ECE 385

Spring 2019

Experiment #3

Logic Processor

Austin Lee (sal3)

Zuling Chen (zulingc2)

Tuesday 3pm ABE

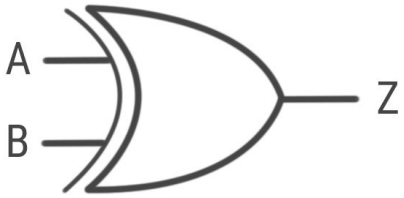
Patrick Wang, Gene Shiue

Introduction

This lab project saw us constructing a logic processor that could perform eight operations on two 4-bit registers. The implemented functions were AND, OR, XOR, and SET HIGH (all bits to 1111) and their respective negated counterparts. Each function would compare each respective bit in the two registers only once, and the output of the function was routed to show on either register A or register B. Additionally, the registers A and B could also stay constant or swap bits.

Pre-lab

A. To invert a signal in a 2-in 1-out circuit, we can simply use an XOR gate. One input will be the original signal while the other input is whether or not the signal should be inverted. The truth table and diagram of the XOR gate is:



A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

B. Developing a system using modular design is an effective method to create working parts of a whole before combining into a full circuit. By recognizing that we only have to design four of the eight functions and can use an XOR gate to the implement negated functions, we already cut the work required by half. This approach helps us save time and reduces potential bugs as we can drastically simplify the number of components needed and parts of the circuit to be built.

Written description/Design

Register Unit

Just as in the Data Storage lab, we used two 4-bit shift registers to hold data and display the output. The important distinction is that the registers do not shift until we EXECUTE a function, and only right shift four times when they do. Additionally, the registers must be able to

parallel load all four bits when the LOADA and/or LOADB signals from the switches are high. We created a truth table to derive how the select signals for the registers were to be controlled:

For Register A:

LOADA	Shift	S1
0	0	0
0	1	0
1	0	1
1	1	X

LOADA	Shift	S0
0	0	0
0	1	1
1	0	1
1	1	X

We can safely assume that LOAD and SHIFT are never activated at the same time; the last row in both truth tables are unreachable states. From looking at the truth table, we recognized that S1 is controlled by LOADA and S0 is controlled by LOADA OR SHIFT. The same logic applies to Register B: S1 (for register B) is controlled by LOADB and S0 (for register B) is controlled by LOADB OR SHIFT.

Function MUX / Computation Unit

We use a 4:1 MUX to control which function is applied to the bits in the registers. The select bits S1S0 are connected directly to our switchboard at switches labeled F1F0. The following shows which function is selected at which select combination:

F1	F0	Function(A, B)
0	0	A AND B
0	1	A OR B

1	0	A XOR B
1	1	SET HIGH (1111)

Not shown are the negated functions NAND, NOR, XNOR, and SET LOW (0000). The output of the function MUX wires into a XOR gate; if the other XOR input F2, the control signal for inverting the original signal, is high, then the function MUX output is inverted and we can access the negated functions.

To implement the functions, each input of the function MUX is from the output of a logic gate but notably the SET HIGH is wired directly to Vcc. For example, when select bits are at 00 and F2 is at 0, we are choosing to AND each respective bit in the two registers. Once F1F0 are both set to low, we use an EXECUTE switch to signal that the registers may begin shifting. Each least significant bit shifted out of register A and B are fed into the required logic gate--in this case NAND then NOT-- where the output is then selected in the function MUX. The function MUX's output signal is checked to be inverted in an XOR gate before entering the serial right shift input of the desired register. The final result is, say R1R0 is at 01, register A stays constant and register B shows the output of A AND B.

Routing MUX

For routing options, the registers A and B can either stay constant, swap, or show the output of the function. The select bits S1S0 for the 4:1 MUX are again wired directly to switches labeled R1R0. The following shows which function is selected at which select combinations:

R1	R0	Register A output	Register B output
0	0	A (constant)	B (constant)
0	1	A	F
1	0	F	B
1	1	B (swap)	A(swap)

The contents of each register are finalized 4 clock cycles (one full right-shift cycle) after the EXECUTE signal is turned on. If R1R0 is at either 00 or 11, the function output is ignored and only the current contents of the registers are manipulated to stay constant or swap.

