ECE 385

Spring 2019

Experiment #3

A Logic Processor

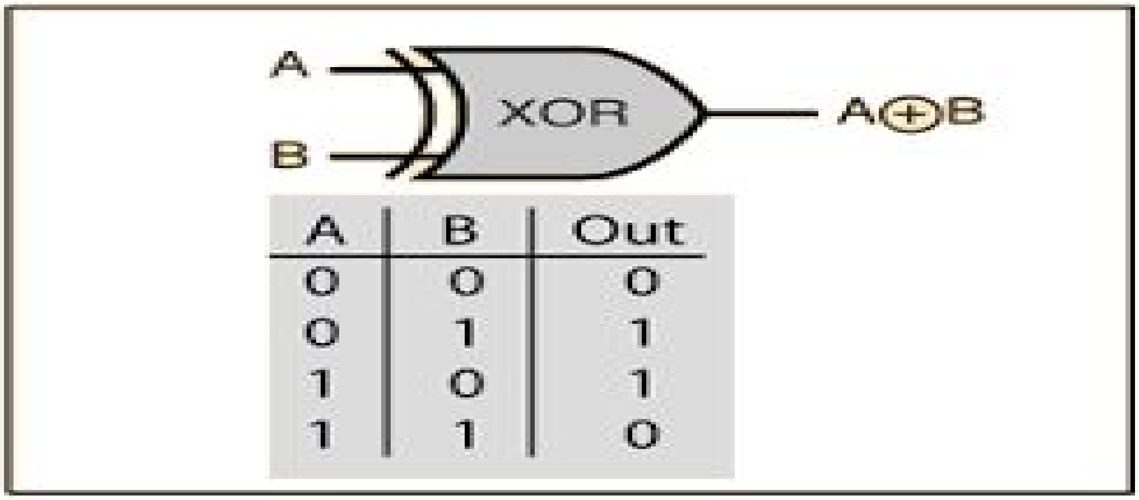
Rob Audino and Soham Karanjikar
Section ABJ, Friday 2:00-4:50
TA: David Zhang

Introduction:

The goal of this experiment was to design a logic processor capable of performing eight different logical operations on two 4 bit words (A and B). These operations included AND, OR, XOR, NAND, NOR, and XNOR, as well as the option to simply produce the words 0000 and 1111. In addition, this logic processor should also have the routing capability to store the result of the logical operation in either of the shift registers that originally held the two 4 bit words (A and B) as well as the ability to not change the original words and also switch the locations of the original words A and B.

Pre Lab Questions:

1. Q: 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). Sketch your circuit.
   A: The simplest circuit that can optionally invert a signal is an XOR gate. Let's take the example of a circuit that is just an XOR gate with inputs A, B and output Out. We can use the B signal as the input to be inverted and A as the option bit. If we assign the values of A with 0 being 'equal' and 1 being 'invert', then we have the following truth table as the result:

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

   This truth table is clearly what is wanted in the problem statement. If A is 0 (equal), then Out simply equals B. If A is 1 (invert), then Out equals NOT(B).

2. Q: Explain how a modular design such as that presented above improves testability and cuts down development time. Propose an approach that could be used to troubleshoot the modular circuit above if it appeared to be completing the computation cycle correctly but was not giving the correct output. (Be specific.)
   A: A modular design such as this can be useful in finding errors within a circuit, as each intermediate output can be tested, allowing the user to identify the exact location of the

error. Using a modular design also allows for easier replacement if a chip were to die. Having a circuit where all the parts are all over the place (not modular) makes it a lot harder to integrate another component into the board or removing something.

Operation of the Logic Processors:

A very specific sequence of switches is required for the loading, computation, and routing operations. For the loading operation, the user must first use the 4 DIN switches (switches 9-12 in our circuit) to select the bits in the word that they want to enter into one of the two registers (A and B). Next, the user has to select either the Load A (switch 5) or Load B (switch 4) switch to 1 in order to load the bits selected by the DIN switches into the selected register (either A or B).

