ECE 385

EXPERIMENT #2

A Logic Processor

I. OBJECTIVE

In this experiment, you will design and build a bit-serial logic operation processor. The design will utilize two 4-bit shift registers, several multiplexers, and some type of counter. The circuit will be capable of calculating eight different functions and routing the results of those operations in four different ways. A finite state machine will be implemented to serve as the control unit of the circuit.

II. INTRODUCTION

In the experiment on storage, shift registers were used to store data by shifting a bit out of one end of the shift register and then shifting the same bit into the other end. However, when a write operation was performed, a new bit was shifted into the register. Taking this process one step further, we will modify, not one, but all the data bits in the register using a designated logical function. The circuit will provide the capability of bit-wise logical operations. We only want to perform the logical function once on each bit of the data, so we will also need to keep the data from circulating more than once through the circuit.

The operations that this circuit will perform are like the bit-wise logical functions provided in machine level programming. For example, your circuit will be able to perform a bitwise AND of the two operands that are stored in the two registers (RegA and RegB) and be able to place the result in either of the registers, leaving the other register unchanged in the process. A bitwise AND means that each bit of RegA is logically ANDed with the corresponding bit in RegB. So, if RegA = $a_3a_2a_1a_0$ and RegB = $b_3b_2b_1b_0$ then the destination register (RegA or RegB) will hold (a_3b_3), (a_2b_2), (a_1b_1), (a_0b_0) after the computation is complete. This is equivalent the machine code instruction of the form:

```
AND R0, R1, R0     /*  R0 and R1 = > R0  */
```

The other functions are OR, XOR, NAND, NOR, XNOR, CLR, SET, and SWAP. The complete set of functions and their corresponding control inputs are tabulated below (Table 1).

The block diagram of the circuit you will design for this experiment is shown below (Fig. 1). It includes

1) a **register unit** that contains two 4-bit registers, which we will refer to as **RegA** and **RegB**,

2) a **computation unit** that executes the desired logical computation,

3) a **routing unit** that routes the signals back to the register unit after computation, and

4) a **control unit** that generates control inputs to the register unit.