State machine / Control unit

Because the design of this circuit uses 4 bits to put in through its functions, the shift registers must shift an exact amount of four times when exec is flipped. In the lab manual, we were provided with a state diagram and a truth table that corresponded to a design that would result in four shifts and then a hold/do-nothing state. Because the shift registers had two selects (S1 and S0) that were required to be either 00 for hold, 11 for parallel load or 01 for right shift, we had to make our own shift signal. However, this shift signal needed to depend on the flipping of the execute switch, the current state of the circuit (whether it was in reset or shift/hold) and the current clock cycle. In the lab manual, we were also given a truth table that helped us make our K-maps to design the inputs for shift, the current state, and the clock cycles. However, we found that it would be easier to rely on a counter instead of control logic with states for our $C1^+$ and $C0^+$ and control that counter's select with our $Shift^+$ signal. The counter we used has P enable and T enable, both of which need to be low to count. To stop it from counting, when Shift is low, we can invert that signal to get a high and use that as P enable, stopping the clock.

Truth Table For $Shift/State/Clock^+$ (3.6 Table 1)

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

K-map for Shift^{+}and Q^{+}

EQ/C1C0 (SHIFT)	00	01	11	10
00	0	X	X	X
01	0	1	1	1
11	0	1	1	1
10	1	X	X	X

SOP for SHIFT

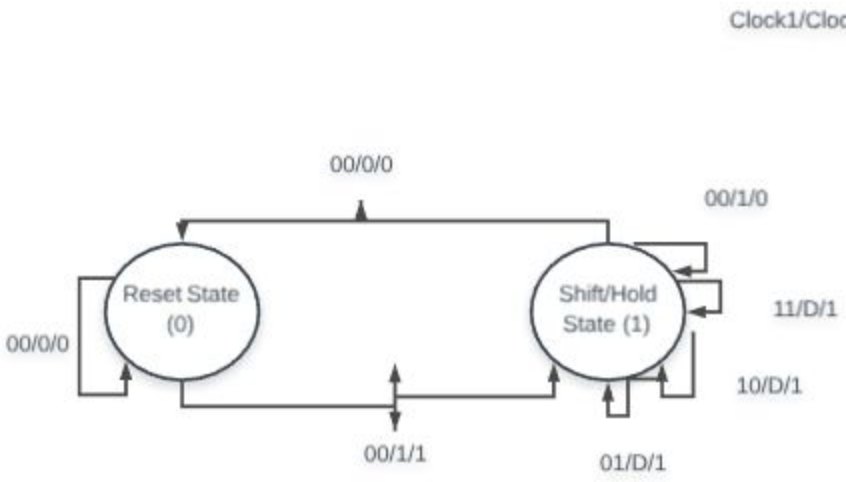
SHIFT = EQ' + C1 + C0

EQ/C1C0 (Q+)	00	01	11	10
00	0	X	X	X
01	0	1	1	1
11	1	1	1	1
10	1	X	X	X

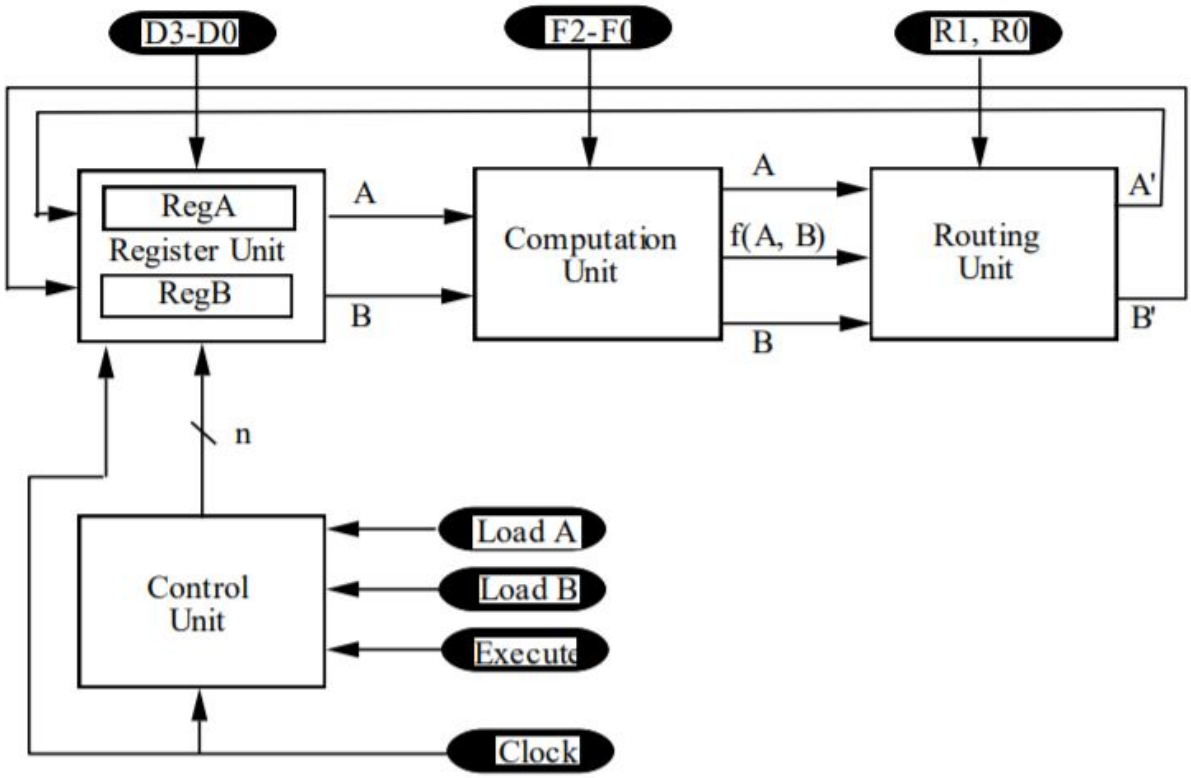
SOP for Q+

Q+ = E + C1 + C0

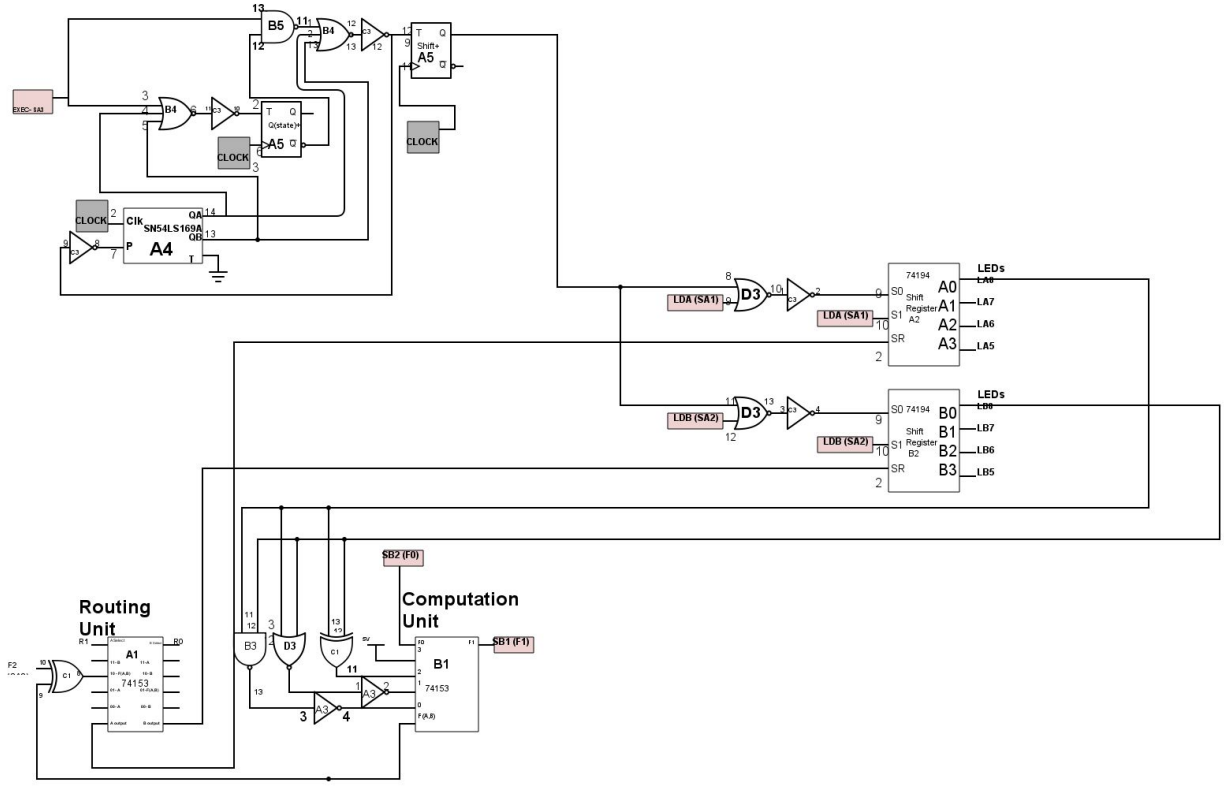
State diagram



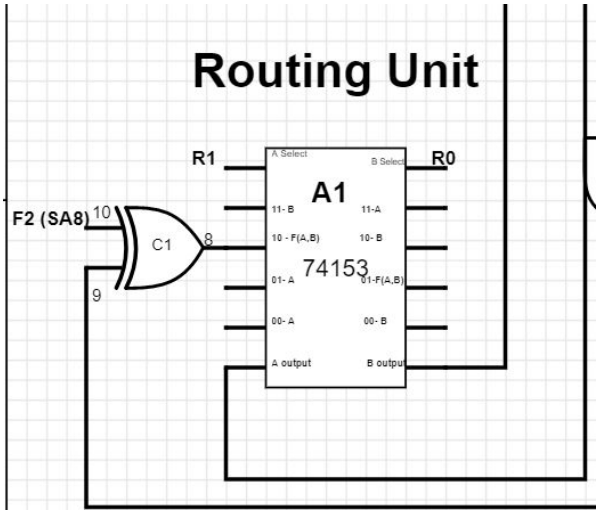
Block diagram (From Lab Manual 3.2 Figure 1)



