ECE385

Spring 2020

Experiment #3

A Logic Processor

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

Introduction

In this lab, we build a bit-serial logic operation processor, which can perform bitwise logic operations on two 4-bit binary strings. There are eight operations and four routing paths built-in to treat the data. The core of this lab is how to properly design the control unit to send correct signals to other parts of the circuit to shift data for exactly four clock cycles to perform logic operations and route back in a preset way after each EXECUTE order.

Pre-lab questions

A. An XOR gate is the simplest circuit that can fulfill the functionality of using one signal to whether to flip a signal or not. As the circuit diagram shown below, if the INPUT signal needs to be inverted, we just set the INVERT to HIGH. In this way, the OUTPUT, which is INPUT XOR INVERT, is always the inverted signal of INPUT. (1 XOR 1 = 0, 0 XOR 1 = 1) Similarly, if the OUTPUT needs to equal to the INPUT, we can turn off the INVERT and the INPUT signal can then pass the gate with no changes. (1 XOR 0 = 1, 0 XOR 0 = 0)

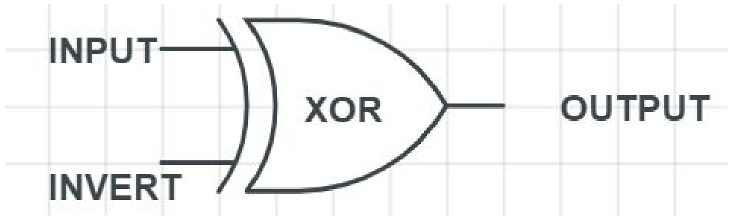


Figure 1. Circuit Diagram for Pre-lab A

B. Considering the example above, if the same functionality is implemented using an inverter or a Flip-flop attached to a 2-to-1 MUX instead of a single XOR gate, more chips are used. Therefore, more components need to be tested and the circuit would be at a higher risk of encountering bugs and component failure. In general, the entire circuit can always be split into smaller modules, each performing some certain functions. In this lab, the entire circuit is split into units including the control unit, computation unit, routing unit, etc. In this manner, the development time is greatly cut down because instead of designing the circuit as one entire piece, we are now dealing with several small pieces of modules, each with some inputs and outputs. Then, we just wire all the parts together to finish the entire circuit. With such a modular design, we can also test each part before attaching them together. This also reduces testability and helps save a lot of time when the circuit has some unexpected bugs to be fixed.

Operation of the logic processor

LOAD REGISTERS

To load data into shift registers A and B, we take advantage of the parallel loading function equipped within the chips. We first adjust switches D3-D0 corresponding to the data to be loaded into registers, and then we turn on either LOAD A or LOAD B, depending on which register needs to be loaded.

INITIATE COMPUTATION & ROUTING

To start a complete operation procedure, we first need to load both register A and B as described above. Then, we have to flip switches indicating F2-F0 to pick a logic operation that the circuit will perform, as well as switches for R1 and R0 to one way to route back the results. After loading in data, adjusting operation mode and routing mode, turn on EXECUTE and the circuit will then perform the entire process as desired.

Written description and diagrams of logic processor

The circuit is composed of four parts. A control unit that outputs the control bits for the mode of shift registers in the Register Unit. A Register Unit that is mainly composed of two four-bit shift registers. This part also takes *D3-D0* as the four-bit input that was parallel loaded into the shift registers. A Computational Unit that takes the output(rightmost bit) of the shift registers *A* and *B* and performs a single bitwise logic operation *f(A, B)* that was selected by input *F2-F0.* This part is mainly composed of several logic gates and a 2-to-1 MUX that selects whether we want the result of the operation or the inverse of it. The output of the Computational Unit is *A*, *B* and *F* that was used as input of the Routing Unit. A Routing Unit that takes *A*, *B* and *f(A, B)* as the data input. This part is mainly composed of two 4-to-1 MUXs that are used to select the data needed to load to the left side of the shift registers in the Register Unit. The selecting bits of this unit are *R1* and *R0* that was controlled by user flipping switches.

Shown below are the corresponding logic operations performed by the Computation Unit selected by *F2-F0* and the corresponding output selected by *R1* and *R0* in the Routing Unit.