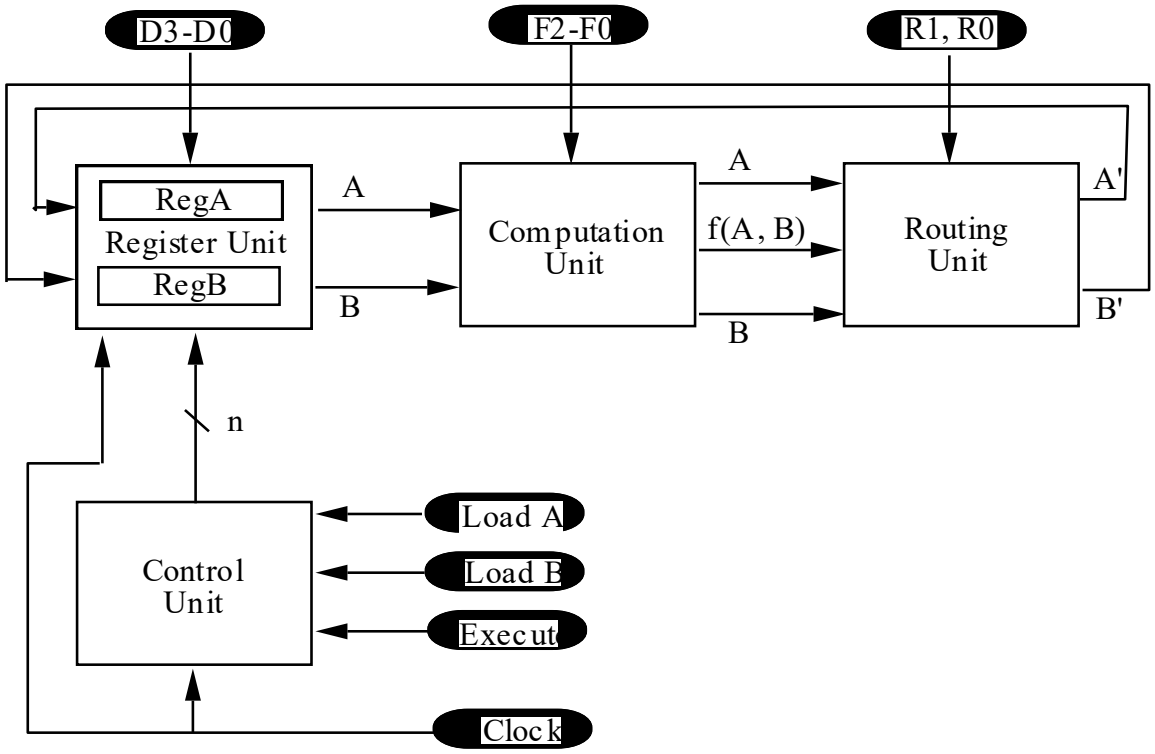

Figure 1: Block Diagram

Register Unit

The register unit will be made up of two 4-bit shift registers that will hold the values of Register A (RegA) and Register B (RegB). The contents of these registers should be displayed so that the contents before and after execution can be inspected. The control of these registers will be provided from the Control Unit while the serial input will be provided from the routing unit.

Computation Unit

The computation unit will accept as inputs the contents of RegA and RegB, and the function selection inputs F2, F1, F0. The unit will output the logical function f(A, B) specified by <F2, F1, F0> and will also output the A and B inputs unchanged. The three outputs will be fed to the Routing Unit.

Routing Unit

The routing unit will accept the A, B, and f(A, B) inputs and, based on the routing selection inputs R1, R0, will determine which signals to feed to the A' (new A) and B' (new B) outputs.

TABLE 1: Functions

Function Selection Inputs			Computation Unit Output
F2	**F1**	**F0**	**f(A, B)**
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

Routing Selection		Router Output	
R1	**R0**	**A***	**B***
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

3.4

Control Unit

The control unit will accept the following inputs: Load A, Load B, Execute, and the clock signal. The Load A and Load B inputs will perform parallel loads from the data input switches (D3-D0) into the A and B registers. Execute tells the control unit that the select switches and the register contents are ready for execution and that the control unit should begin the computation cycle. The control unit then shifts the register unit the required number of times and halts until the next execution is requested. Obviously, some type of mechanism to keep track of the shifts will be required. The clock input should be taken from the function generator to make the computation cycle appear to be instantaneous while also leaving the debugging capacity of single stepping.

