ECE 385

Spring 2021

Experiment #2

A Logic Processor

Aishik Chakraborty, Hongbo Zheng

Section ABA

TA: Dong Kai Wang

Purpose of Circuit

The purpose of this lab was to build a logic processor capable of 8 different functions: AND, OR, XOR, set all to 1, NAND, NOR, XNOR, and set all to 0. It would then take the result of the operation and store it in either input A or input B, or not store it at all. The first part of the lab was implementing a 4-bit version of this using discrete logic. In the second part we created an 8-bit version on Quartus and implemented it on our FPGA. This lab was an attempt at building a very elementary version of a CPU.

Written Description of the Circuit

Overview

We first broke up the project into 4 sections which we individually designed: ALU, Routing unit, shift registers, and Control unit.

The ALU was designed by implementing AND, OR, XOR, and 1111 each in combinational logic (using NAND's, an XOR, and VCC) and then feeding the outputs to a 4-to-1 MUX. The select bits on the MUX were F0 and F1. The output of the MUX was then fed through an XOR gate (translated using NAND gates) with F2 as one of the inputs to the XOR.

The routing unit was comprised of a series of 2 4-to-1 MUXs, one for routing the output to overwrite B, and the other to overwrite A. The select signal for the MUXs are R0 and R1. This design was borne out of necessity as there were only a few types of MUXs available and a set number of routing possibilities.

The two shift registers were built with the 74194 4-bit bidirectional universal shift registers. We utilized the parallel load and halt functionality.

Designing the control unit was perhaps the most challenging part of this lab. Initially we tried to use pure combinational logic and think through how the necessary functionality would guide the design, however we found this approach to not work out. We instead used the truth table provided in lecture and used the equations derived from said table to come up with the final design. The two equations we arrived at were: F = EQ' + C1 + C0 and Q = E + C1 + C0. We had to use a D-Flip-Flop in order to handle an edge case. More specifically, if the enable signal is left high after the operation is completed, the D-Flip-Flop serves to detect and restart the process once a low and subsequent high enable signal is observed. A counter is also used in our design to detect when 4 clock cycles have passed after successful execution. This is done so that all 4 bits have the computation specified applied, and so that the machine can halt after that task is completed.

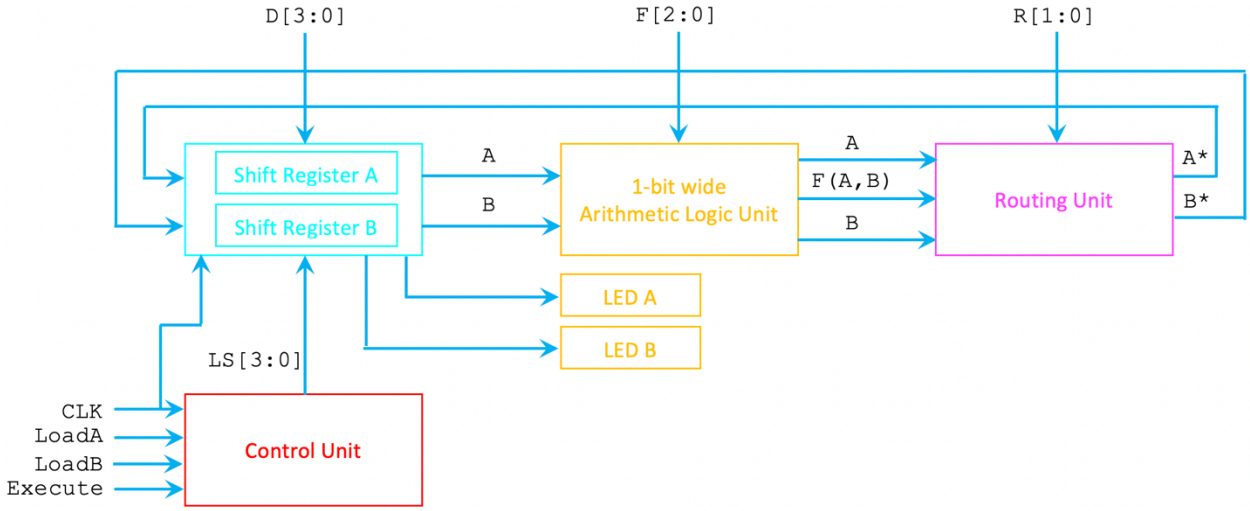
Lab 2.2 included extending our logic processor to work with 8-bit numbers using System Verilog.

Block Diagrams

High Level Block Diagram

The high-level block diagram shows all of the inputs to the logic processor as a whole and shows the accompanying output. It treats the internal components all as a combined black box. Overall, there are 6 categories of user selected inputs, and the outputs are manifested on 2 LED's. There are 3 possible outputs: Keep A and B, replace A with F (the output from ALU), or replace B with F.

Block Diagram for 4-bit Logic Processor



The more detailed block diagram shows how each of the 4 components interact with each other. The inputs to the shift registers are the 4 possible bits loaded in parallelly by the user, as well as the serial loading of values into the register during execution. The contents of these registers are displayed on the LEDs to the user.

The ALU has a 3-bit user input specifying one of 8 possible functions it can apply to the two inputs from the shift registers. The inputs to the ALU are serial with respect to each shift register (one bit at a time