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

Table 1. Outputs selected by corresponding inputs of Computation and Routing Units

Instead of Using logic gates to perform eight different logics and use an 8-to-1 MUX to select using selecting bit F2, F1, F0, we only implemented half of the structure described above because the second half of the Computation table is just the inverse of the first one. We used a 4-to-1 MUX selecting the four operations we implemented and a 2-to-1 MUX selecting the actual signal or the inverse of it using selecting bit *F2.* We could have just used an XOR gate for this part as mentioned in the pre-lab to reduce the gates.

Shown below is the High-level block diagram for the circuit. As shown, the circuit takes *D3-D0, F2-F0, R1, R0, Load A, LoadB, Execute* as the user input and uses

Clock for necessary components. Because of the Control Unit, the shift registers of the Register Unit will stop shifting after four shifts and the resulting four bits in the shift register are connected to the LEDs to show the result of the computation.

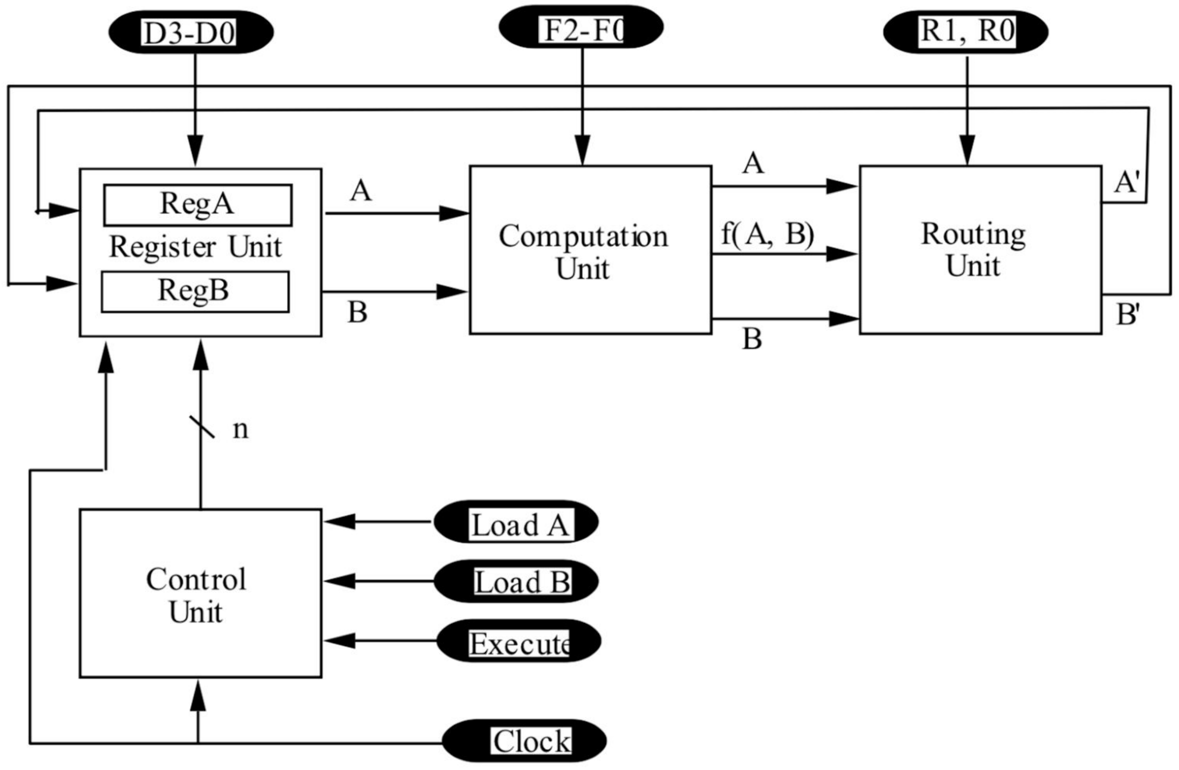


Figure 2. High-level block diagram

Shown below is the Mealy State Diagram for the finite state machine of the Control Unit. The finite state machine has two states (Represented by *Q*, see the detailed circuit description in the next section). The state value becomes 0 when it is in the Reset state and becomes 1 when it is in the Shift/Hold state. The first input labeled in light blue in the diagram represents the two-bit count value from the counter. The second input labeled in dark blue in the diagram represents the one bit execute controlled by the user flipping the switch. The output in black is basically the output of the Control Unit that is represented by *S* in the detailed circuit description. When in Shift/Hold state, the machine takes an arc combination of 00/0/0 and transfers into Reset State. When in Reset State, the machine takes an arc combination of 00/1/1 and transfers into Shift/Hold State. Otherwise, the combinations either do not trigger the machine or do not change the state of the machine. The finite state machine is essential in making the shift register keep shifting when *Execute* is turned off within four clock cycles.

