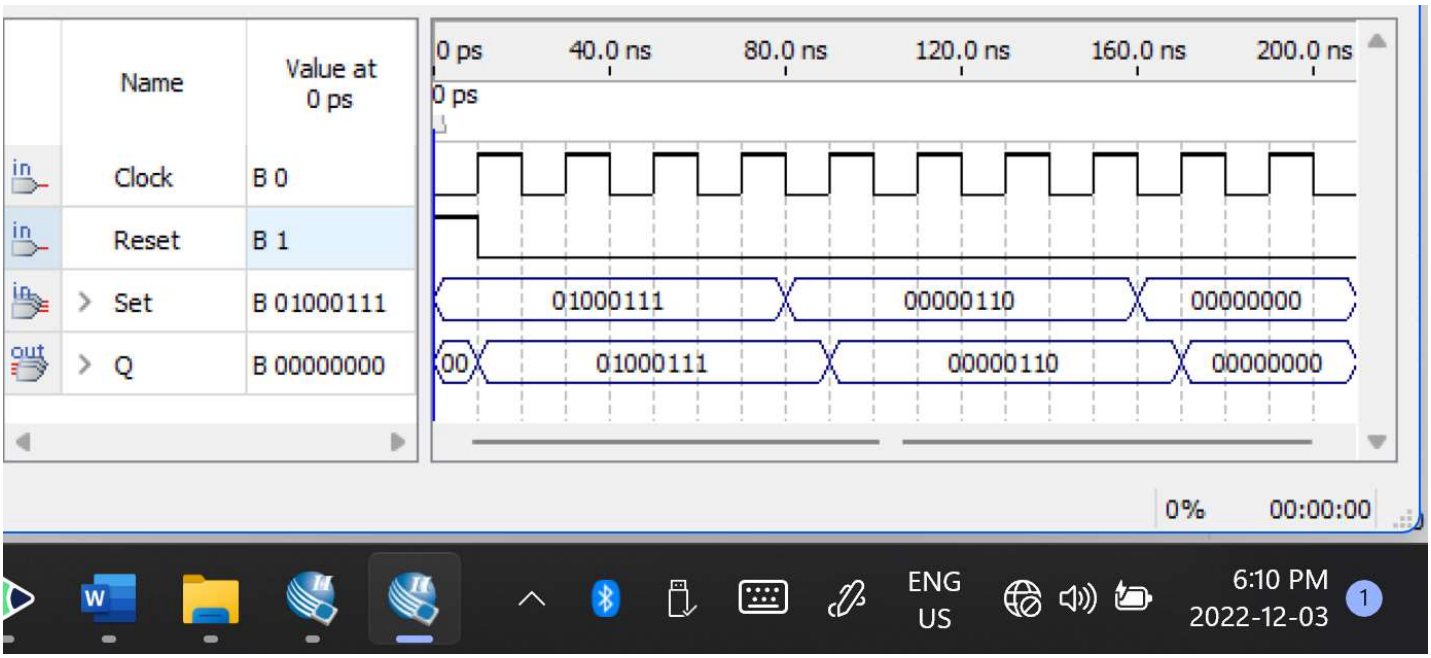


Figure 4: Waveform of “8-Input” Latch.

FSM

The FSM or Finite State Machine is the main part of the Control Unit and is responsible for selecting which of the 9 operations the ASU is supposed to do, as well as the outputting digits of the student number consecutively. It does so by cycling through 9 states in a specific order defined by a state diagram. Each rising edge of the clock input forces the next state, except when data_in is low, in which case the state remains unchanged even when a rising edge is met. Also, there is a reset input which will cause the state to go back to the first state, s0, regardless of what is happening data_in or the clock and will only advance to the next state once reset is low and a clock rising edge occurs w/ the data_in set to high.

The FSM was coded in VHDL according to the state diagram and state table below:

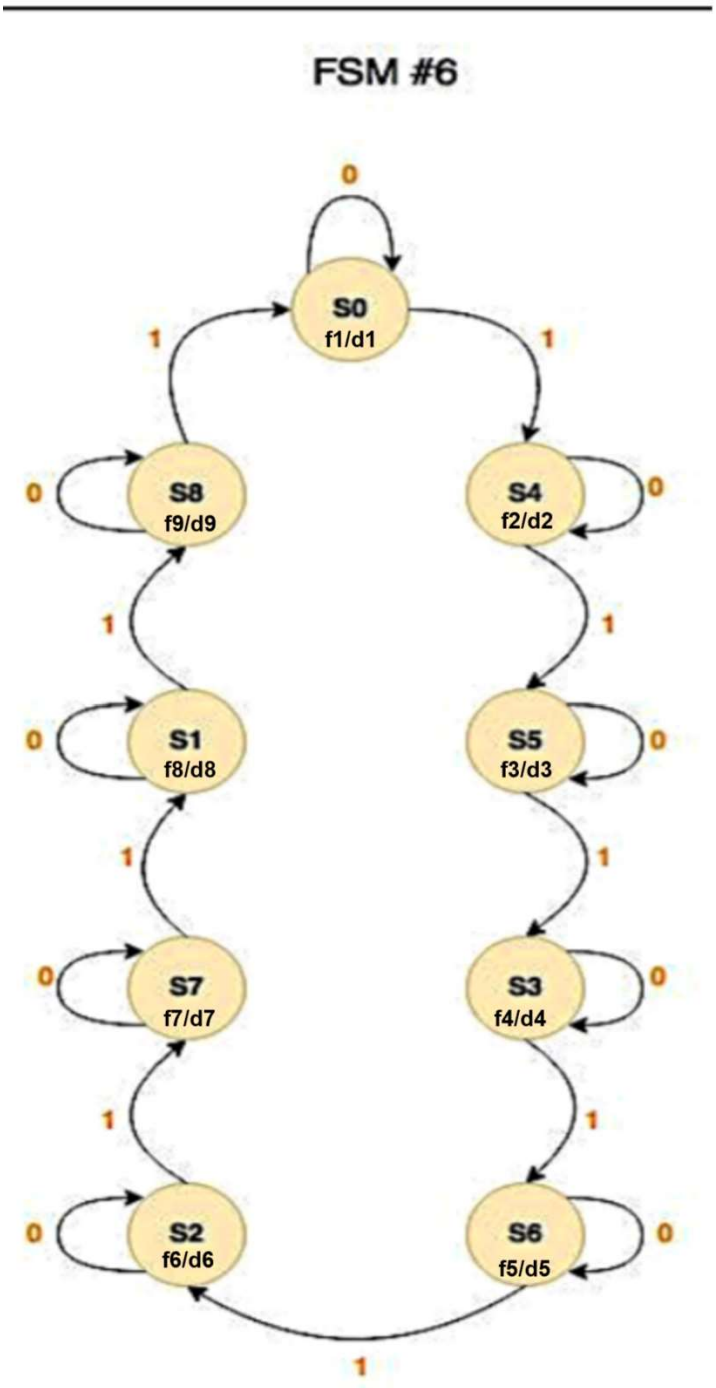


Figure 5: State Diagram for FSM[2].