For any computation, the user has to use the F2, F1, and F0 switches (switches 1-3 in our circuit) to select which logical operation they want to be completed. These options are described by the table below:

F2	F1	F0	Operation
0	0	0	AND
0	0	1	OR
0	1	0	XOR
0	1	1	1111
1	0	0	NAND
1	0	1	NOR
1	1	0	XNOR
1	1	1	0000

Once an operation is selected, the user has to select a routing option in order to store the result somewhere. These operations are selected with R1 (switch 7) and R0 (switch 8) and are described by the following table:

R1	R0	New A	New B
0	0	A	B
0	1	A	Result
1	0	Result	B
1	1	B	A

Once the both the computation and the routing options have been selected, the user is free to flip the execute switch (switch 6). This switch will begin the computing and routing process only when it is flipped from 0 to 1, and performs exactly 4 operations: one for each bit of the two words.

Written Description of Blocks in Block Diagram:

Register Unit:

The register unit consists of two 4 bit shift registers: one for each of the two 4 bit words (A and B). The registers have three modes that we used: parallel load, shift right, and hold. When Load A or Load B switch is set to high, the corresponding register goes into parallel load mode so that the data from the Data In switches loads into the register. If the execution cycle is ongoing, then the registers shift right to each clock pulse and the data from the output of the routing unit shifts in. If the execution cycle is completed and the state machine is in rest/hold state, then the registers goes into hold mode. All of this can be done by manipulating the S1 and S0 pins on the shift registers. S1 is the signal from 'LoadA' and S0 is 'Load A' XOR 'Shift signal 'S''.

Computation Unit:

The computation unit is comprised of a few different modules. The module that does the computations(functions) consists of chips that do the specific operation. For the 1111 function, we directly connected it directly to a 5V rail; the 0000 function was the inverse of this. The operations were chosen through a 4:1 mux and the select bits of this mux were F1 and F0. Since we did not have chips to perform AND, OR and XNOR, we had to implement a 2:1 mux where one of the inputs was the output form the function module and the other input was the same line, but inverted. The select bit for this 2:1 mux was F2. The output of this mux is then sent to the routing unit.

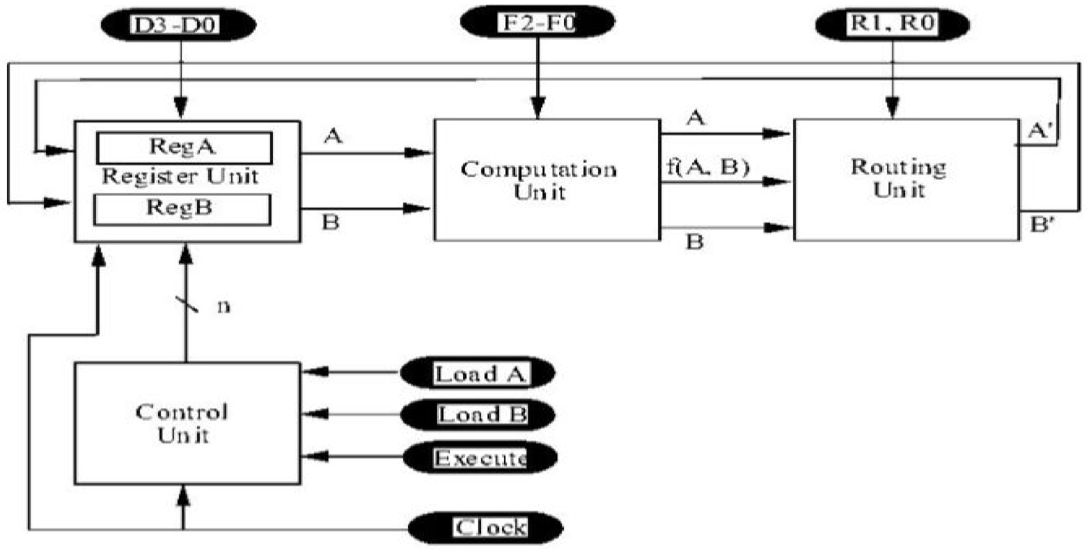
Routing Unit:

The routing unit consists of two 4:1 muxes: one for shift register A and one for shift register B. However, only 3 of the inputs to the 4:1 were utilized. The inputs to the muxes A were A, B, and the output of Computation Unit. These muxes are controlled through switches R1 and R0. Since one chip contains two 4:1 muxes and the select bits are the same for both, we had to correctly map out the inputs to the muxes so that the outputs would be correct for both of the registers; in doing so, we did not have to use two seperate chips.