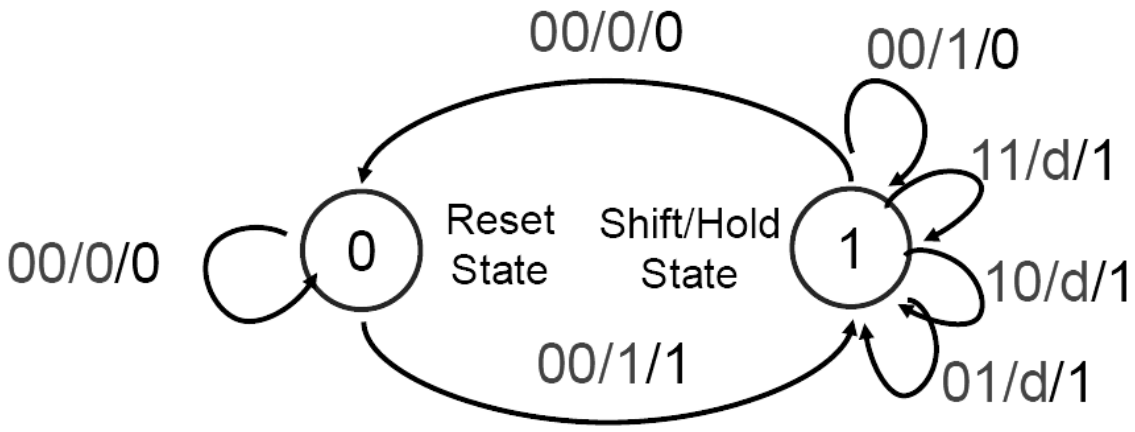

Figure 2: State Diagram

To accomplish this, a finite state machine is devised to control the operation of the register unit. There are two common state machine types: the Moore machine and the Mealy machine. During operation, both machines take in a set of inputs, transitions through a finite number of states, and output the relevant controls. The biggest difference between the two state machines is that the outputs of the Moore machine depend solely on the current state, each serving a specific output configuration, while the outputs of the Mealy machine depends on a combination of the current state and the current inputs. From this point of view, it is apparent that the Mealy machine will be able to achieve the same level of control by using fewer states than what's required by the Moore machine, which also makes the circuit implementation a little bit easier.

Table 1 shows an example of the Mealy machine for our control unit. The inputs of the Mealy machine are the ‘Execute’ switch, a single-bit state representation ‘Q’, and two-bit count ‘C1C0’. The ‘Execute’ switch dictates when the circuit should initiate the computation cycle. The single-bit state ‘Q’ split the Mealy machine into two states that serves distinct purposes – one is the reset/rest state, and the other one is the shift/halt state. And the count bits ‘C1C0’ are used to keep track of the number of shifts in the shift/halt state. Note that ‘C1C0’ represents a simplification, using a counter here reduces the number of states we must explicitly encode. The outputs of the mealy machine are the output signal ‘Reg. Shift’ (‘S’) which goes to the register unit, the next state ‘Q’, and the next count ‘C1C0’. Notice that the state descriptions are not very precise here, and this is the characteristics of the Mealy machine. Unlike the Moore machine where every state is directly linked to an operation and thus the purpose is clearly defined, the Mealy machine groups similar operations into a single state, then uses a combination of the current state and the current inputs to perform an operation.

To produce a state machine, you should follow the actual sequential operation of the circuit, where the state and the counts starts from (QC1C0=‘0000’), and the ‘Execute’ switch is held low (‘E’=’0’). As long as the ‘Execute’ switch remains low, the circuit stay in a rest state, where the register unit stays put and the next state and count also stay unchanged ($SQ^+C1^+C0^+$=‘0000’). However, at the immediate clock edge after the ‘Execute’ switch is flipped up (EQC1C0=‘1000’), the state machine moves to the shift/halt state, and sends out the signal to shift the registers and begins to increment the counter ($SQ^+C1^+C0^+$=‘1101’). The state machine in total should then carry out three additional shifts regardless of the condition of the ‘Execute’ switch. After the four shifts, the state machine will state in ($SQ^+C1^+C0^+$=‘0100’) if the ‘Execute’ switch remains high, or transitions back to ($SQ^+C1^+C0^+$=‘0000’) if the ‘Execute’ switch drops low. This completes one full cycle of bit-serial logic operation as the state machine comes back to where it had started and awaits for another full cycle of operation when the ‘Execute’ switch is flipped up again.

To transform the state machine into a physical circuit, we will next build a state transition table. Follow through the entire operation cycle to fill out as much of the state transition table as possible. You will then notice that not all combinations are valid. For example, if we are in the reset/rest state, it is not possible for our counts to hold any value other than ‘00’, and therefore we will never encounter, for example, (EQC1C0=‘0011’) during our circuit operation. If the combination is not valid, a ‘don’t care’ (‘d’) should be

placed in the outputs for easier implementation of the circuit (remember that it is ok to circle the K-map minterm over the ‘don’t cares’ without affecting the functionality of the circuit).

TABLE 1: Control unit state transition table using the Mealy state machine

Exec. Switch (‘E’)	Q	C1	C0	Reg. Shift (‘S’)	Q^+	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	D
0	0	1	0	d	d	d	D
0	0	1	1	d	d	d	D
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	D
1	0	1	0	d	d	d	D
1	0	1	1	d	d	d	D
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0

After the state transition table is made, we then proceed on producing the K-maps and circling the minterms. Since the transitioning of each of the four state machine components (S, Q, C1, C0) is dictated by four inputs (E, Q, C1, C0), you will need to convert the table to four K-maps and obtain the resulting circuits for each of the four components. Notice that while (Q, C1, C0) are internal components of the state machine, the only actual output ‘S’ will be used to control the shifting of the two registers. To adapt ‘S’ and ‘LoadA’, ‘LoadB’ to the two registers for the shift/load/halt operations, simple combinational logic will need to be devised.