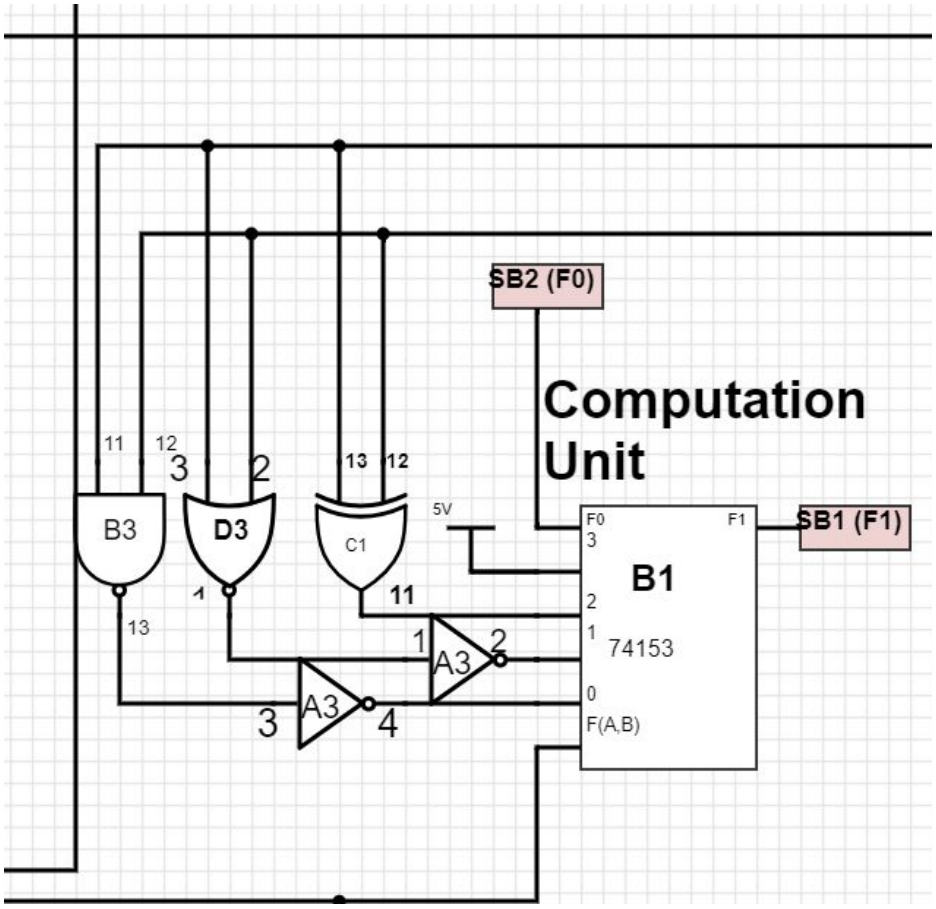
Circuit diagram



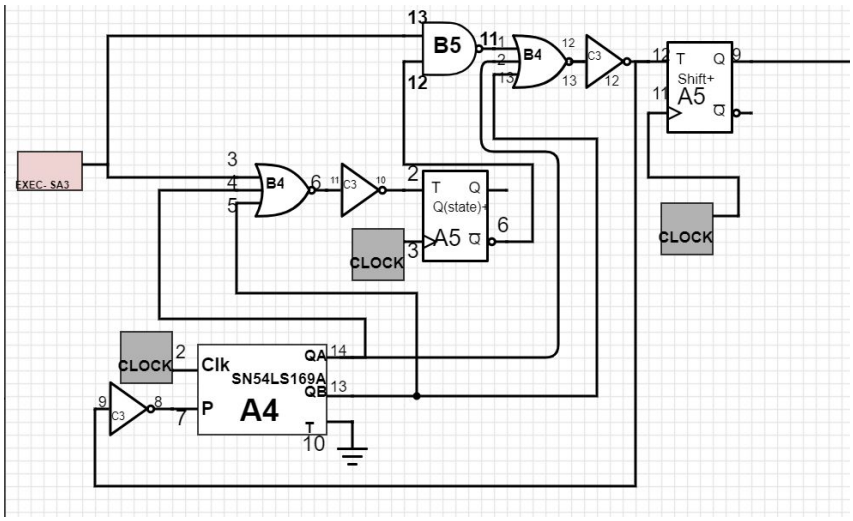
Routing Unit



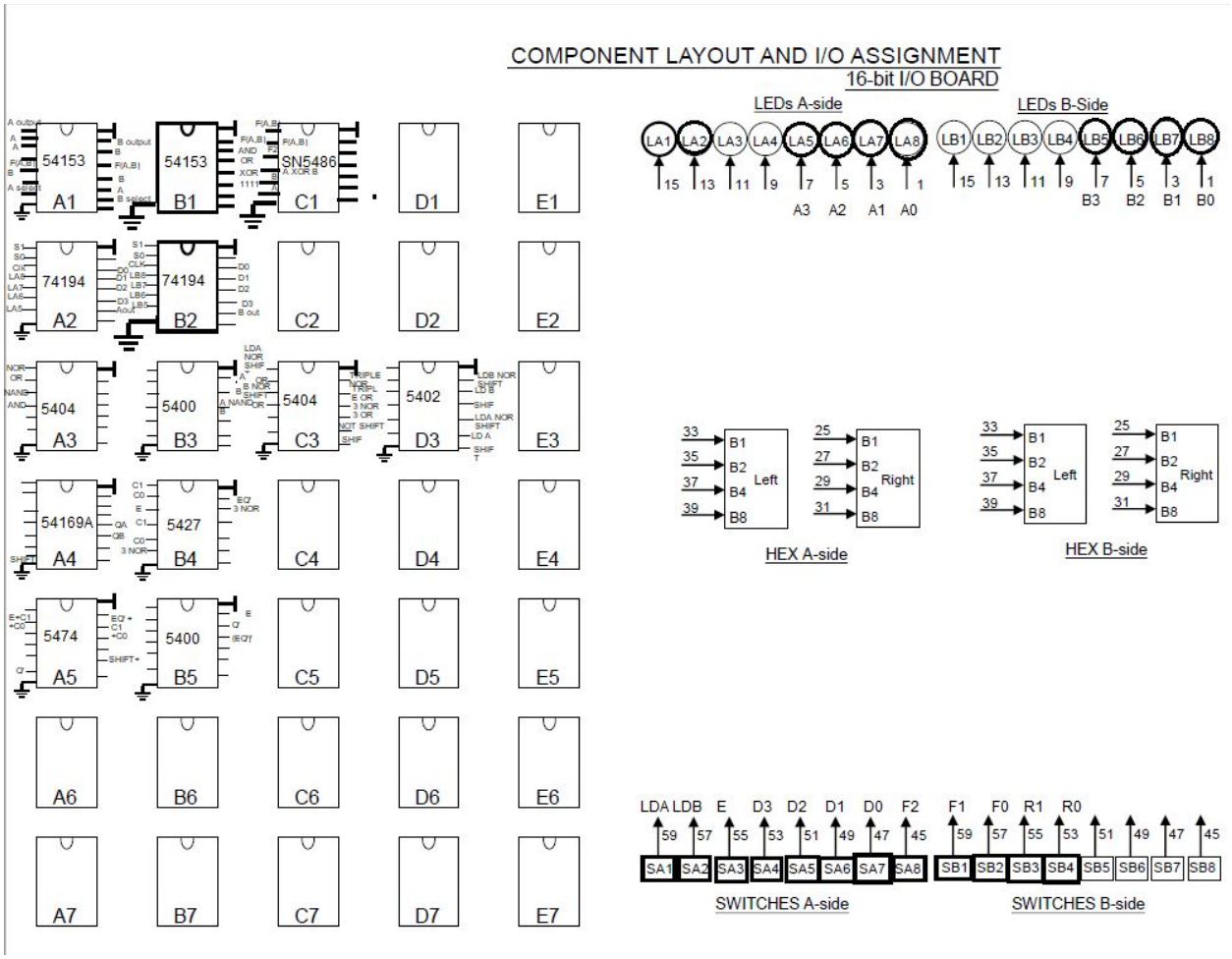
Computation Unit



Shift and Q state Unit



Component layout sheet



Conclusion / Post-lab

We were met with a few challenges during the finalization of our circuit. One of our MUX chips had loose pins which stagnated our progress for some time. When that was resolved, we realized that our registers were shifting infinitely. The SHIFT signal would remain high for four clock cycles, turn to low for one, but then revert back to high, etc. We knew we could isolate the issue to the control logic since we had built each section of the lab independently; we were sure there were no issues in the register wiring or other systems. After making various truth tables and state transition diagrams, we found the issue to be that we were relying on SHIFT* (the next shift state) in our logic for the control unit. That meant that the registers were halting one cycle too late and the SHIFT state would always revert back from low to high. To fix this, we simply had to change the control logic to use the current SHIFT state and that promptly resolved our bug. We also had minor issues with the F2 XOR gate because we had originally

built an XOR using NAND gates. Once we switched it out for the XOR logic gate chip, it worked as intended. Besides that, the rest of the process went smoothly. Modular design helped us to debug individual parts of the circuit rather than taking a look at the circuit as a whole. We tested out the routing unit first and the control logic unit, leaving the computation unit for last. Because we did this, finding a bug and fixing a bug in one unit would result in us not having to revisit the unit. Our state machine was designed in mind to just have two states, a reset state where the shift signal sent is a zero and a hold/shift state that needed us to loop exactly four times. A mealy machine is a machine whose outputs depend on its inputs and current states but a moore machine solely depends on the current state. A mealy machine proved more ideal for this lab because it would result in fewer states than using the Moore machine and that would make building and designing the control logic a lot easier.

All in all, we were completely successful in achieving the objectives of this experiment.