Control Unit:

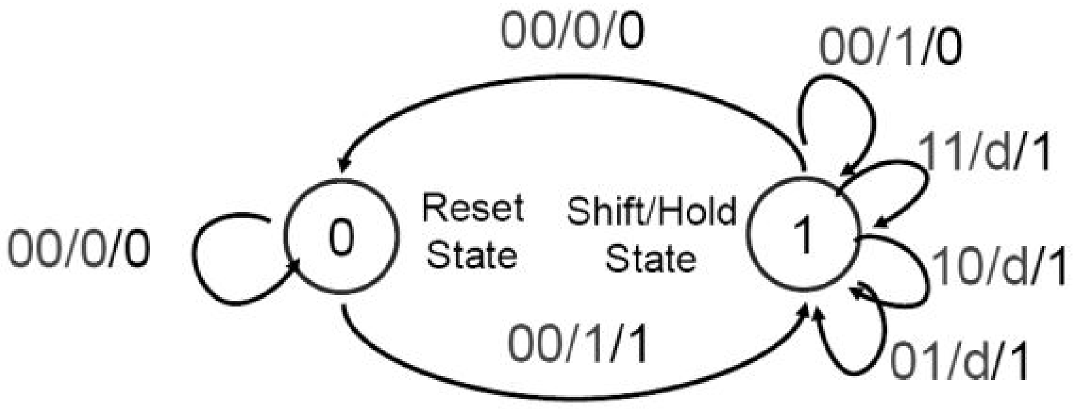
The control unit we implemented was relatively simple. We utilized one 4-bit shift register that was always parallel loaded, so every clock cycle, the contents of the shift register were updated according to the parallel inputs. The logic for the control unit was implemented as a Mealy state machine. The 4 bits in the shift register were each assigned a value: Shift signal 'S', Current state ‘Q’, Counter bit ‘C1’, Counter bit ‘C0’. The inputs to this unit from the switches were Execute, Load A and Load B. The next states of all the bits in the shift register are described in the report with their corresponding K-maps.

Block Diagram:



State Machine Diagram:

We used a Mealy machine so that it required less states and was easier to implement. The least significant bit in the diagram refers to the shifting state, the 2nd bit refers to the execute signal, the first 2 bits refer to the counter.



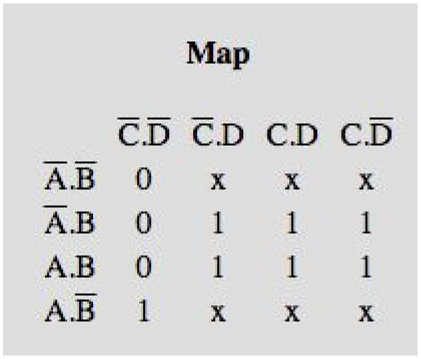
Design Steps Taken:

The design was hard to understand at first, but once we figured out the chips to use and that it only involved simple logic, the task became a lot easier. We were planning to use flip flops for the control unit at first, but a 4 bit shift register made a lot more sense since it had exactly what we needed. The tradeoff with this implementation was that we lost individual control over each block, and we could not load individual bits; however, this was irrelevant because we wanted to update each bit simultaneously. For the routing unit and ALU, we used muxes because we were instructed in lecture to do so. For the storing of data, we used another two shift registers, as we had experience with them from the previous lab. The rest of the logic was implemented using logic chips such as NAND, XOR, NOR, and Inverters. The K-maps are shown below.