data_in	reset	Clock	Previous State	Next State
0	0	x	S0	S0
0	0	x	S1	S1
0	0	x	S2	S2
0	0	x	S3	S3
0	0	x	S4	S4
0	0	x	S5	S5
0	0	x	S6	S6
0	0	x	S7	S7
0	0	x	S8	S8
x	1	x	x	S0
1	0	0	S0	S0
1	0	1	S0	S4
1	0	0	S1	S1
1	0	1	S1	S8
1	0	0	S2	S2
1	0	1	S2	S7
1	0	0	S3	S3
1	0	1	S3	S6
1	0	0	S4	S4
1	0	1	S4	S5
1	0	0	S5	S5
1	0	1	S5	S3
1	0	0	S6	S6
1	0	1	S6	S2
1	0	0	S7	S7
1	0	1	S7	S1
1	0	0	S8	S8
1	0	1	S8	S0

Table 2: Truth Table for

state	current_state	state in decimal	current_state	ID digit in decimal
S0	0000	0 (f1)	0101	5 (d1)
S1	0001	1 (f2)	0000	0 (d2)
S2	0010	2 (f3)	0000	0 (d3)
S3	0011	3 (f4)	1001	9 (d4)
S4	0100	4 (f5)	0000	0 (d5)
S5	0101	5 (f6)	0100	4 (d6)
S6	0110	6 (f7)	0111	7 (d7)
S7	0111	7 (f8)	0000	0 (d8)
S8	1000	8 (f9)	0110	6 (d9)

Table 3: State Table for FSM.

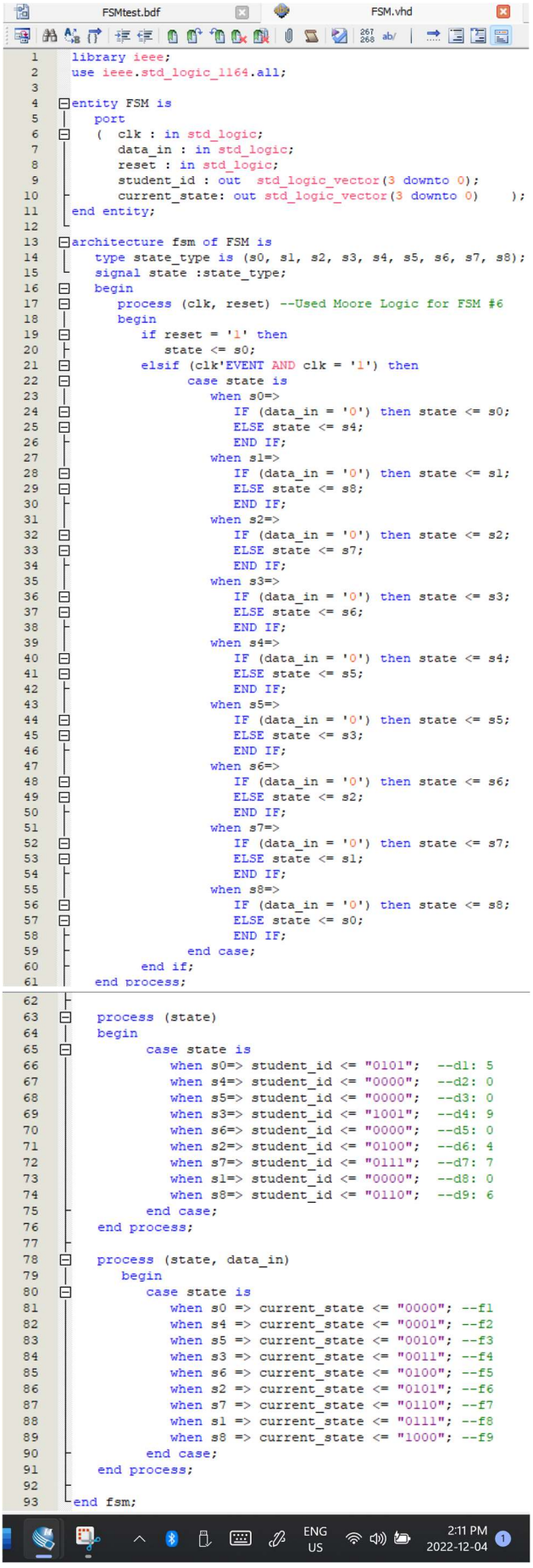


Figure 6: VHDL code for FSM.

The block symbol for the FSM is pictured below:

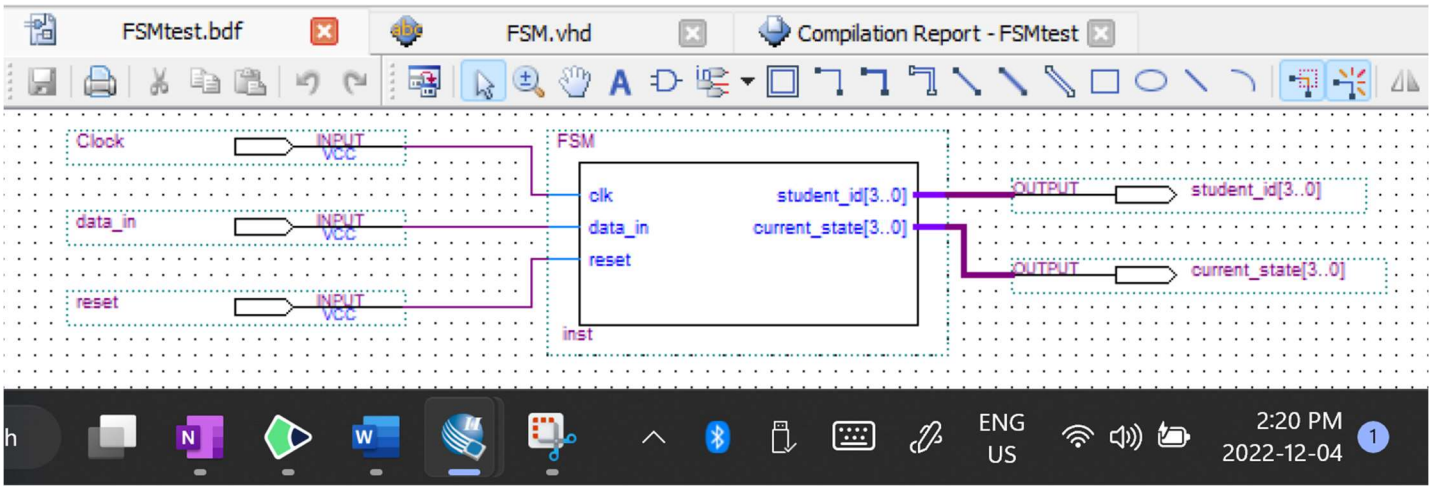


Figure 7: FSM Block Symbol.