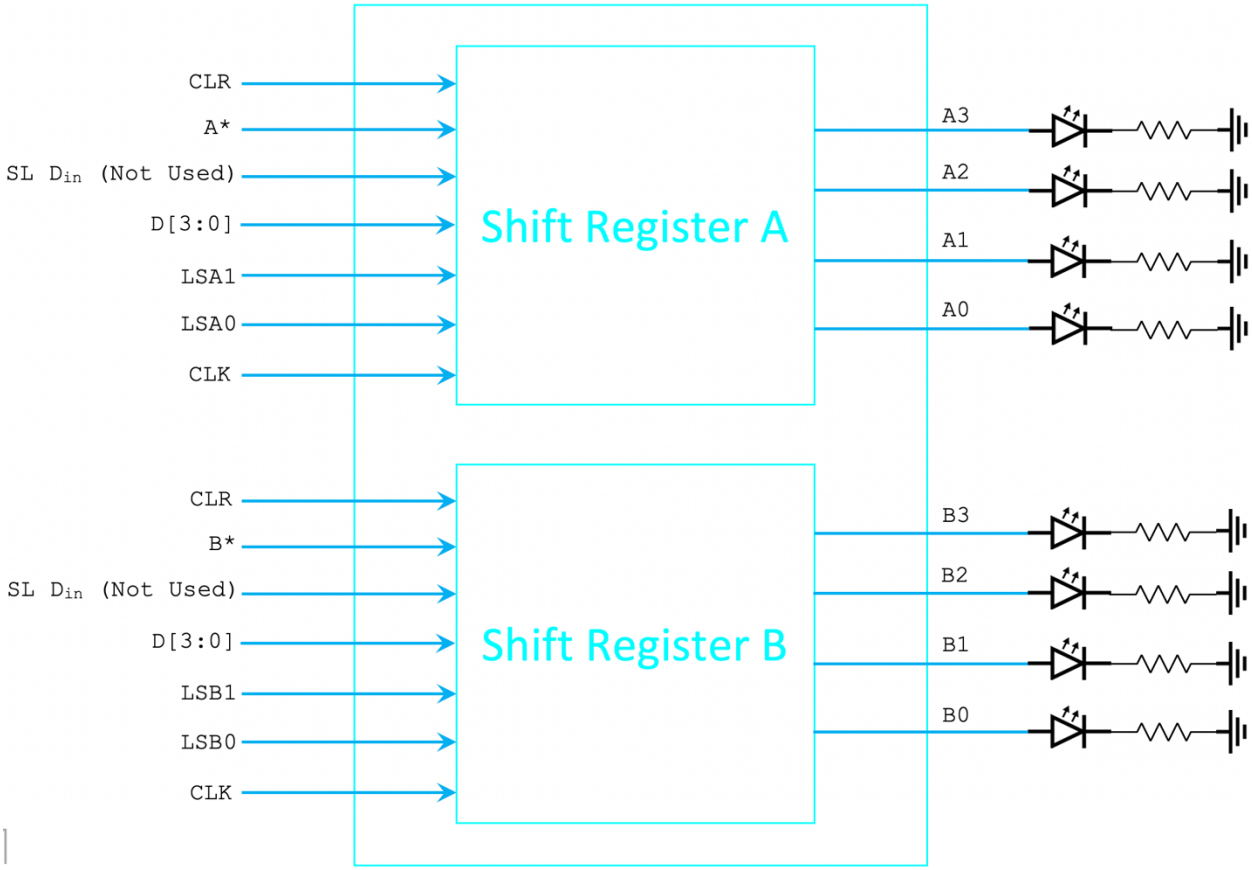
from each of the shift registers). The ALU outputs the result of the operation specified, and passes along the original contents of A and B from the shift registers to the routing unit.

The routing unit takes a 2-bit user input which encode for 3 usable routing instructions. At this point, the 3 inputs from the ALU are consolidated into 2 outputs A* and B* representing what should finally be stored in the shift registers once everything is completed.

The control unit makes sure the right number of shifts are done (4 clock cycles) in order for 1 computational cycle to be completed. It also coordinates the loading of values into the shift registers as well as the execution of tasks.

Lab Documentations

Register Unit

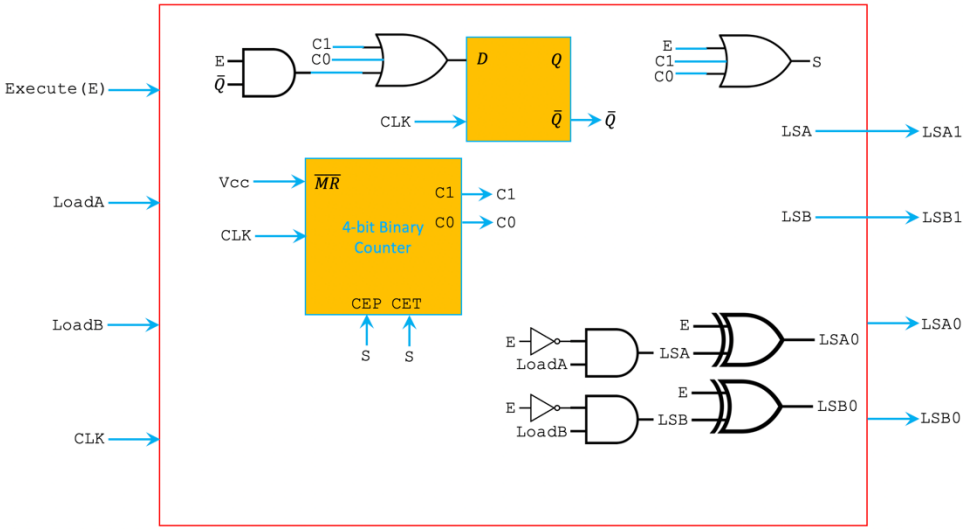


LS1	LS0	Mode
0	0	Hold Current Value
0	1	Shift Right
1	0	Shift Left
1	1	Parallel Load

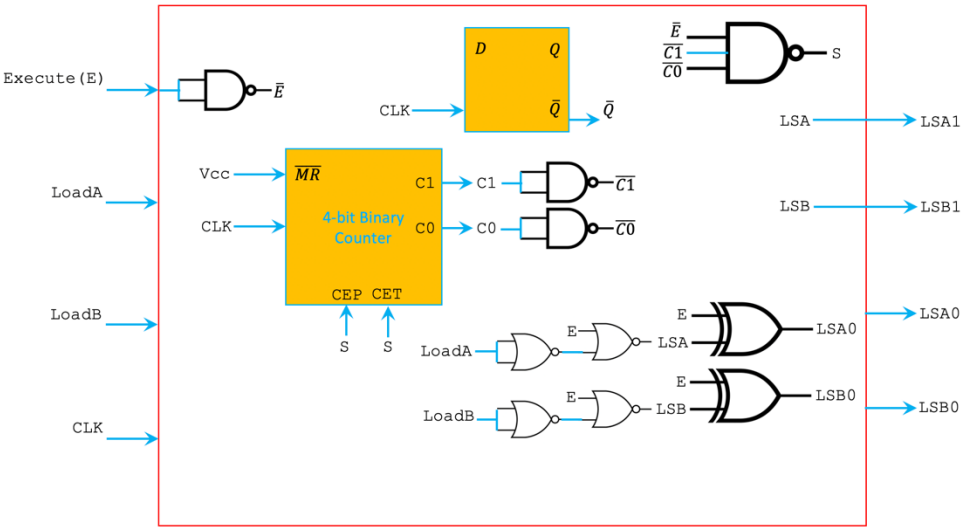
Table 1. Select signal and modes of Shift Register

Both shift registers are identical in design. The CLR pin is connected to power (Vcc) in order for the chip to operate, CLK connected to the clock cycle generated by the DE10, select is mapped to two switches, and the loading functionality for both register A and B are both mapped to the same 4 switches. The registers are connected to 4 LED's (each is connected with an 1kΩ resistors in series) to display the contents to the user.