<u>Karnaugh Maps</u>:

A=Execute
B=Q
C=C1
D=C0

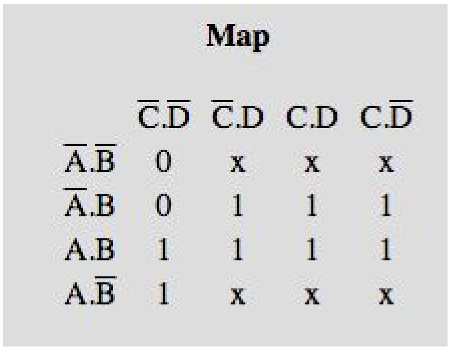
Register Shift state: y = D + C + AB'



Map

	$\overline{C}.\overline{D}$	$\overline{C}.D$	$C.D$	$C.\overline{D}$
$\overline{A}.\overline{B}$	0	x	x	x
$\overline{A}.B$	0	1	1	1
$A.B$	0	1	1	1
$A.\overline{B}$	1	x	x	x

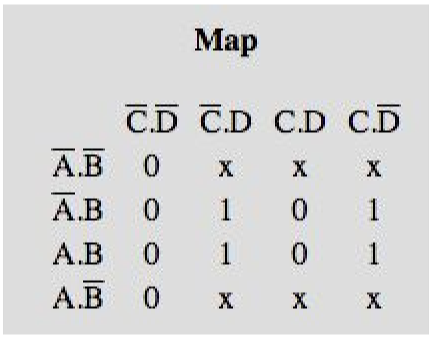
Q+: y = D + C + A



Map

	$\overline{C}.\overline{D}$	$\overline{C}.D$	$C.D$	$C.\overline{D}$
$\overline{A}.\overline{B}$	0	x	x	x
$\overline{A}.B$	0	1	1	1
$A.B$	1	1	1	1
$A.\overline{B}$	1	x	x	x

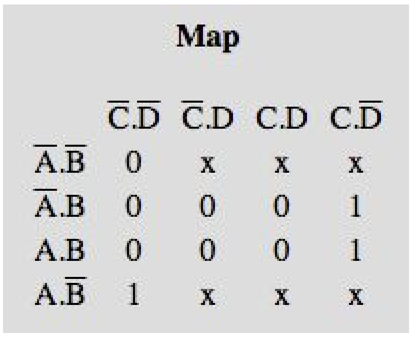
C1+: y = C'D + CD'



Map

	$\overline{C}.\overline{D}$	$\overline{C}.D$	$C.D$	$C.\overline{D}$
$\overline{A}.\overline{B}$	0	x	x	x
$\overline{A}.B$	0	1	0	1
$A.B$	0	1	0	1
$A.\overline{B}$	0	x	x	x

C0+: y = CD' + AB'