Finally, the waveform below verifies the proper function of the FSM when compared to the truth and state tables previously shown.

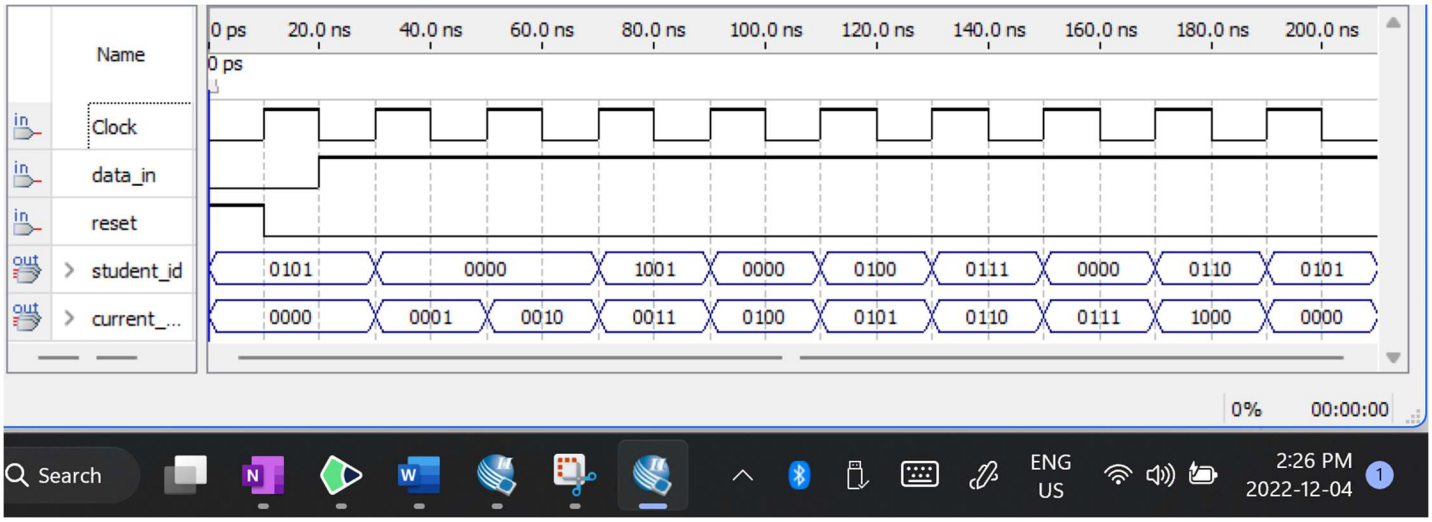


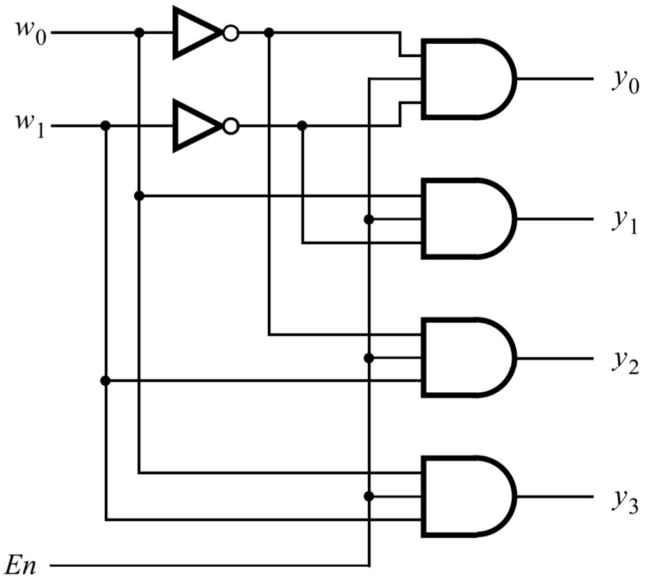
Figure 8: Simulation waveform for FSM.

4:16 Decoder

The decoder has the responsibility of translating the FSM's current_state signal, which is a 4-bit Boolean unsigned integer from 0 to 8, to 16-bit microcode for the "OP" output. This output fed to the ALU which can interpret it to choose which of it's 9 functions to preform on its inputs. The 4:16 decoder was constructed by using 2 3:8 decoders which itself was constructed from two 2:4 decoders. The 2:4 decoder was created using VHDL code. A 2:4 Decoder has the following circuit diagram and truth table:

En	w_1	w_1	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

Table 4: Truth table for 2:4 Decoder.



(c) Logic circuit

Figure 6.16 A 2-to-4 decoder.

Figure 9: Logic Diagram of 2:4 decoder [3].

The process of how a 2:4 decoder gives us a 4:16 decoder is shown visually below:

dec2to4.vhd | dec2to4.bsf

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4   ENTITY dec2to4 IS
5       PORT ( w : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
6              En : IN STD_LOGIC ;
7              y : OUT STD_LOGIC_VECTOR(0 TO 3) ) ;
8   END dec2to4 ;
9
10  ARCHITECTURE Behavior OF dec2to4 IS
11      SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0);
12  BEGIN
13      Enw <=En & w ;
14      WITH Enw SELECT
15          y <="1000" WHEN "100",
16              "0100" WHEN "101",
17              "0010" WHEN "110",
18              "0001" WHEN "111",
19              "0000" WHEN OTHERS;
20  END Behavior ;
```

ENG US 3:24 PM 2022-12-04

Figure 9: VHDL code for a 2:4 Decoder

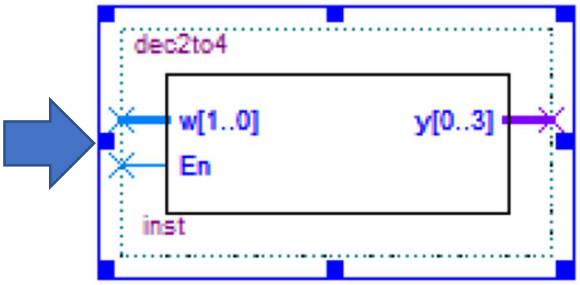


Figure 10: Block Symbol for 2:4 Decoder

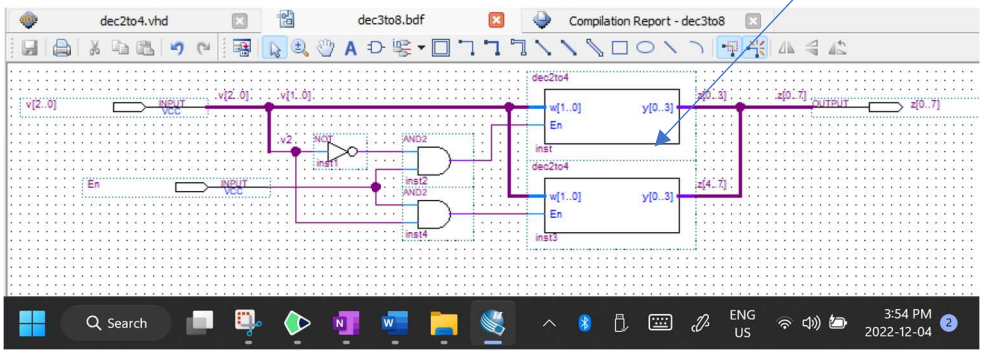


Figure 11: Block Diagram for 3:8 Decoder

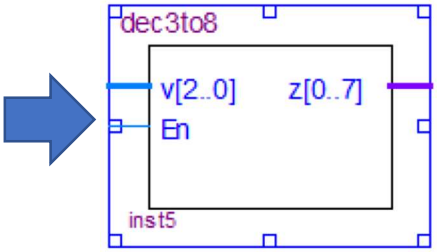


Figure 12: Block Symbol for 3:8 Decoder

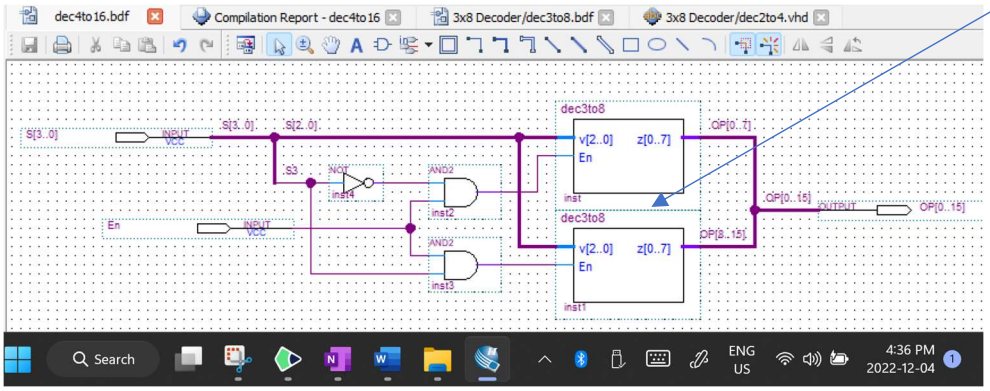


Figure 13: Block Diagram for 4:16 Decoder

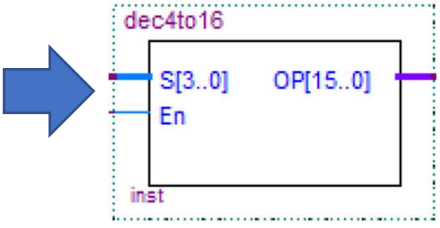


Figure 14: Block Symbol for 4:16 Decoder

Finally, we have the waveform file, which verifies the functionality of the 4:16 decoder.

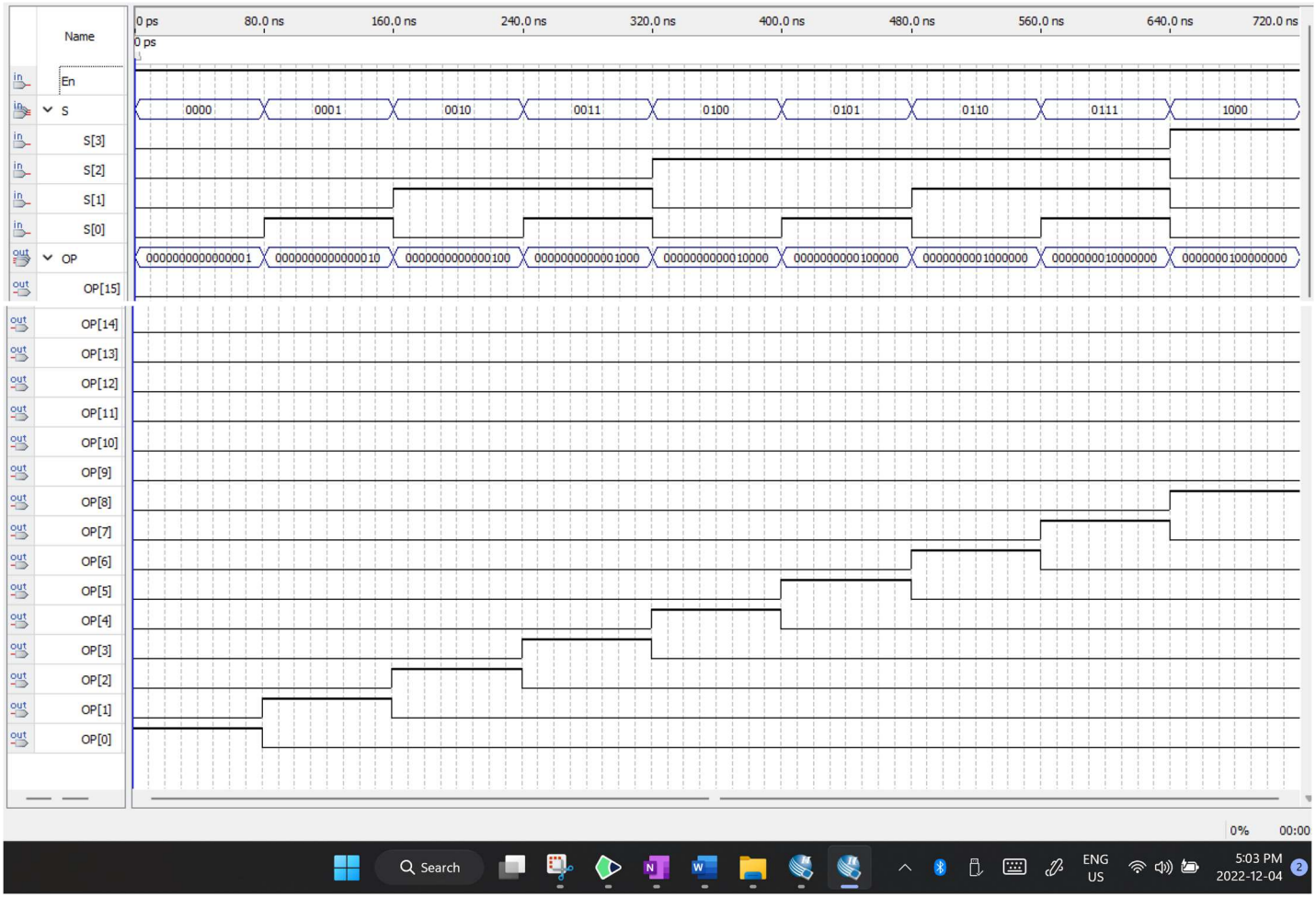


Figure 14: Waveform for 4:16 Decoder

ALU_1

ALU_1 is the Arithmetic/Logic unit core for Problem set 1. On each rising clock signal, it performs the problem set's functions on the inputs given to it by each of the 2 registers. It outputs the solution on two 4-bit buses for each of the 7-segment LEDs, along with a negative bit for when a function outputs a negative value. It also reads each of the Student ID digits sequentially from the FSM, but it is not used in this first problem set.

Those functions, the generated microcode and the expected outputs are shown below for the student number inputs, which is found by taking the last 4 digits of the student number and turning them into two hexadecimal numbers [4]. In this case, the student number is 500904706, so input A is $(47)_{16}$ and input B is $(06)_{16}$.

Function	**F#**	**Microcode**	sign	Out(bin)	Out (>Hex)	**Hex2 (disp)**	**Hex1 (disp)**	**Hex0 (disp)**
A+B	**1**	**0000000000000001**	+	0100 1101	4 D	**0000000**	**0110011**	**0111101**
A-B	**2**	**0000000000000010**	+	0100 0001	4 1	**0000000**	**0110011**	**0110000**
A'	**3**	**0000000000000100**	+	1011 1000	B 8	**0000000**	**0011111**	**1111111**
(A AND B)'	**4**	**0000000000001000**	+	1111 1001	F 9	**0000000**	**1000111**	**1110011**
(A OR B)'	**5**	**0000000000010000**	+	1011 1000	B 8	**0000000**	**0011111**	**1111111**
A AND B	**6**	**0000000000100000**	+	0000 0110	0 6	**0000000**	**1111110**	**1011111**
A XOR B	**7**	**0000000001000000**	+	0100 0001	4 1	**0000000**	**0110011**	**0110000**
A OR B	**8**	**0000000010000000**	+	0100 0111	4 7	**0000000**	**0110011**	**1110000**
(A XOR B)'	**9**	**0000000100000000**	+	1011 1110	B E	**0000000**	**0011111**	**1001111**

Table 5: Functions, Microcode & Expected output for ALU_1.

The VHDL code for ALU_1 and the block diagram of the entire processor (since it is different for each problem set) is displayed below:

SimpleGPP.bdf ALUt1.vhd