Control Unit



Replace AND OR NOT with NAND NOR



(Some inputs of the counter are not shown because they are left unconnected)

Since execute and load cannot be high at the same time, we want to have some combinational logic to prevent from happening. (Implemented for both LoadA and LoadB)

Execute (E)	Load (original Load signal)	Load* (new Load signal)
0	0	0
0	1	1
1	0	0
1	1	0

Table 2. Truth Table of Load* signal

$$Load^* = \bar{E}Load$$

According to the data sheet of the SN74LS194E (4-bit bidirectional universal shift register), the LS1 and LS0 signals should have the following functions based on Execute and Load* signals.

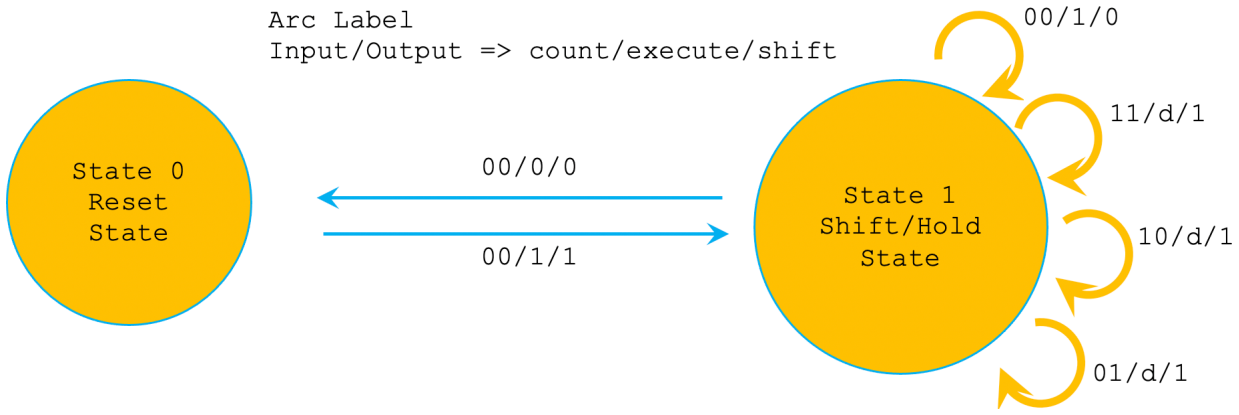
Execute (E)	Load*	LS1 (Load Shift 1)	LS0 (Load Shift 0)	Function
0	0	0	0	Hold Current values of A B
0	1	1	1	Parallel Load values into A B
1	0	0	1	Shift Right
1	1	X(1)	X(0)	Impossible

Table 3. Truth Table for LS1 and LS0 based on inputs Execute and Load*

$$LS1 = Load^*$$

$$LS0 = E \oplus Load^*$$

FSM Diagram



Execute (E)	Q	C1	C0	Register Shift (S)	Q^+	C1	C0
0	0	0	0	0	0	0	0
0	0	0	1	X	X	X	X
0	0	1	0	X	X	X	X
0	0	1	1	X	X	X	X
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	X	X	X	X
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
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. FSM Truth Table