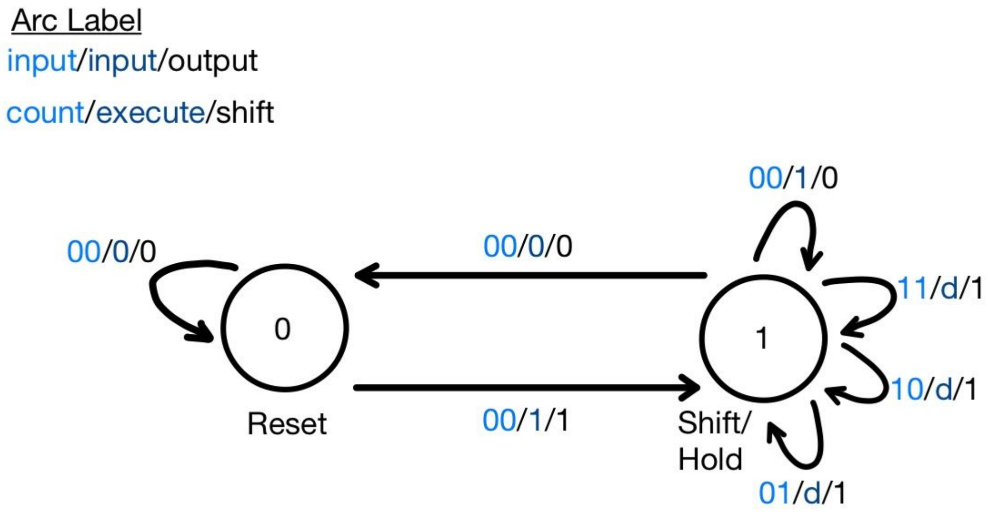


Figure 3. Mealy State Diagram

Design steps and detailed circuit schematics

E	*Q*	*C1*	*C0*	*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

Table 2. Truth Table for State Machine and Output of Control Unit

S	*C1C0*				
		00	01	11	10
	00	0	d	d	d
EQ	01	0	1	1	1

	11	0	1	1	1
	10	1	d	d	d

Table 3. K-Map for S

$$S = EQ' + C1 + C0$$

Q^+	*C1C0*				
EQ		00	01	11	10
	00	0	d	d	d
	01	0	1	1	1
	11	1	1	1	1
	10	1	d	d	d

Table 4. K-Map for Q^+

$$Q^+ = E + C1 + C0$$

Counter Load Pin	*C1C0*				
EQ		00	01	11	10
	00	0	d	d	d
	01	0	1	1	1
	11	0	1	1	1
	10	1	d	d	d

Table 5. K-Map for Counter Load (Same as S)

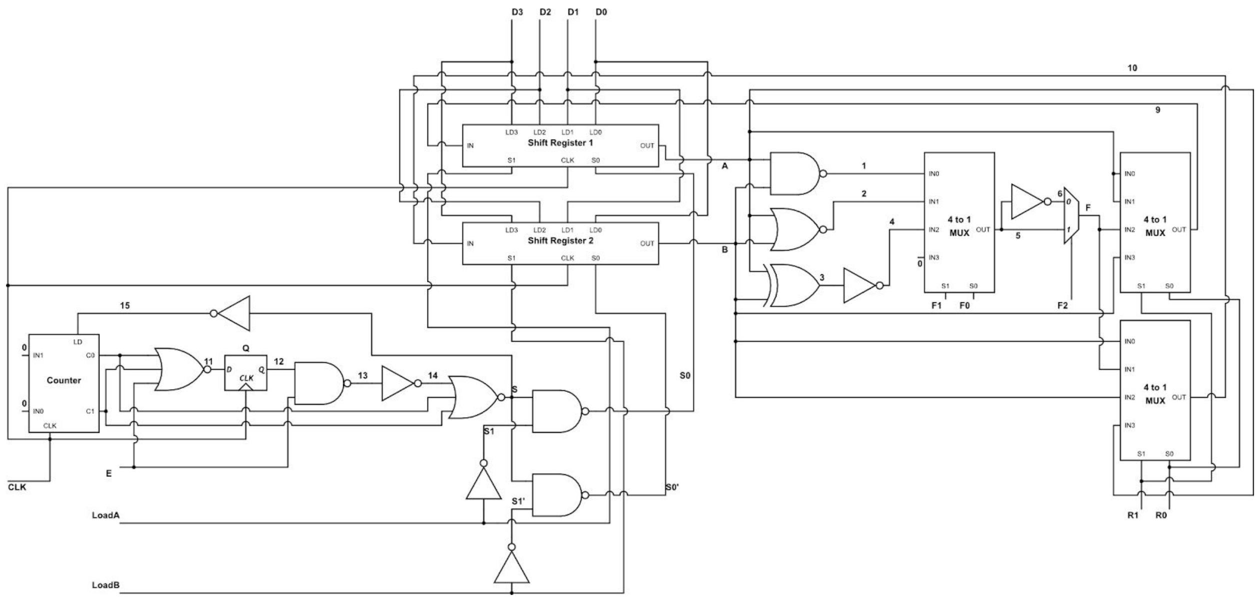
Counter Load Pin= EQ' + C1 + C0 (for SN74S169 counter)