```
1  Library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5
6  entity ALUt1 is
7     port(
8     Clock : in std_logic; --input clock signal
9     A,B : in unsigned(7 downto 0); --8-bit inputs from latches A and B
10    student_id: in unsigned(3 downto 0); --4 bit studost id from FSM
11    OP : in unsigned(15 downto 0); --16-bit selector for Operation from Decoder
12    Neg: out std_logic; --is the result negative? Set -ve bit output
13    R1: out unsigned(3 downto 0); -- lower 4-bits of 8-bit Result Output
14    R2: out unsigned(3 downto 0)  -- higher 4-bits of 8-bit Result Output
15    );
16 end ALUt1;
17
18 architecture calculation of ALUt1 is --temporary signal declarations.
19    signal Reg1, Reg2, Result : unsigned(7 downto 0):= (others => '0');
20    signal Reg4: unsigned(7 downto 0);
21    begin
22       Reg1 <= A; --temporarily store A in Reg1 local variable
23       Reg2 <= B; --temporarily store B in Reg2 local variable
24
25    process(Clock, OP)
26    begin
27       if(rising_edge(Clock)) THEN --Positive edge signal
28          Neg <= '0';
29
30          case OP is
31             WHEN "0000000000000001" => Result <= Reg1 + Reg2;
32             WHEN "0000000000000010" =>
33                if (Reg2<Reg1) then Result <= (Reg1 -Reg2);
34                Neg <= '0';
35                else Result <= (Reg1 +(NOT Reg2 + 1));
36                neg <= '1';
37                end if;
38             WHEN "0000000000000100" => Result <= NOT Reg1;
39             WHEN "0000000000001000" => Result <= (NOT (Reg1 AND Reg2));
40             WHEN "0000000000010000" => Result <= (NOT (Reg1 OR Reg2));
41             WHEN "0000000000100000" => Result <= Reg1 AND Reg2;
42             WHEN "0000000001000000" => Result <= Reg1 XOR Reg2;
43             WHEN "0000000010000000" => Result <= Reg1 OR Reg2;
44             WHEN "0000000100000000" => Result <= Reg1 XNOR Reg2;
45             WHEN OTHERS => Result<= "--------";
46          end case;
47       end if;
48    end process;
49
50    R1 <= Result(3 downto 0); --Since the output seven segments can
51    R2 <= Result(7 downto 4); -- only 4-bits, split the 8-bit to two 4-bits.
52
53 end calculation;
```

Figure 15: VHDL code for ALU_1's core

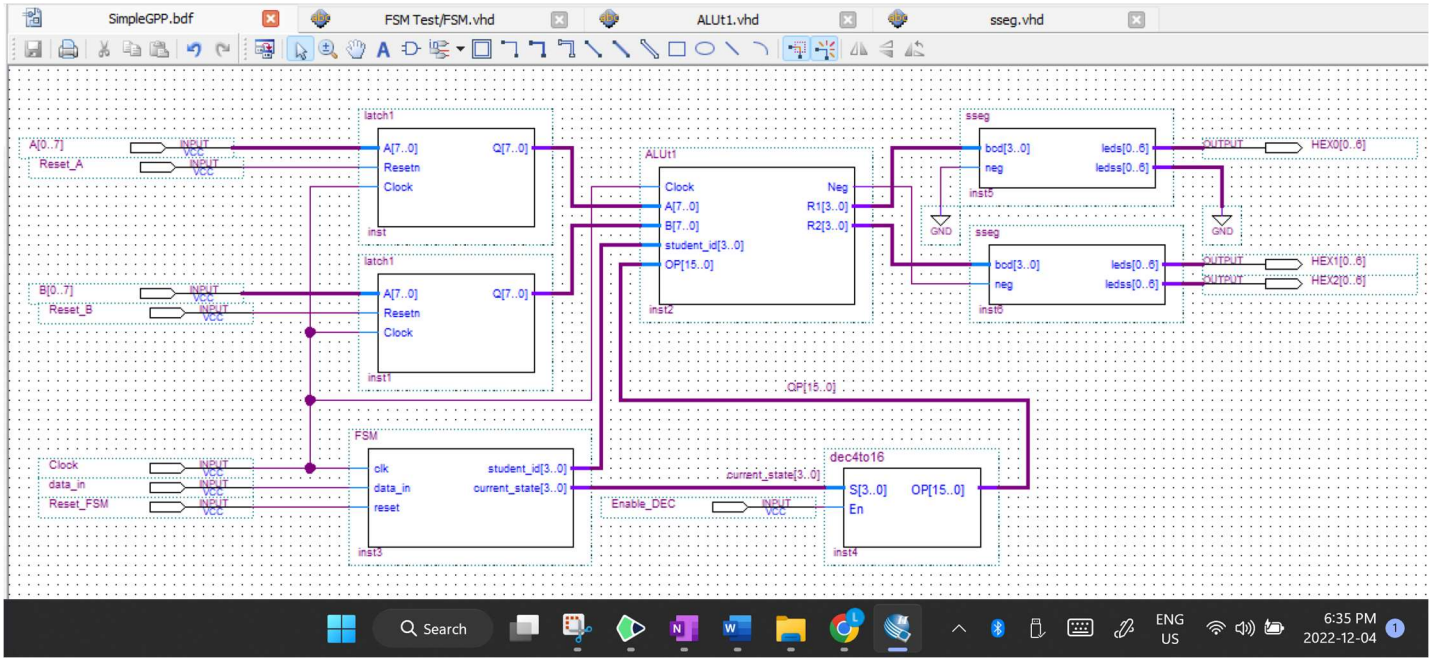


Figure 16: Block Diagram for ALU_1.

Finally, the functionality of ALU_1 is verified by comparing its outputs to the expected outputs. Note that until 30ms where the first rising edge is met while data_in is high, there is no output and the 7-segs display their default position of 0. After 30ms is where the cycle starts, and one can observe the outputs of the processor.

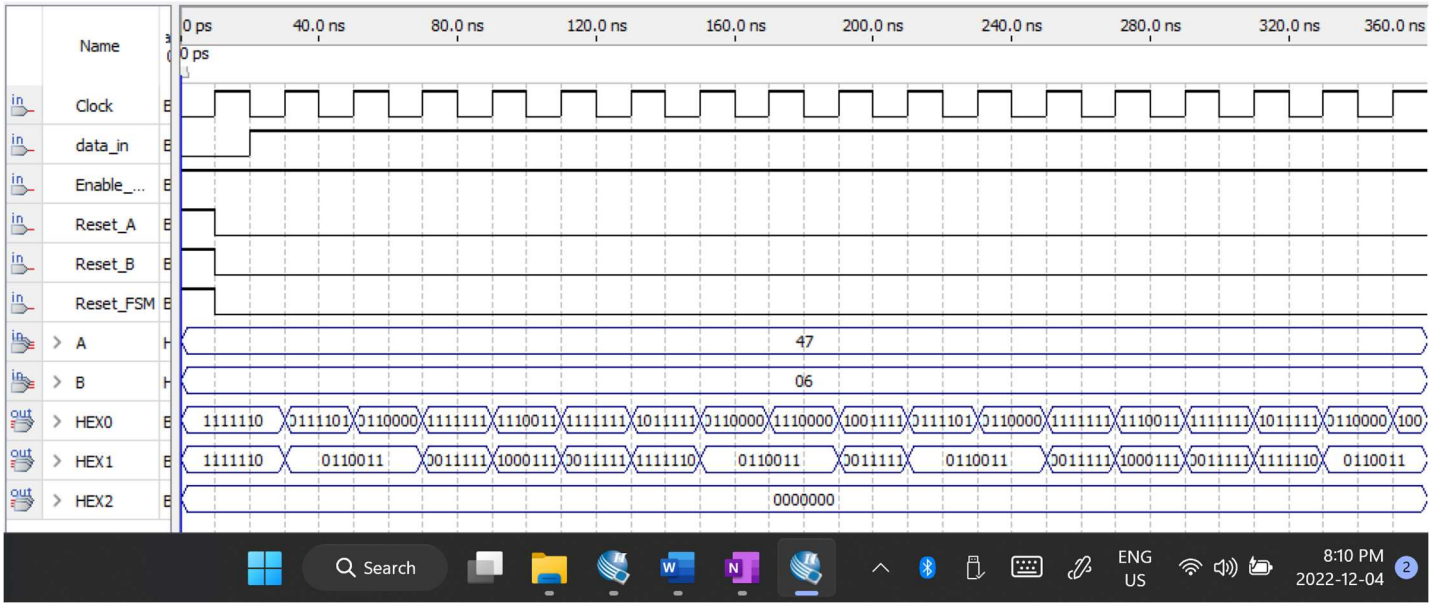


Figure 17: Waveform for ALU_1

ALU_2

ALU_2 has its set of functions. Those functions, the corresponding microcode and the expected outputs are shown:

Function	#	Microcode	sign	Out(bin)	Out (->Hex)	Hex2 (disp)	Hex1 (disp)	Hex0 (disp)
A+2	1	0000000000000001	+	0100 1001	4 9	0000000	0110011	1110011
B SHR 2, In=0	2	0000000000000010	+	0000 0001	0 1	0000000	1111110	0110000
A SHR 4, In=1	3	0000000000000100	+	1111 0100	E 4	0000000	1001111	0110011
Min(A,B)	4	0000000000001000	+	0100 0111	4 7	0000000	0110011	1110000
A ROR 2	5	0000000000010000	+	1101 0001	D 1	0000000	0111101	0110000
B Inverse Bit Order	6	0000000000100000	+	0110 0000	6 0	0000000	1011111	1111110
A XOR B	7	0000000001000000	+	0100 0001	4 1	0000000	0110011	0110000
(A+B) - 4	8	0000000010000000	+	0100 1001	4 9	0000000	0110011	1110011
A OR B	9	0000000100000000	+	0100 0111	4 7	0000000	0110011	1110000

Table 6: Functions, Microcode & Expected output for ALU_2.

The block diagram for ALU_2 is the same as it was for ALU_1, apart from the naming of the ALU core, so there is no need to show the screenshot again.

There is a difference in the code, but strictly for the process which contains the calculations. A screenshot of that code is shown below.