S		C1C0			
		00	01	11	10
EQ	00	0	X	X	X
	01	0	1	1	1
	11	0	1	1	1
	10	1	X	X	X

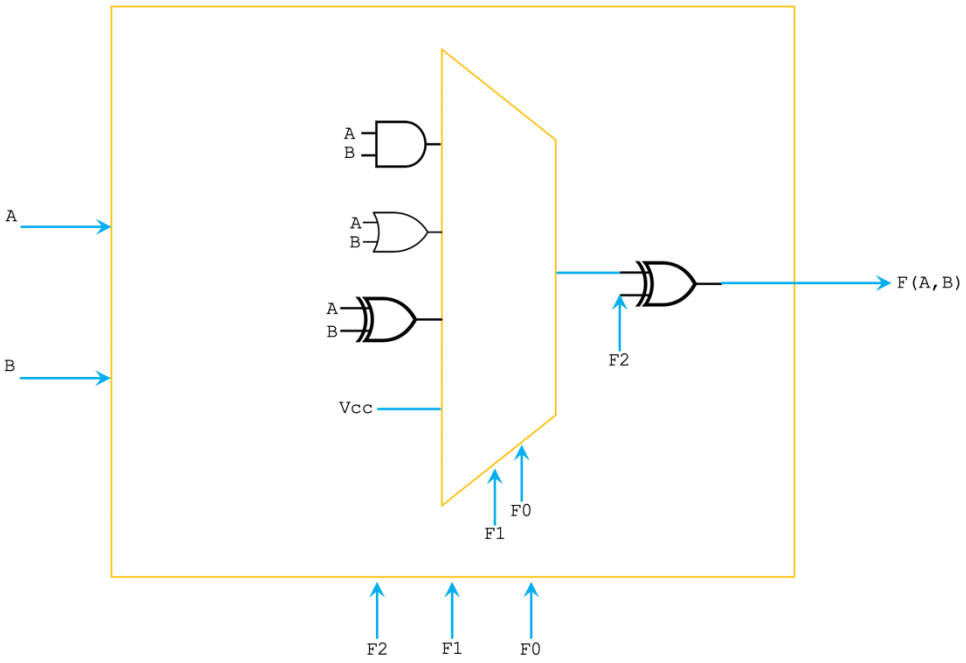
$$S|_{sop} = E\bar{Q} + C1 + C0$$

Q^+		C1C0			
		00	01	11	10
EQ	00	0	X	X	X
	01	0	1	1	1
	11	1	1	1	1
	10	1	X	X	X

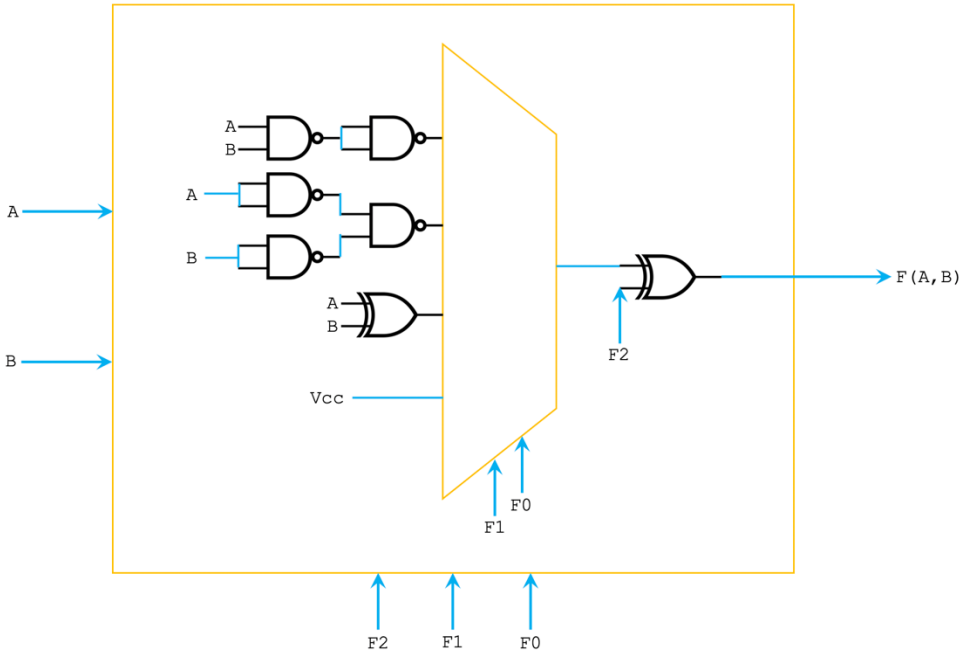
$$Q^+|_{sop} = E + C1 + C0$$

Because execution is primarily dependent on the state of the execute signal, there are many don't cares in C0 and C1. Thus, it seemed more efficient to use a counter in our design in accordance with the mealy machine we based our design on. We check C0 and C1 because those signify 4 cycles, C2 and C3 in combination count up till 15 which is unnecessary. We connected both CEP and CET to shift so that the counter only counts after a shift; this enables us to precisely count 4 shifts.

Computational Unit



Replace AND, OR gates with NAND gates

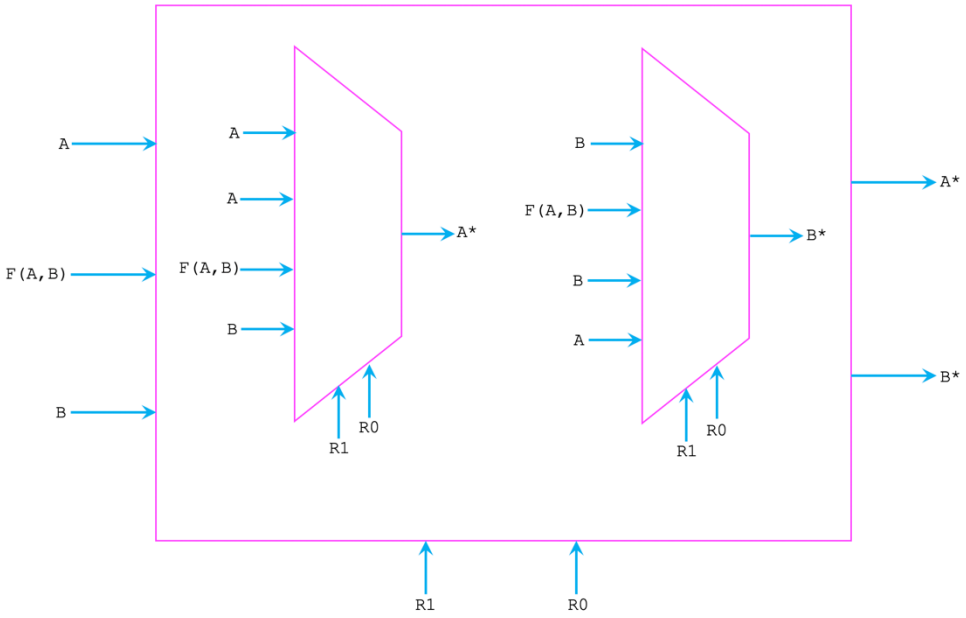


F2	F1	F0	F(A,B)
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0

Table 3. ALU functions and their corresponding select signal F2 F1 F0

The computation unit has the 4 of the 8 functions implemented in logic using a combination of NAND and XOR chips, as well as the power source (Vcc). These 4 functions were connected to a 4x1 MUX whose output was connected to an XOR with the other input of the XOR being the final select signal. The XOR was critical because we needed an element which was 2 input 1 output with one of the inputs having the potential to invert.