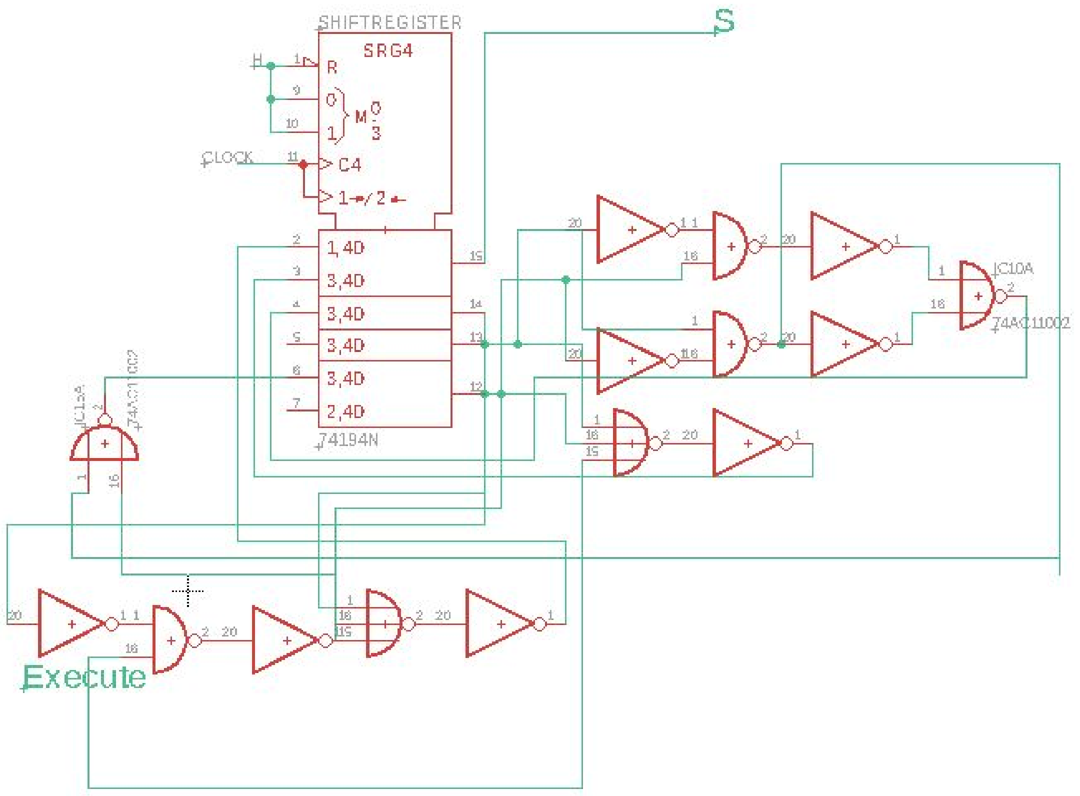


Map

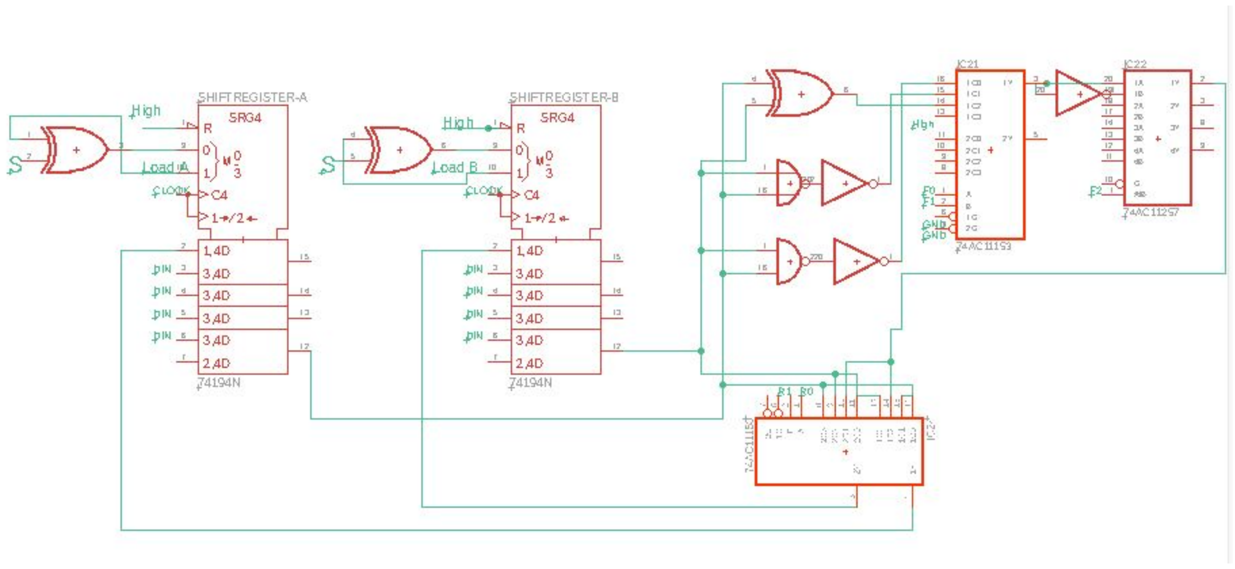
	$\overline{C}.\overline{D}$	$\overline{C}.D$	$C.D$	$C.\overline{D}$
$\overline{A}.\overline{B}$	0	x	x	x
$\overline{A}.B$	0	0	0	1
$A.B$	0	0	0	1
$A.\overline{B}$	1	x	x	x