Demo Points Breakdown:

Week 1

1.0 point: Show correct loading of the A and B registers.

1.0 point: Show the computation cycle is of the right length.

1.0 point: Demonstrate the four routing operations.

1.0 point: Demonstrate the eight function operations.

2.0 points: Show the computation cycle completes even if the EXECUTE switch is returned to the low position mid-computation (requires a slow clock)

Week 2

1.0 point: Functional simulation completed successfully for the 8-bit serial processor (annotations necessary)

1.0 point: RTL block diagram of the 8-bit logic processor extended from 4-bits. This can be automatically generated using Vivado, but you must show your TA you know how to generate this successfully.

2.0 points: Show a Vivado Debug Core waveform of the 8-bit logic processor, performing the operation as specified by your TA. Note that you will need to be able to operate the FPGA version of your design live for your TA via the switches and show the waveform to your TA.

III. PRE-LAB

Week 1

Design, document and build the circuit described in Part II. Your circuit should be able to perform correctly all the functions listed. You will want to study each of the chips carefully before deciding on one or the other. Be sure to make your design as efficient as possible (there is more than one way to design this circuit).

3.8

A square wave from the Pulse Generator should be used as the basic system clock. Load A, Load B, Execute, D3-D0, R1, R0, and F2-F0 should be inputs from the switches. For working at home, the switches provided with your lab kit as well as the DE10-Lite switchbox may be used. The control unit must be designed to perform the desired function once and only once each time the execute switch is flipped on. Results of the operation should be obtained even if the execute switch is flipped off in the middle of the computation cycle. You may only assume that the execute switch will remain high for at least one full clock period. Display the contents of Register A and Register B on LEDs. You may also want to include an LED that indicates when the computation cycle is complete for debugging purposes.

Week 2

A. Source code for a SystemVerilog implementation of the 4-bit logic processor has been provided for you. You will create a Vivado project to synthesize, simulate, and extend the 4-bit logic processor to 8-bits. To start, complete the bit-serial logic processor exercise from the Introduction to Vivado and Tutorial (IVT. 1-40). Include a copy of the generated diagram from Vivado of the 8-bit logic processor and the simulation waveform (with annotations) in your Lab 2 lab report and demonstrate you know how to do this to your TA. Additionally, demonstrate the generation of a Debug Core trace showing some operations of your 8-bit logic processor running on the FPGA.

IV. LAB

Follow the Lab 2 demo information on the course website.

V. POST-LAB

Document changes to your design and correct your Pre-Lab write-up, explaining any difficulties you had in debugging your circuit. Outline how the modular approach proposed in the pre-lab helps you isolate design and wiring faults, be specific and give examples from your actual lab experience.

Make sure you report discusses the following:

Describe the simplest (two-input one-output) circuit that can optionally invert a signal (i.e., one input determines if the output is equal to the other input or equal to the other input inverted). Explain why this is useful for the construction of this lab.

Explain how a modular design such as that presented above improves testability and cuts down development time.

Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

What are the differences between vSim and Vivado Debug Cores? Although both systems generate waveforms, what situations might vSim be preferred and where might debug cores be more appropriate?

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below. Note that your report must be organized into chapters with paragraphs, rather than a bullet list of materials.

1) Introduction
    a. Summarize what high-level function your circuit performs. What operations can the processor do? How many bits can it operate on? Etc. The introduction should be approximately 3 - 5 sentences.
2) Operation of the logic processor
    a. Describe the sequence of switches the user must flip to load data into the A and B registers.
    b. Describe the sequence of switches the user must flip to initiate a computation and routing operation.
3) Written description, block diagram and state machine diagram of logic processor.
    a. Written description: describe in words each block in the high-level diagram (a short paragraph for at least the register unit, computation unit, routing unit, and control unit).
    b. Include a high-level block diagram. It is acceptable to use the one in the lab manual, provided it is modified as necessary to reflect what you implemented.
    c. State Machine Diagram