Routing Unit



R1	R0	Destination A*	Destination B*
0	0	A	B
0	1	A	F(A,B)
1	0	F(A,B)	B
1	1	B	A

Table 4. Destination A and B based on select signal R1 R0

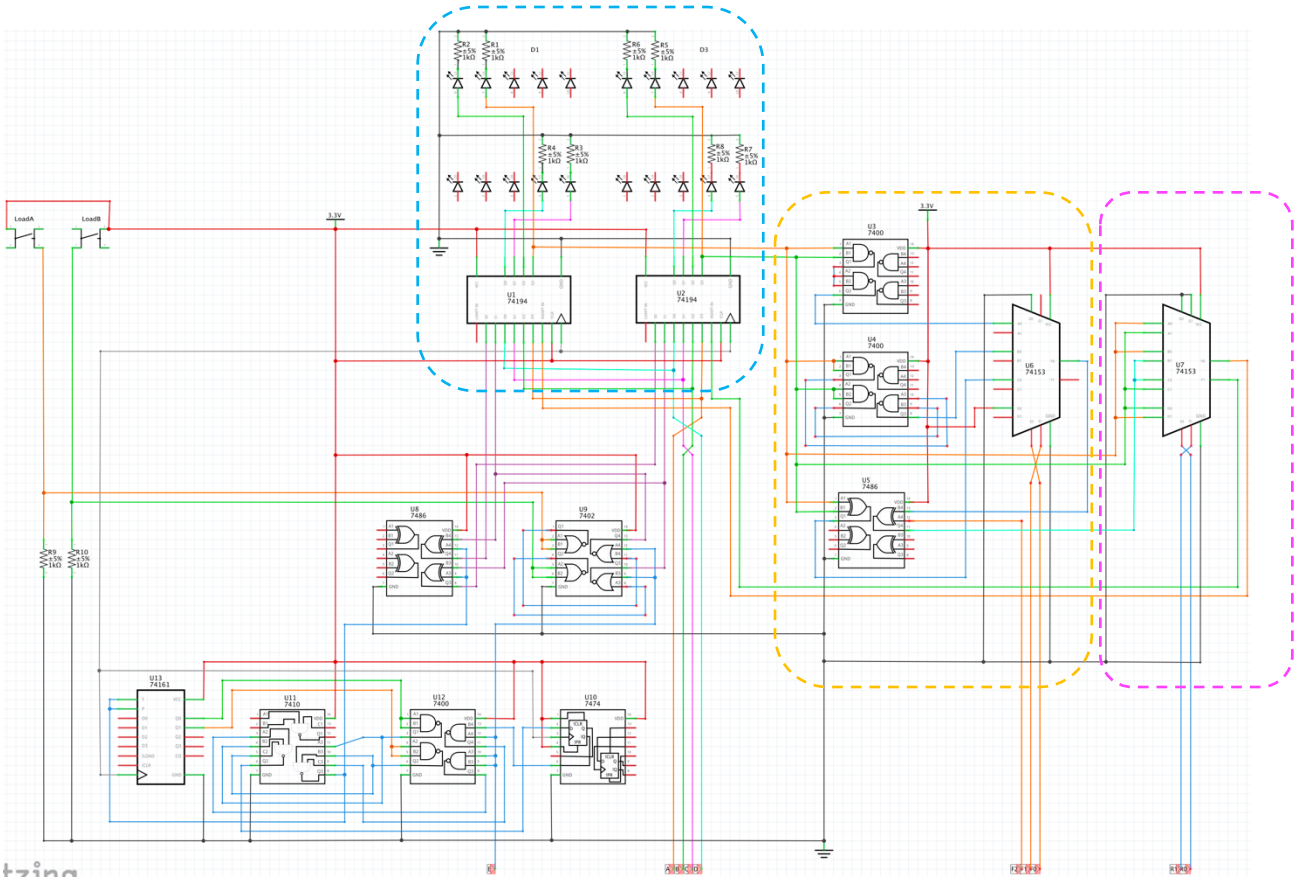
Schematic View

Blue Box – Register Unit

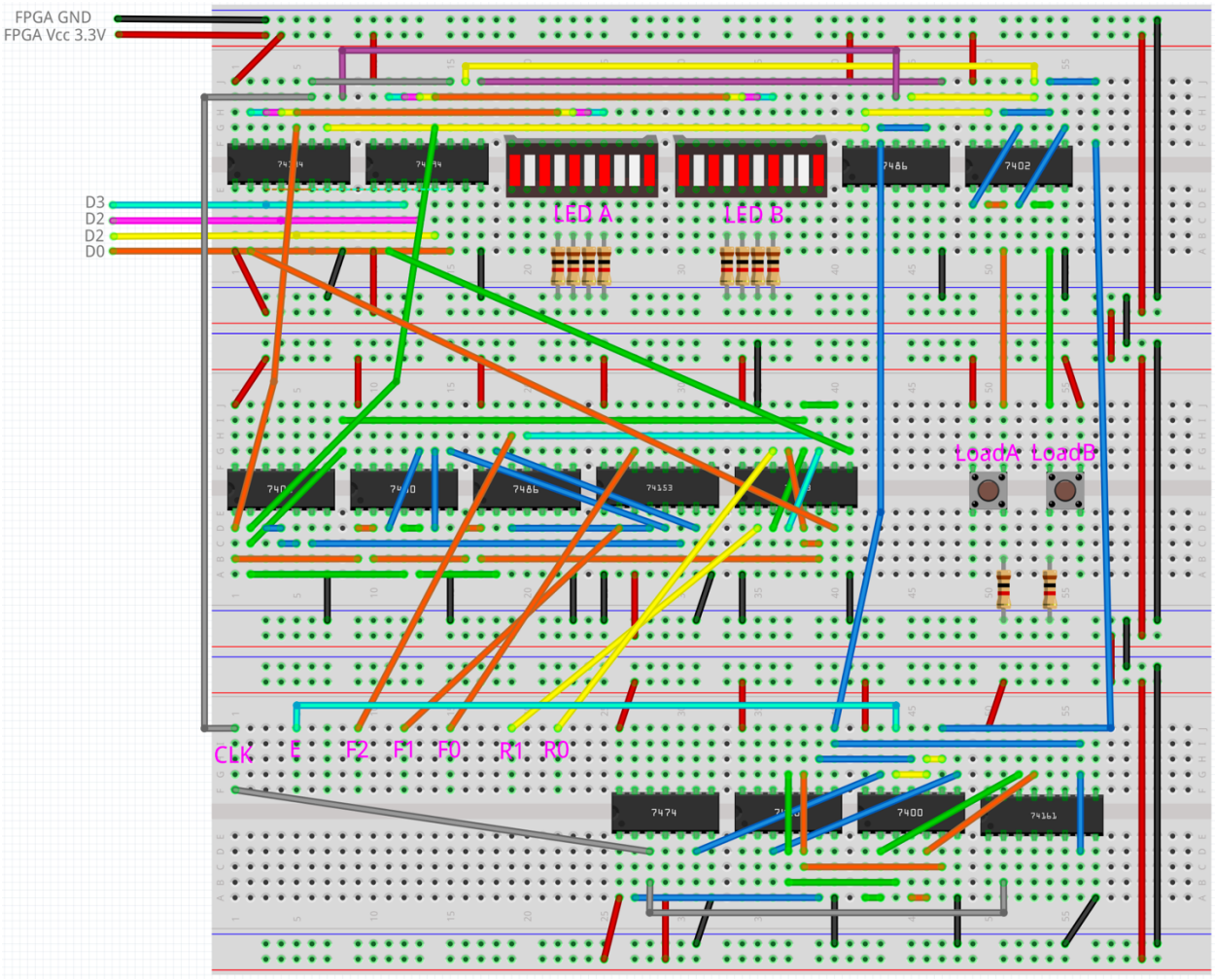
Orange Box – Computation Unit (ALU)