```
26  process(Clock, OP)
27  begin
28     if(rising_edge(Clock)) THEN --Do the calculation @ positive edge of clock
29        case OP is
30        Neg <= '0';
31           WHEN "0000000000000001" =>
32              --Func #1: Increment A by 2
33              Result <= Reg1 + 2;
34           WHEN "0000000000000010" =>
35              --Func #2: Shift B to right by two bits, input bit = 0 (SHR)
36              Result <= shift_right(Reg2, 2);
37           WHEN "0000000000000100" =>
38              --Func #3: Shift A to right by four bits, input bit = 1 (SHR)
39              Reg4 <= shift_right(Reg1, 4);
40              Reg4(7) <= '1';
41              Reg4(6) <= '1';
42              Reg4(5) <= '1';
43              Reg4(4) <= '1';
44              Result <= Reg4;
45           WHEN "0000000000001000" =>
46              --Func #4: Find smaller value b/w A and B, output it ( Min(A,B) )
47              if (Reg2<Reg1) then Reg4 <= Reg1;
48              else Reg4 <=Reg2;
49              end if;
50              Result <= Reg4;
51           WHEN "0000000000010000" =>
52              --Func #5: Rotate A to right by two bits (ROR)
53              Result <= (Reg1 ROR 2);
54           WHEN "0000000000100000" =>
55              --Func #6: Invert the bit-significance order of B
56              Result(7) <= Reg2(0);
57              Result(6) <= Reg2(1);
58              Result(5) <= Reg2(2);
59              Result(4) <= Reg2(3);
60              Result(3) <= Reg2(4);
61              Result(2) <= Reg2(5);
62              Result(1) <= Reg2(6);
63              Result(0) <= Reg2(7);
64           WHEN "0000000001000000" =>
65              --Func #7: Produce the result of XORing A and B
66              Result <= Reg1 XOR Reg2;
67           WHEN "0000000010000000" =>
68              --Func #8: Produce the summation of A and B, then decrease it by
69              Result <= (Reg1 - Reg2) - 4;
70           WHEN "0000000100000000" =>
71              --Func #9: Produce all high bits on the output
72              Reg4 <= Reg1 OR Reg2;
73              Result <= Reg4;
74           WHEN OTHERS =>
75              --There are no o/ functions
76              Result<= "--------";
77        end case;
78     end if;
79  end process;
```

Figure 18: Relavant Part of VHDL code for ALU_2's core

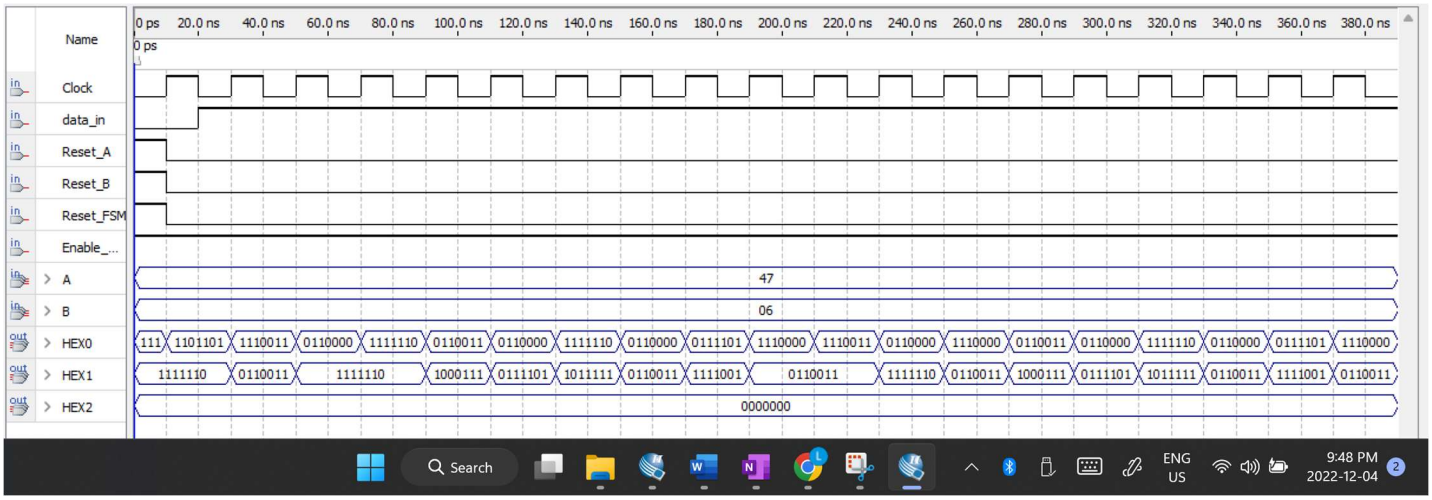


Figure 19: Waveform for ALU_2

ALU_3

ALU_3, unlike the previous ALUs, take advantage of the Student_ID out from the FSM. The ALU core reads the 4-bit binary value and decides whether it is even. If it is, it outputs and 1 for yes and if not, it outputs 0 for no over the 4-bit R1 bus. The other bus is no longer needed and is therefore grounded. The signal is interpreted by the new 7-segment LED decoder which has been modified to be able output y (R1=0001) and n (R1=0000) for yes and no respectively. The digits of the student ID and the expected outputs are shown:

d#	Digit	Output (bin)	Even?	Hex0 (disp)
1	5	0101	no	0010101
2	0	0000	yes	0111011
3	0	0000	yes	0111011
4	9	1001	no	0010101
5	0	0000	yes	0111011
6	4	0100	yes	0111011
7	7	0111	no	0010101
8	0	0000	yes	0111011
9	6	0110	yes	0111011

Table 7: ALU_3 Expected Outputs (in order)

Both the ALU core and 7-segment LED decoder have modified code for problem set 3. Both are shown below:

gpp3b.bdf | Compilation Report - gpp3b | ../ALU3b.vhd*