Shown above are the Truth Tables for the output of the Control Unit and state machine and the corresponding K-Maps. We used K-Maps in our design for the logic part. The first step of our design is to write out the output *S* and the next state of *Q* using the logic expressions.

At first, we did not think of the state machine but just connected Q^{+} as a part of the logic combination of *S*. After building the Computation Unit, we tested the output and made sure this unit was working correctly. Afterward we built the Routing Unit but could not test this unit since we did not have the Control Unit working.

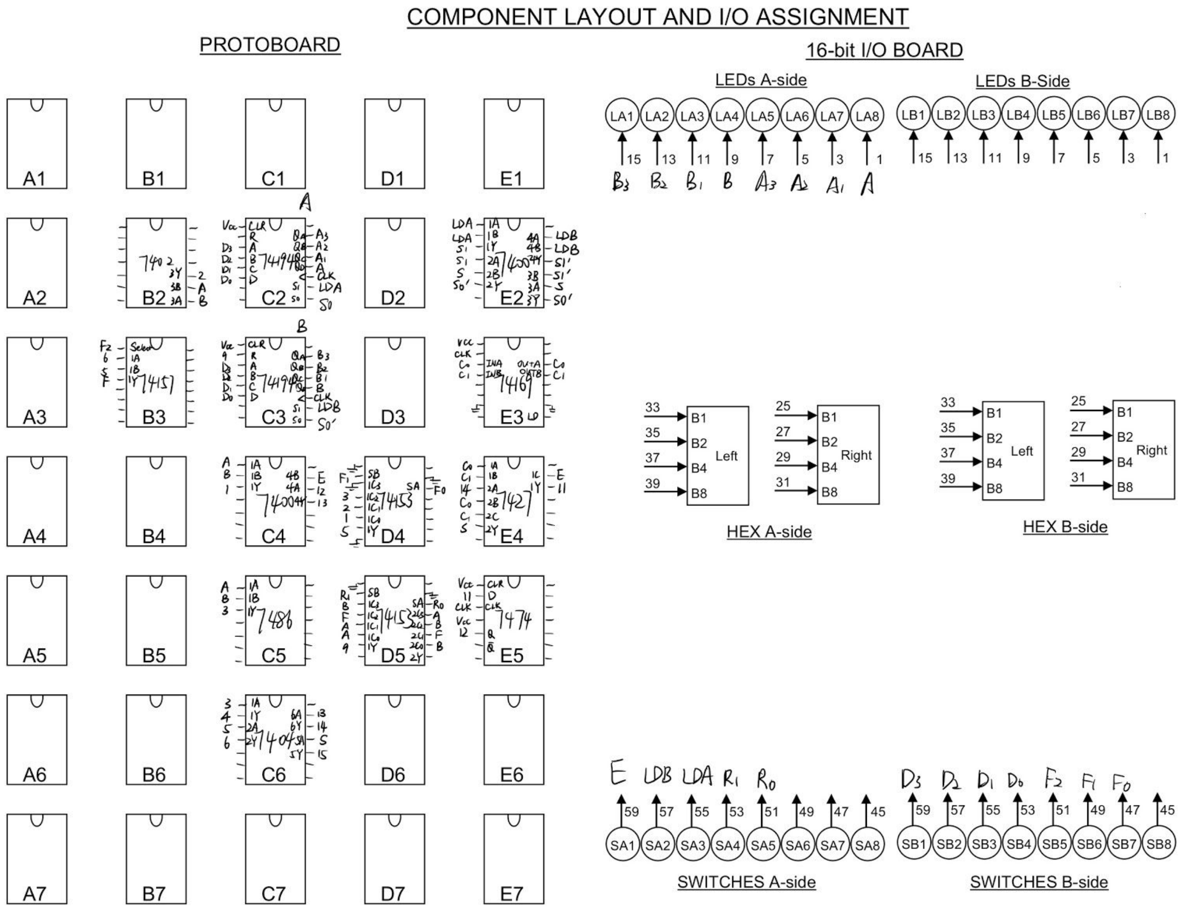
Secondly, we built the Control Unit but the behavior of the Circuit was not expected. We took a close look at the Control Unit and the state machine. We figured out the importance of the State machine and added the flip-flop to the Control Unit.