Pink Box – Routing Unit

Rest of the schematic – Control Unit



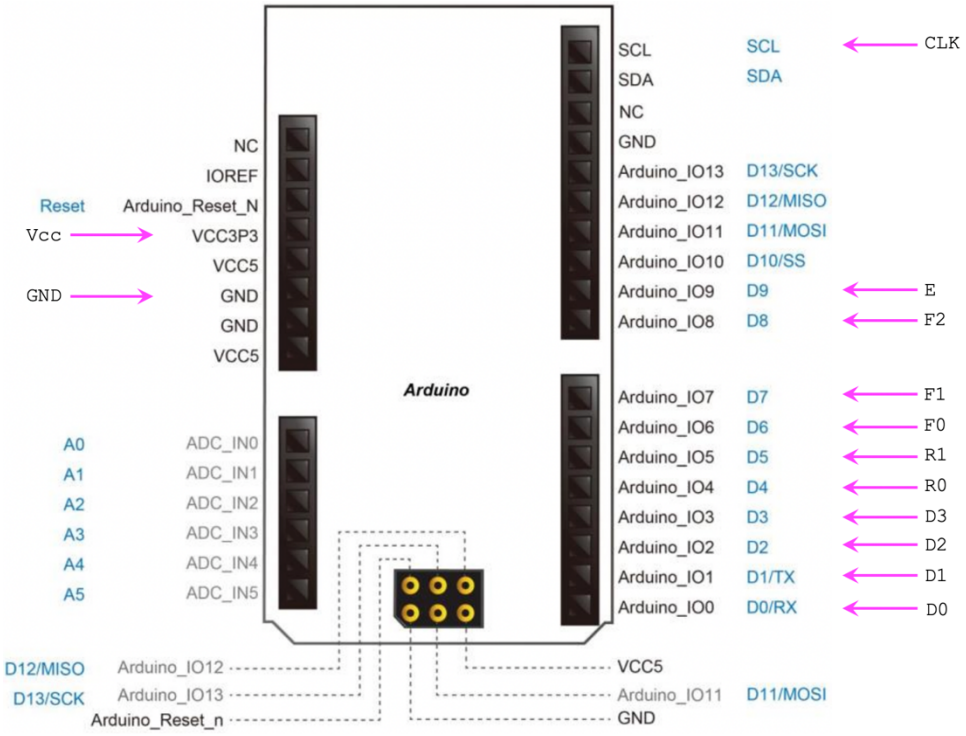
Breadboard View



DE10-Lite board Pin Layout

Signal	Switch	Function
E	DE-10Lite Switch 9	Execute (Start shifting 4 clock cycles)
F2	DE-10Lite Switch 8	ALU Select
F1	DE-10Lite Switch 7	
F0	DE-10Lite Switch 6	
R1	DE-10Lite Switch 5	Routing Select
R0	DE-10Lite Switch 4	
D3	DE-10Lite Switch 3	4-bit Input for Register A and Register B
D2	DE-10Lite Switch 2	
D1	DE-10Lite Switch 1	
D0	DE-10Lite Switch 0	
LoadA	Left PushButton on breadboard	Load 4-Input into Register A
LoadB	Right PushButton on breadboard	Load 4-Input into Register B
CLK	SCL on DE-10Lite	Clock signal for sequential components
Vcc	VCC3P3 on DE-10Lite	Power TTL chips and logic ‘1’
GND	GND on DE-10Lite	GND TTL chips and logic ‘0’

Table 5. Signals and their corresponding switches and functions



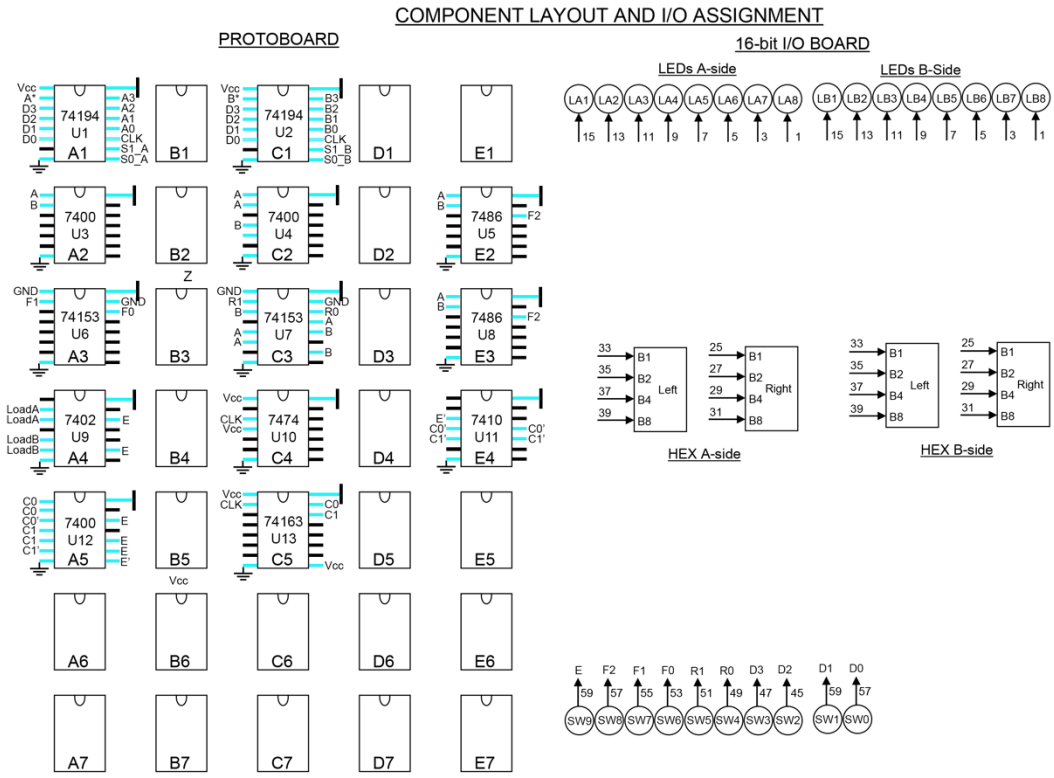
Component Layout

Reference Designator	Type	Description	Place used
U3, U4, U12	SN7400	Quad 2-Input NAND	U3 ALU U4 ALU U12 Control Unit
U9	SN7402	Quad 2-Input NOR	Control Unit
U5, U8	SN7486	Quad 2-Input Exclusive-OR	U5 ALU U8 Control Unit
U11	SN7410	Triple 3-Input NAND	Control Unit
U6, U7	SN74153N	Dual 4-to-1 Multiplexer	U6 ALU U7 Routing Unit
U1, U2	SN74LS194E	4-bit Bidirectional Universal Shift Register	Register Unit
U10	SN7474	Dual D Positive Edge Triggered Flip Flop	Control Unit
U13	SN74163E	4-bit Synchronous Binary Counter	Control Unit

Table 6. TTL chips used to build the 4-bit Logic Processor

Signal	Function
A3	Display on LED A
A2	Display on LED A
A1	Display on LED A
A0	Display on LED A
B3	Display on LED B
B2	Display on LED B
B1	Display on LED B
B0	Display on LED B
A	Register A Shift Right Output
B	Register B Shift Right Output
C1	Output from Counter Chip
C0	Output from Counter Chip

Table 7. Signal references of Component Layout



8-bit Logic Processor on FPGA

Modules

Module: reg_8.sv

Inputs: Clk, Reset, Shift_In, Load, Shift_En, D [7:0]

Outputs: Shift_Out, [7:0] Data_Out

Description: This is a positive-edge triggered 8-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 A and B values which are used in ALU unit.

Module: register_unit.sv

Inputs: clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D

Outputs: A_out, B_out, [7:0] A, [7:0] B

Description: This component contains 8 positive-edge triggered 8-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 the Register Unit which is also the high-level block for register A and register B.

Module: Processor.sv

Inputs: clk, Reset, LoadA, LoadB, Execute, [7:0] Din, [2:0] F, [1:0] R

Outputs: [3:0] LED, [7:0] Aval, [7:0] Bval, [6:0] AhexL, [6:0] AhexU, [6:0] BhexL, [6:0] BhexU

Description: This component contains the register unit, control unit, computation unit, routing unit, and 7Segment display of the 8-bit logic unit.