Detailed Circuit Schematic:

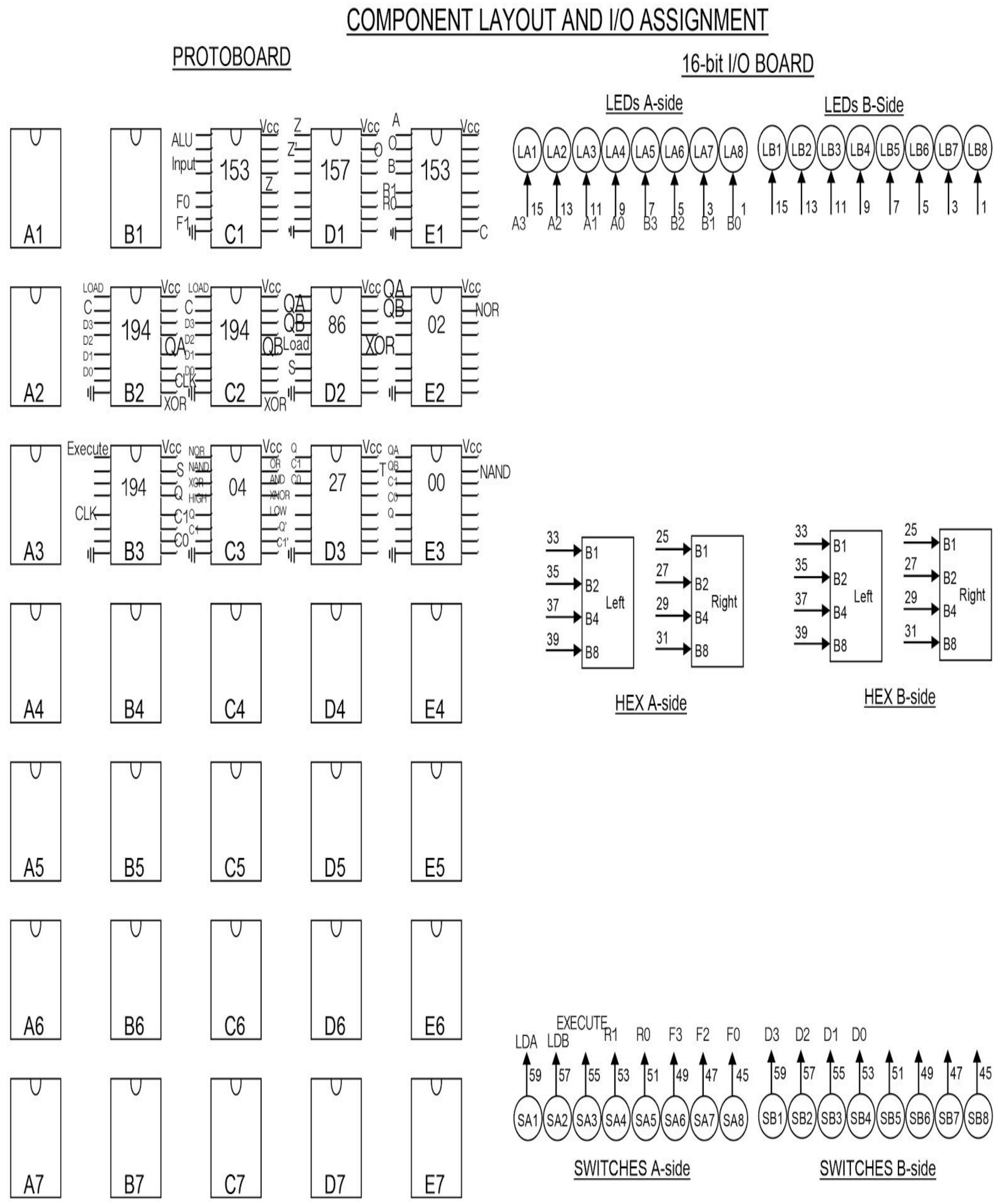
Control Unit:



Rest of the Circuit:



Layout Sheet: (S)


COMPONENT LAYOUT AND I/O ASSIGNMENT
PROTOBOARD
16-bit I/O BOARD
LEDs A-side
LEDs B-Side
LA1 LA2 LA3 LA4 LA5 LA6 LA7 LA8
LB1 LB2 LB3 LB4 LB5 LB6 LB7 LB8
15 13 11 9 7 5 3 1
A3 A2 A1 A0 B3 B2 B1 B0
C1 153 ALU Input F0 F1 Vcc Z
D1 157 Z Z' Vcc O
E1 153 A O B R1 R0 Vcc C
B2 194 LOAD C D3 D2 D1 D0 Vcc QA CLK XOR
C2 194 LOAD C D3 D2 D1 D0 Vcc QA QB XOR
D2 86 QB Load S Vcc QA QB XOR
E2 02 Vcc NOR
B3 194 Execute CLK Vcc S Q C1 C0
C3 04 NOR NAND XOR HIGH Q C1 Vcc OR AND XNOR LOW Q' C1'
D3 27 Q C1 C0 Vcc T
E3 00 QA QB C1 C0 Q Vcc NAND
A1 B1 A2 A3 A4 B4 C4 D4 E4 A5 B5 C5 D5 E5 A6 B6 C6 D6 E6 A7 B7 C7 D7 E7
33 35 37 39 B1 B2 B4 B8 Left
25 27 29 31 B1 B2 B4 B8 Right
HEX A-side
HEX B-side
LDA LDB EXECUTE R1 R0 F3 F2 F0
D3 D2 D1 D0
59 57 55 53 51 49 47 45
SA1 SA2 SA3 SA4 SA5 SA6 SA7 SA8
SB1 SB2 SB3 SB4 SB5 SB6 SB7 SB8
SWITCHES A-side
SWITCHES B-side

Description of all bugs encountered and corrective measures taken:

The only bug encountered was that we forgot to ground the strobe pin of the MUX's inputs. We encountered and solved this in demo on advice from a TA.

Post Lab Questions:

1. Q: 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 help you isolate design and wiring faults, be specific and give examples from your actual lab experience.
   A: We knew the correct design from the start, so we were able to proceed with our wiring with relatively little difficulty. The state machines, routing instructions, and computation instructions were given to us in lecture, and we implemented those relatively easily. The only error worth noting was the fact that we forgot to ground the strobe pin for the MUX's, but we fixed this in the demo and were able to earn full points. Testing the circuit at various points along the way helped us to avoid errors, and the modular design described in the prelab helped us to identify the errors when they came.
2. Q: What are the differences between Mealy and Moore machines?
   A: The output values of Moore machines are solely dependent on the current state. In addition, the outputs of Moore machines are synchronized with the clock. The output values of Mealy machines are dependent on both the current state and current inputs. To synchronize a Mealy machine, the inputs of the circuit need to be synchronized with the clock, and the outputs have to be sampled just before the clock edge.

Conclusion:

In all, this lab went fairly smoothly. Since we were essentially given the mealy machine in lecture, we had little logic left to figure out and our only errors ended up being trivial hardware errors. The board took only a couple hours to wire up, and our only error in wiring was not grounding the strobe inputs to the MUX's, and once we did, our circuit worked perfectly. To avoid this error, we could have more closely read the datasheet for the chip and realized what pins had to be grounded as well as connected to high. In summary, Lab 3 was another useful lab, not only in learning to wire, but also in adding a state machine into our working knowledge of creating hardware. We dealt with shifting bits, multiple logical operations, and routing, which will all come up in future labs.