i. Explicitly state if you used a Mealy or Moore machine (or some hybrid)
ii. Label each state (bubble)
    1. Give each state a meaningful name.
    2. Specify the binary flip-flop values associated with each state.
    3. If you are using a Moore Machine, label the values of all meaningful outputs associated with each state.
iii. Label each arc.
    1. The combination of inputs which trigger the arc.
    2. If you are using a Mealy Machine, label the values of all meaningful outputs associated with each arc.
    3. It is OK to label each bubble/arc with a single identifier and put the rest of the requested info on a table to reduce clutter.

4) Design steps taken and detailed circuit schematic diagram.
   a. Written procedure of the design steps taken.
      i. If you used k-maps or truth tables during design, include them here. K-maps are usually helpful for creating the next state logic in the control unit. Additionally, you should show any transformations required to 'map' the logic expressions to the discrete logic chips in your kit (e.g., you do not have AND or OR chips – so how did you modify the logic to use NAND/NOR expressions?)
      ii. Written description of the design considerations taken (did you consider multiple implementations of the same circuit and the tradeoffs of each?)
   b. Detailed Circuit Schematic
      i. Draw a gate level schematic of your circuit. It is OK to use small standard blocks like MUXes, flip-flops, and shift registers, but custom blocks like your control unit must have a gate level schematic.
      ii. If the schematic becomes too large, components such as the control unit can be represented as black boxes on the top-level schematic, and a detailed schematic of that component can be included below.
      iii. You must use a CAD tool (e.g., Fritzing, KiCAD, EAGLE, Quartus) for this portion. Note that Quartus has the standard 7400 parts in the schematic editor. Fritzing parts may be downloaded from the course website.
5) Breadboard view / Layout sheet

a. Use either the **manual layout sheet** from the general guide **or** the **Fritzing breadboard view**. *Note that this is a different diagram from the schematic in 4.b.*
b. If using the **Fritzing breadboard view, make sure you follow the example in General Guide (Figure 9) and do the following:**
    i. You should have no air-wires, all your wires should be visible – though they may be a little bit cluttered as the design is complex.
    ii. You may use the 'breadboard label' or 'note' to label the inputs, it is not necessary put all the switches into the breadboard view (you may if you want).
    iii. Similarly, you may use breadboard label to label the outputs (for the LEDs), you do not need to put in the physical LED strip (you may if you want).
    iv. If you wish to avoid having too many crossing wires, you may use 'breadboard label' to make virtual connections.
    v. Use multiple breadboards as necessary, your breadboard view should be complete, in the sense that someone should be able to build a working version of the circuit with only your breadboard view.
c. If using **the manual layout sheet** from the general guide, follow the rules provided in the General Guide (Figure 8):
    i. Note that the general guide indicates that with the **manual layout sheet** you should not draw the wires, but instead label them with unique names.
    ii. This does not conflict with 5.b.i, which applies only to the **Fritzing breadboard view**.
    iii. Follow the other rules in the example in the section leading up to General Guide (Figure 8), note that your layout must be complete, in the sense that someone should be able to build a working version of the circuit with only your layout sheet.

6) 8-bit logic processor on FPGA
a. Summary of all .SV modules and the changes you made to extend the processor to 8-bits. Specifically, you need to describe even modules which were provided, but you did not create.
b. RTL block diagram - please only include the top-level design if using the RTL viewer.

c. Include a simulation of the processor that has notes (annotations) that give information such as what operation is being performed, where the result was stored, etc.
d. Include procedure used to generate Vivado Debug Core trace, as well as the result of such trace executing an example operation. E.g.: “Step 1, Set the trigger on signal <signal here>…”
e. Include the output of the debug core trace performing an operation on the 8-bit logic processor (this doesn’t need to be the same operation as the one your TA asked to demonstrate, but it needs to be non-trivial).

7) Description of all bugs encountered, and corrective measures taken.
8) Conclusion
   a. Summarize the lab in a few sentences.
   b. Answer to all post-lab questions (they may be placed in conclusion or dispersed in more appropriate sections of the report).
   c. Make note of if there were any parts of the documentation which were unclear or otherwise need attention.