Purpose: This module is the high-level block for the 8-bit logic unit

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This component is the 7-segment display which can display the hex numbers on the LEDs.

Purpose: This module is used for program debugging because it can display the current values and results in register A and register B.

Module: Control.sv

Inputs: Clk, Reset, LoadA, LoadB, Execute

Outputs: Shift_En, Ld_A, Ld_B

Description: This component is the control unit of the 8-bit logic processor. It controls parallel load values into register A and B, and execution of shifting (8 clock cycles)

Purpose: This module is used as a control unit for the 8-bit logic processor which has sequential designs in it.

Module: compute.sv

Inputs: [2:0] F, A_In, B_In

Outputs: A_Out, B_Out, F_A_B

Description: This component is the computation unit of the 8-bit logic processor (ALU), it can perform 8 different operations.

Purpose: This module is used to compute bitwise logics (based on the 3-bit function select signals) between A and B in 8 clock cycles in order to update the value in either A or B based on the select of routing unit.

Module: Router.sv

Inputs: [1:0] R, A_In, B_In, F_A_B

Outputs: A_Out, B_Out

Description: This component is the routing unit of the 8-bit logic processor, it controls the data sending back to register A and register B inside the register unit.

Purpose: This module is used to control the data sending back to register A and register B based on the 2-bit select signals in the routing unit. The user can keep the original values of A and B or choose to update either A or B.

Module: Synchronizers.sv

Inputs: Clk, d

Outputs: q

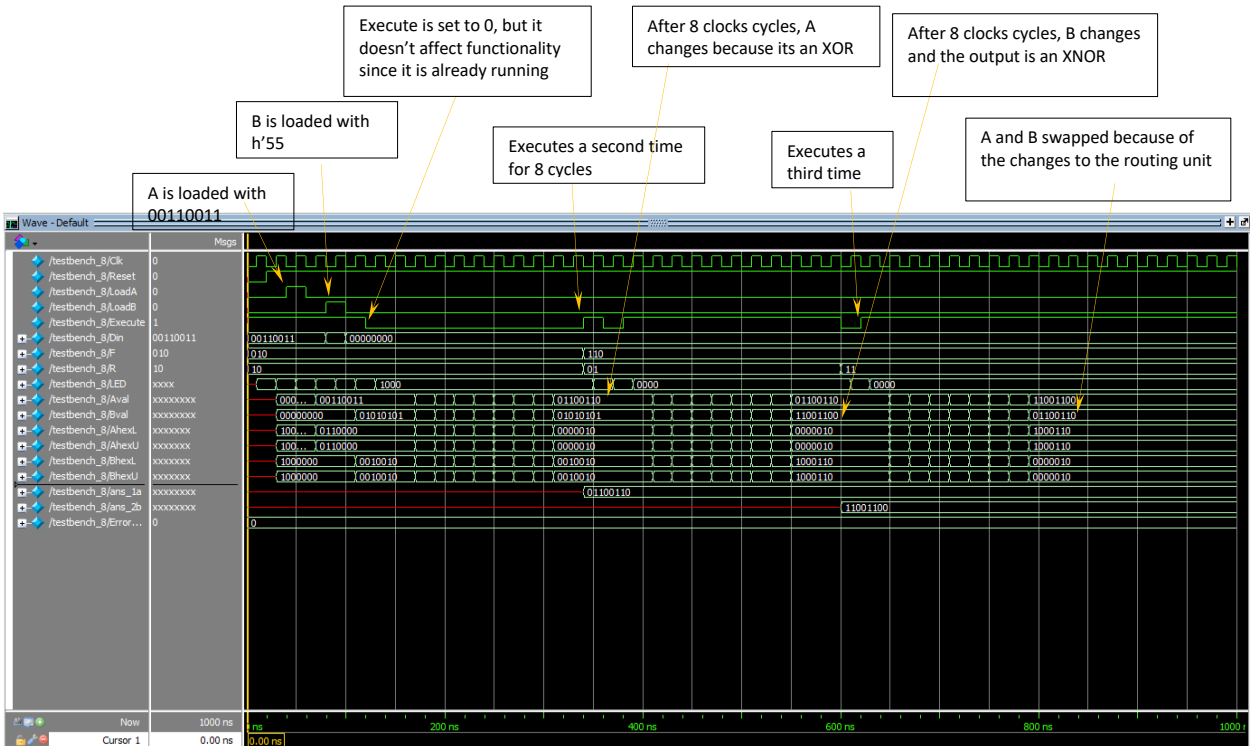
Description: Synchronizer with no reset (for switches/buttons)

Purpose: These are synchronizers required for bring asynchronous signals into the FPGA