```
1   Library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.STD_LOGIC_UNSIGNED.ALL;
4   use IEEE.NUMERIC_STD.ALL;
5
6   entity ALU3b is
7   port( Clock : in std_logic; --input clock signal
8         A,B : in unsigned(7 downto 0); --8-bit inputs from lat
9         student_id: in unsigned(3 downto 0); --4 bit student i
10        OP : in unsigned(15 downto 0); --16-bit OP selector fi
11        Neg: out std_logic; --is the result negative? Set -ve
12        R1: out unsigned(3 downto 0); --lower 4-bits of 8-bit
13        R2: out unsigned(3 downto 0)  --higher 4-bits of 8-bit
14      );
15  end ALU3b;
16
17  architecture calculation of ALU3b is
18     signal Result : unsigned(7 downto 0):= (others => '0');
19     begin
20        process(Clock, OP)
21        begin
22           if(rising_edge(Clock)) THEN
23              case OP is
24                 WHEN OTHERS =>
25                    --There are no functions
26                    Result<= "--------";
27                    Neg<='0';
28              end case;
29           end if;
30        end process;
31
32     Result(3 downto 0) <= student_id(3 downto 0);
33     R1 <= Result(3 downto 0);
34     R2 <= "0000";
35
36  end calculation;
```

Ln 7 Col 50 VHDL File 100% 00:00:03

ENG US 10:33 PM 2022-12-04

Figure 20: VHDL code for ALU_3's core

Note: SSEG code was shown only for this iteration of it since it is different from the one created and reported on in previous labs.

gpp3b.bdf | Compilation Report - gpp3b | ../ALU3b.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4   ENTITY sseg3b IS
5
6   PORT (bcd: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
7      neg: IN STD_LOGIC;
8      leds: OUT STD_LOGIC_VECTOR(0 TO 6);
9      ledss: OUT STD_LOGIC_VECTOR(0 TO 6)
10     );
11  END sseg3b;
12
13  ARCHITECTURE Behavior OF sseg3b IS
14     BEGIN
15        PROCESS(bcd)
16        BEGIN
17           CASE bcd IS
18              WHEN "0000" => leds <= "0010101"; --n
19              WHEN "0001" => leds <= "0111011"; --y
20              WHEN OTHERS => leds <= "0001000"; --_
21           END CASE;
22        END PROCESS;
23
24        PROCESS (neg)
25           BEGIN
26              IF (neg = '1') THEN ledss <= "0000001";
27              ELSE ledss <= "0000000";
28              END IF;
29        END PROCESS;
30
31  END Behavior;
32
```

ENG US 11:00 PM 2022-12-04

Figure 21: VHDL code for modified 7-segment decoder

Because the of the omission of all but one sseg device, the block diagram for the problem set 3 processor is considerably different and is therefore shown below:

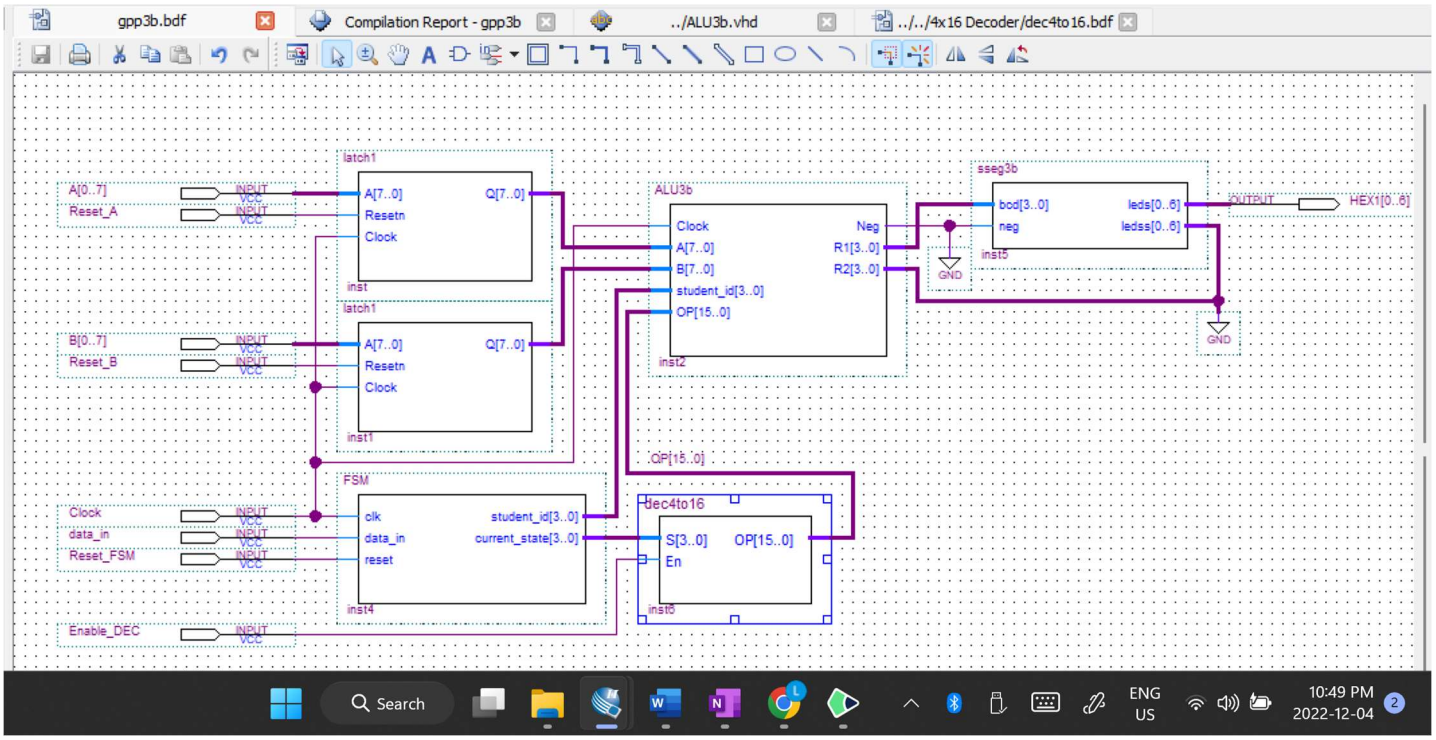


Figure 20: Block Diagram for ALU_3

Finally, the waveform illustrates that the ALU_3 processor is functioning correctly as the outputs match what is expected.

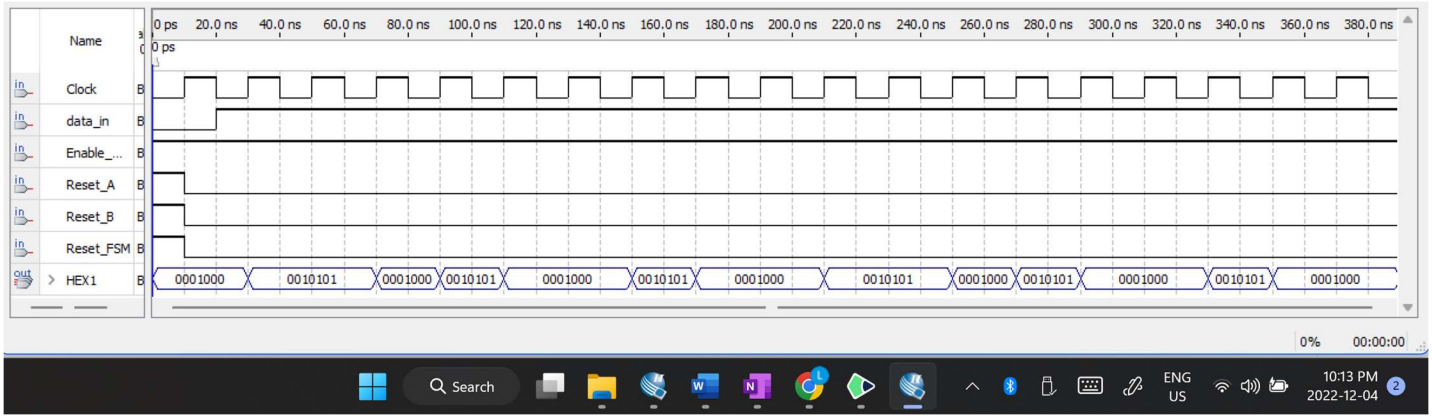


Figure 21: Waveform for ALU_3

Conclusion

This concludes the laboratory 6 formal report. As seen from the outputs of the processor for each of the problem sets, the simple general-purpose processor made in Quartus using VHDL code and block diagrams is an effective device with a range of utilities.

References

[1] S. D. Brown and Z. G. Vranesic, "Section 7.2 Gated SR Latch Figure 7.6 (a)," in *Fundamentals of digital logic with VHDL Design*, 3rd ed., New York, NY: McGraw-Hill, 2009, p. 386.

[2] "FSM #6 of Figure 1 State Diagram Assignments" in *Lab 5 - VHDL for Sequential Circuits: Implementing a customized State Machine,* Ryerson University Department of Electrical and Computer Engineering, Toronto, ON, p.3

[3] S. D. Brown and Z. G. Vranesic, "Section 6.2 Decoders Figure 6.15 (c)," in *Fundamentals of digital logic with VHDL Design*, 3rd ed., New York, NY: McGraw-Hill, 2009, p. 332

[2] "Part I: Procuring input data" in *Lab 6 - Design of a Simple General-Purpose Processor,* Ryerson University Department of Electrical and Computer Engineering, Toronto, ON, p.1