VII. APPENDIX

Module descriptions are an important part of the reports in ECE 385, and since this is the first significant FPGA lab, a brief example of how to write a module description is shown below.

Suppose you needed two 16-bit registers to store operands A and B in an adder that computes the sum of A and B. Here is example code of a 16-bit register with asynchronous reset and synchronous load that can be used for that purpose.

```
module reg16 (input [15:0] Din, input Clk, Load, Reset,
output logic [15:0] Dout);

always_ff @(posedge Clk or posedge Reset)
      begin
            if(Reset)
                  Dout <= 16'h0000;
            else if(Load)
                  Dout <= Din; //If load=1, perform parallel load on
      clock edge
      end
endmodule
```

And here is how a section of the report would describe it:

Module: `reg16.sv`

Inputs: `[15:0] Din, Clk, Load, Reset`

Outputs: `[15:0] Dout`

Description: This is a positive-edge triggered 16-bit register with asynchronous reset and synchronous load. When `Load` is high, data is loaded from `Din` into the register on the positive edge of `Clk`.

Purpose: This module is used to create the registers that store operands A and B in the adder circuit.

Simple modules can have a description and purpose that are just a sentence or two each, but more complicated modules require more detailed descriptions.

VIII. APPENDIX II

The following are the recommended pin assignments to set up the Vivado Debug Core for Lab 2.2. Note that they are broken up into two groups (one for inputs and one for outputs)

Pin Assignment Table (Inputs)

Port Name	IO Standard	Location	Comments
Din[0]	LVCMOS33	G1	On-board slider switch (SW0)
Din[1]	LVCMOS33	F2	On-board slider switch (SW1)
Din[2]	LVCMOS33	F1	On-board slider switch (SW2)
Din[3]	LVCMOS33	E2	On-board slider switch (SW3)
Din[4]	LVCMOS33	E1	On-board slider switch (SW4)
Din[5]	LVCMOS33	D2	On-board slider switch (SW5)
Din[6]	LVCMOS33	D1	On-board slider switch (SW6)
Din[7]	LVCMOS33	C2	On-board slider switch (SW7)
F[0]	LVCMOS33	B2	On-board slider switch (SW8)
F[1]	LVCMOS33	A4	On-board slider switch (SW9)
F[2]	LVCMOS33	A5	On-board slider switch (SW10)
R[0]	LVCMOS33	A6	On-board slider switch (SW11)
R[1]	LVCMOS33	C7	On-board slider switch (SW12)
LoadB	LVCMOS33	J1	On-Board Push Button (KEY1)
LoadA	LVCMOS33	G2	On-Board Push Button (KEY2)
Clk	LVCMOS33	N15	100 MHz Clock from the on-board oscillators
Reset	LVCMOS33	J2	On-Board Push Button (KEY0)
Execute	LVCMOS33	H2	On-Board Push Button (KEY3)

3.14

Pin Assignment Table (LEDs/HEX Displays)

Port Name	IO Standard	Location	Comments
hex_grid[0]	LVCMOS33	G6	On-Board eight-segment display grid
hex_grid[1]	LVCMOS33	H6	On-Board eight-segment display grid
hex_grid[2]	LVCMOS33	C3	On-Board eight-segment display grid
hex_grid[3]	LVCMOS33	B3	On-Board eight-segment display grid
hex_seg[0]	LVCMOS33	E6	On-Board eight-segment display segment
hex_seg[1]	LVCMOS33	B4	On-Board eight-segment display segment
hex_seg[2]	LVCMOS33	D5	On-Board eight-segment display segment
hex_seg[3]	LVCMOS33	C5	On-Board eight-segment display segment
hex_seg[4]	LVCMOS33	D7	On-Board eight-segment display segment
hex_seg[5]	LVCMOS33	D6	On-Board eight-segment display segment
hex_seg[6]	LVCMOS33	C4	On-Board eight-segment display segment
hex_seg[7]	LVCMOS33	B5	On-Board eight-segment display segment