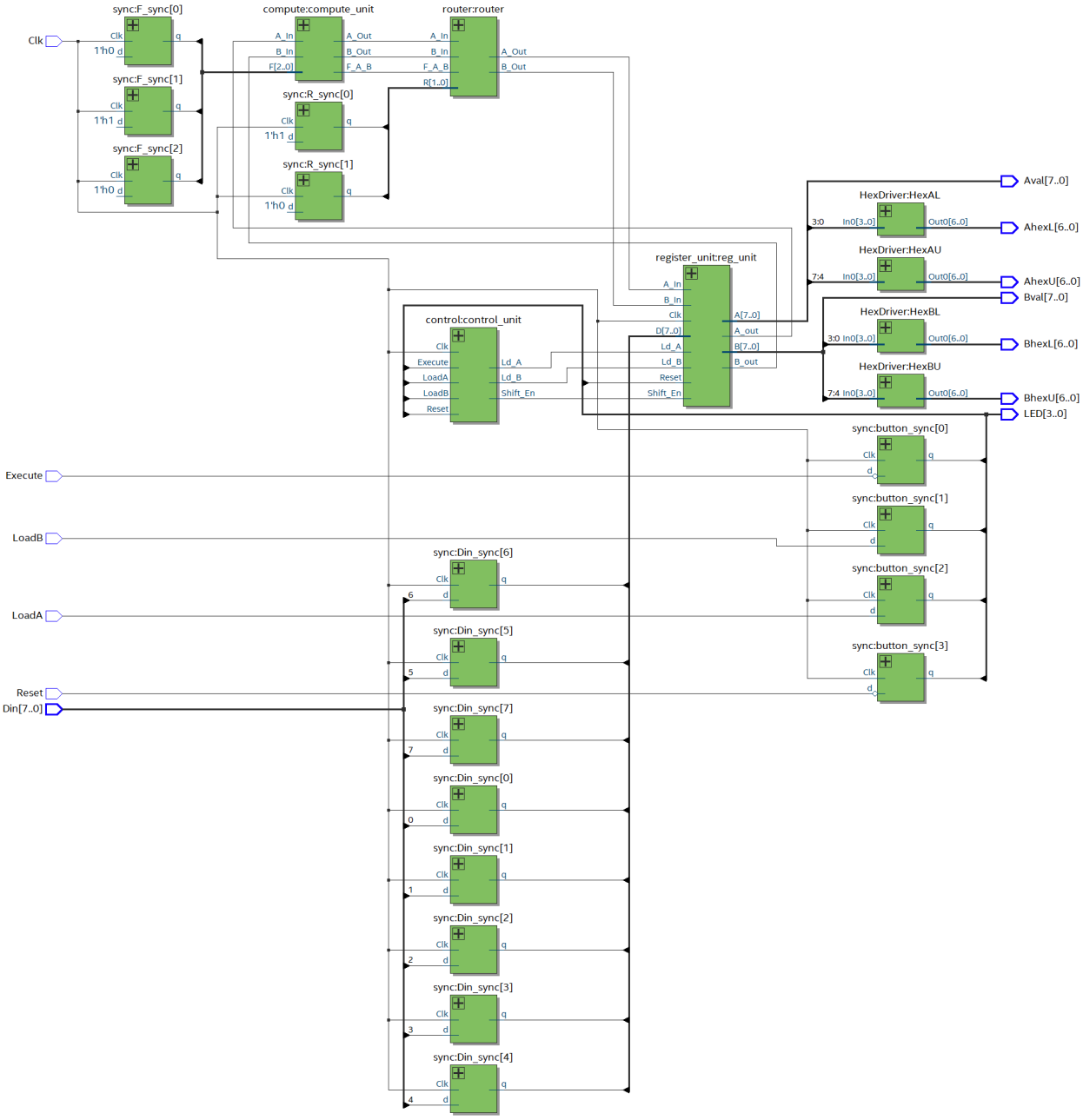
Changes made to the .SV files

1. Change input logic (D) and output logic (Data_Out) in Reg_4.sv from [3:0] to [7:0]
2. Change input logic (D) and output logic (A, B) in Register_unit.sv from [3:0] to [7:0]
3. Change input logic (Din) and output logic (Aval) in Processor.sv from [3:0] to [7:0]
4. Change enum logic from [2:0] to [3:0] and add G, H, I, J inside the curly braces
   F: next_state = G;
   G: next_state = H;
   H: next_state = I;
   I: next_state = J;

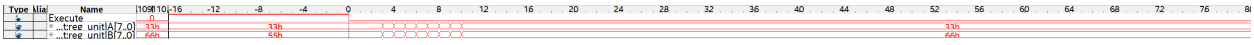
Quartus ModelSim Waveform



RTL Block Diagram



SignalTap ILA trace



1. Run synthesis
2. Compile the project
3. Program the project on the DE-10Lite board
4. Open Tools -> Signal Tap Logic Analyzer
5. Create a new Instance
6. Add Node Execute, Reg_unitA[7:0], Reg_unitB[7:0]
7. Set the trigger condition of Execute to 'Either Edge'
8. Add Node Clk to Clock
9. Click Start Compilation
10. Click Run Analysis
11. Load A with 8'h33 and Load B with 8'h55
12. Click Execute Pushbutton on DE-10Lite board

Post Lab Questions

Our initial design for this lab included an 8x1 MUX for the ALU, and large sequence of purely combinational logic for the control unit. By designing the circuit modularly, we were able to simplify the design by treating each module as a black box with just input and outputs. This way, it was easier to conceptualize the design. Also, the modular design, combined with the unit testing of all the chips enabled us to save time in debugging and pinpoint problems quickly when they arose. For example, we had verified that our ALU, shift registers, and routing unit worked individually, and that all the chips we were using worked as well. When we combined everything together however, there was an issue where the routing of values seemed to be reverse. This was quickly identified to be a wiring mix-up. We faced many issues when testing the individual standalone components like the ALU, Routing unit, etc. Debugging these if they had all been put together would have been a nightmare.

The simplest 2 input 1 output circuit that can optionally invert a signal is an XOR gate. The functionality of the XOR gate is shown below in the truth table. This is useful for this lab because we needed such an element to make out ALU work with a 4x1 MUX. It enabled us to avoid the usage of an 8x1 MUX, and minimized the number of IC's needed.

Select	Input	Output
0	0	0
0	1	1
1	0	1
1	1	0

Input B is kept when Select = 0

Table 6. Truth Table of an XOR gate

Select	Input	Output

Input B is inverted when Select = 1

0	0	0
0	1	1
1	0	1
1	1	0

Table 7. Truth Table of an XOR gate

A Moore machine can be very lengthy because the output is dependent on only the current state. In a Mealy machine, the output is dependent not only on the current state, but also on the inputs, which enable it to have a more compact design. We felt that implementing the Mealy machine was the easier choice in this situation because of the reduction in perceived complexity. Using a counter felt like a better solution because it reduced complexity. The Mealy machine also reduces the problems regarding gate delays because the output of the machine is changed in accordance with the rising edge of a clock signal. The benefits of the Moore machine include easier traceability due to the output being depended only on the current state, as well as perhaps an improvement in speed due to it being more in line with how computers operate.

Model sim is used to test the circuit entirely on Quartus without having to program on the FPGA. A testbench is used to check functionality of the design. On signal tap, the code is actually complied and programmed on to the board. The signals you want to check should be specified. You can manually enter inputs to test, after which, the results of the test are visible on Signal tap. You can only execute once on signal tap as opposed to model sim (multiple test cases).

Conclusion

As stated in the overview the most difficult part of this lab (2.1) was the design of the control unit. “Initially we tried to use pure combinational logic and think through how the necessary functionality would guide the design, however we found this approach to not work out. We instead used the truth table provided in lecture and used the equations derived from said table to come up with the final design.” Unit testing each individual chip used was time consuming, as was figuring out how to make the chips work. A particular conundrum we faced was choosing between the 74194 and the 74195 chips for our shift register. After trying to operate both, we found it much easier to use the 74194. Since there were so many parts to this design, cable(wire) management served to make the design look nice, but more importantly made it easier to understand and as a consequence easier to debug. While this lab was frustrating at times (mainly in the control unit design), completing it successfully was incredibly rewarding to the both of us.

For lab 2.2, the most difficult part was getting Quartus to work and learning how to use the tool on a basic level. We learned an overview of hardware programming, as well as testing methodologies used to efficiently identify and correct bugs. We learned 2 different modes of debugging with SignalTap and ModelSim, hardware testing physically and testing purely in the software respectively. Lab 2.2 gave us a big picture overview of how computers are actually designed and built in industry, which complemented lab 2.1 which provided us with a solid understanding of basic processor design from ground up.