After fixing the previous part, the register was shifting but never stopped. We took a close look at the Counter output and figured that the state machine makes counter stops at some state and arc combinations. It means we have to hold the counter using the control bit of the counter. We used Load that loads into the counter certain values (*C1* = 0, *C0* = 0 in this case). And we found that it was just the inverse of *S* after drawing out the K-Map using relationships of *C1, C0* and $C1^{+}$, $C0^{+}$. The Circuit still did not seem to work correctly but the register was shifting and stopped after four clock cycles. Using the oscilloscope, we found several bugs and fixed them. Then the circuit worked.



Detailed circuit schematics

Component layout



Component layout

Description of bugs encountered

In the wiring process for the Routing Unit, we have mistaken the pins of 4-1MUX so that we connect four input signals in a completely opposite order. When we tested the circuit and eventually spotted the error region is within the Routing Unit, we visually examined all the wire corrections of each pin to check the circuit flow and verify them with the chip diagrams provided in the datasheet. We also check signals in and out of the 4-to-1 MUX to test whether it is working properly. Then we found the mismatch between four inputs, which caused chaos for the entire unit.

Loading value into the registers was considered the easiest part of this lab but we were having trouble with it at first. After correcting the Control Unit, we found that the loaded value was incorrect even though the value of control and input are all expected. To be specific, when flipping the switch on and off, we found that the value loaded changes. We immediately thought of debouncing-switch. After switching to debouncing-switch for *Load A, Load B* and *Execute*, the bug was solved.

It was mentioned in the designed step that we did not consider the state machine too much at first. It was essential to have the states for controlling the registers to stop after four clock cycles. It was also essential to have the counter enable bit to make sure the counter does not keep counting in some states that the counter should stop.

Because of the lack of resistors for building the switches, we used several wrong resistors that had very large resistances. Some parts of the circuit voltage level became low (around 1-3 V) and many logic chips did not seem to work anymore. After testing the input and output of the logic gates, we figured it out and replaced those resistors with the standard 1kΩ resistors for the switches.

Post-lab questions

The optional NOT mentioned in the pre-lab part can be used in the Computational part of the circuit to select *f(A, B)* or its inverse using a single XOR gate. We did not use this design in our implementation but we can use it as an improvement (reduce a 2-to-1 MUX and a not gate). We followed the instructions in the pre-lab and built the circuit by units. Testing and debugging one unit is simple but testing a series of units is complex. For example, we built the Register Unit and Computation Unit first. We gave the units sample input and tested the behavior of the two units first. After testing the behaviors of all possible inputs, we were confirmed that this part would work fine if the input signal is fine. This modular approach has significantly reduced the amount of time in the debugging process.

We thought of the Moore State Machine and the Mealy State Machine in the designing process. Moore State Machine is easily understood but Mealy State Machine took some time. In the construction process, the Moore State Machine is harder to build for more states and more connections. Thus Moore State Machine would possibly create more bugs. That is the reason why we chose to use Mealy State Machine instead. However, in the debugging process, we think Moore State Machine has more advantages. We could test the signals from different states and find out which part could create the bug easily in Moore State Machine.

Conclusion

In this lab, we created a simple logic unit that performs bitwise logic operations for four-bits binary strings. We constructed the circuit by different units and tested them separately to minimize the occurrence of bugs. We understood what State Machines could do in control logic and the trade-offs between different State Machines in practice. We also encountered several bugs, some coming from designing errors and some from practical construction.