ECE 385 Lab Report 2

Experiment #2: A Logic Processor

Spring 2022

Trusha Vernekar & Soham Manjrekar

Dohun Jeong (DJ)

tnv2 & sohammm2

Introduction

Lab 2 was divided into 2 parts: A Hardware design of a bit-serial logic operation processor and a Software design of the same processor extended to 8 bits. In Lab 2.1, the Hardware lab, we built a 4-bit logic operation processor supporting 8 operations: AND, OR, XOR, 1111, NAND, NOR, XNOR, and 0000. In Lab 2.2, we manipulated code to build a 4-bit logic processor provided to us to create the same processor for 8-bits instead. The inputs and outputs were stored in 2 registers and were displayed using the seven-segment display LEDs.

Operation of the logic processor

Describe the sequence of switches the user must flip to load data into the A and B registers.

In order to load data in registers A and B, all switches must be off initially. We had 4 switches indicating the data bits D3-D0 for register A and the same for register B. Once the user knows that data must go in each of the registers, the data must be manually entered into each register by turning on/off the switches. D3-D0 signifies the data bits, D3 is the most significant bit and D0 is the least significant bit. Once D3-D0 is set for both Register A and Register B, Load A is turned on to load the data in Register A, and Load B is turned on to load the data in Register B. Once this is done, Load A and Load B must be turned off before any further computation.

Describe the sequence of switches the user must flip to initiate a computation and routing operation.

Our logic processor can compute 8 different processes and the way one of these processes can be chosen is by choosing bits F2, F1, and F0. After the user decides which process to compute, the bits F2-F0 for the corresponding process (see table 1 below) are turned on manually by switches. After these are set, the switches for F2, F1, and F0 are left on. The user must now decide which register should hold the value of the resulting computation. The same can be chosen by selecting the bits R1 and R0 in the routing unit. The routing unit will decide if each register is going to hold its initial value, the computed value, or the value of the other register. The same will be shown by the LEDs connected. Once the user decides this, bits R1 and R0 must be set for the same (see table 2 below) by manually operating the switches. Once R1 and R0 are set, they must be left on. The user must now press the Execute button to initiate the computation and routing.

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

Table 1: Computational unit

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 2: Routing unit

Written description of the logic processor

In Lab 2.1, we had 4 different units: Register Unit, Computational Unit, Routing Unit, and Control Unit. Each unit had a unique task in the entire logic processor.

Register Unit

The main task of this unit was to store the two inputs in Registers A and B and pass them along to the computational unit. This unit consists of 2 4-bit Bidirectional Universal Shift Register (74194A/E), one for A and B. Once Load A/Load B is on for each register, each bit from the data bits D3-D0 are added to each register through the parallel input pins 3 to 6. These bits in both the registers are displayed on 4 LEDs each coming from outputs Q_A to Q_D. Since our computational unit computes the processes for each bit, the initial inputs loaded in the register through parallel load must be shifted into the computational unit bit by bit, with the least significant bit (Q_D) shifted first. Hence, we had to initiate a right shift on the bits. To do this, we connected Load A/Load B to S1 in both the registers, and the Shift bit was connected to S0 in both the registers. This Shift bit is calculated by performing the OR function on Load A/Load B and S which is the output from the control unit. We did this because the shift is initiated once S1 is Low and S0 is High. The result after computation might be stored in one of the register units depending on the signals R1 and R0 in the routing unit.

Computational Unit

The main task of this unit is to compute 8 bitwise operations on a single bit coming in from the register unit (the least significant bit that is coming out of Q_D after the right shift of the bits). This unit consists of one 8:1 Multiplexer, one Quad 2-input NAND, one Quad 2-input NOR, and one Hex Inverter. Since there were 1 of 8 bitwise operations to be selected, an 8:1 MUX was used depending on the F2, F1, and F0 signals passed by the user (via switches) into S2, S1, and S0 of the chip. The main operations such as AND, OR, XOR, NAND, NOR, and XNOR were computed using multiple NAND gates. The NAND operation was done using one NAND gate. The AND operation was done by inverting the output from the NAND by using the Hex Inverter. The NOR operation was computed using 1 NOR gate. The OR operation was done by inverting the output from the NOR gate. The XOR operation was computed using 3 NAND gates (refer to figure 1). The XNOR was computed by inverting the output from the XOR system. For operation 1111, the input to the MUX for this operation was set to high. For operation 0000, the input to the MUX for this operation was set to low. On the chip itself, the strobe bit is set to low.

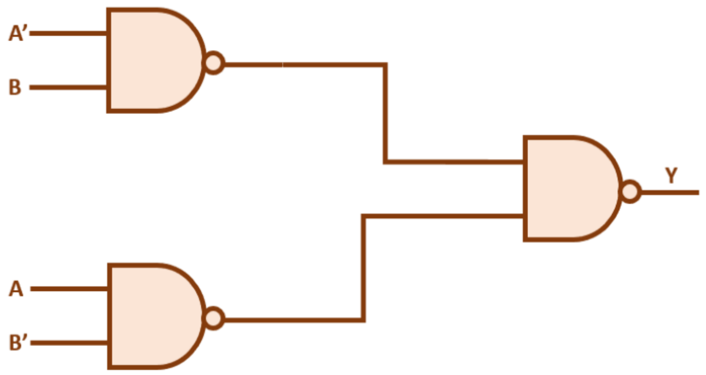


Figure 1: NOR gate using NAND gates

Routing Unit

The main task of this unit is to take A and B from the registers and the calculated F from the computational unit and route it according to the R-value. This unit consists of one dual 4:1 MUX, that chooses both the NewA and NewB values. The select bit was chosen depending on the R1 and R0 signals passed by the user (via switches) into B and A of the chip. If the R1,R0 signal is 0,0 then NewA = A and NewB = B, if the R1,R0 signal is 0,1 then NewA = A and NewB = F, if the R1,R0 signal is 1,0 then NewA = F and NewB = B, if the R1,R0 signal is 1,1 then NewA = B and NewB = A. The NewA and NewB values are routed from the chip back to the Shift Registers, NewA is routed into the Serial Right pin of the A Shift Register, while NewB is routed into the Serial Right pin of the B Shift Register. The strobe bits for both of the 4:1 MUXes used were set to 0.

Control Unit

The main task of this unit is to send the signal S which decides if the bits in the register unit will be shifted or not. In this unit, we needed the signal S to be high for 4 counts per button press of execute (so that the 4 bits in the register unit are shifted). Hence, a 4-bit Synchronous Up-Down Counter was used to initiate this. The control unit turns the signal S low once the 4 counts are over automatically. This unit is designed based on a Mealy State Machine which will be discussed later. In this state machine, the bit S is set to once the execute button is pressed regardless of any change. Based on the truth table and K-maps shown later in this report, the next states S^+ and Q^+ are decided. The next state Q^+ is passed into a Flip-Flop for further computations. Using the equations calculated from the K-Map provided later, the bit S is calculated. This signal S is passed in the ENP of the Up-Down Counter and the ENT is set to high. Due to this, the count stops after 4 cycles.

High-Level block diagram of logic processor

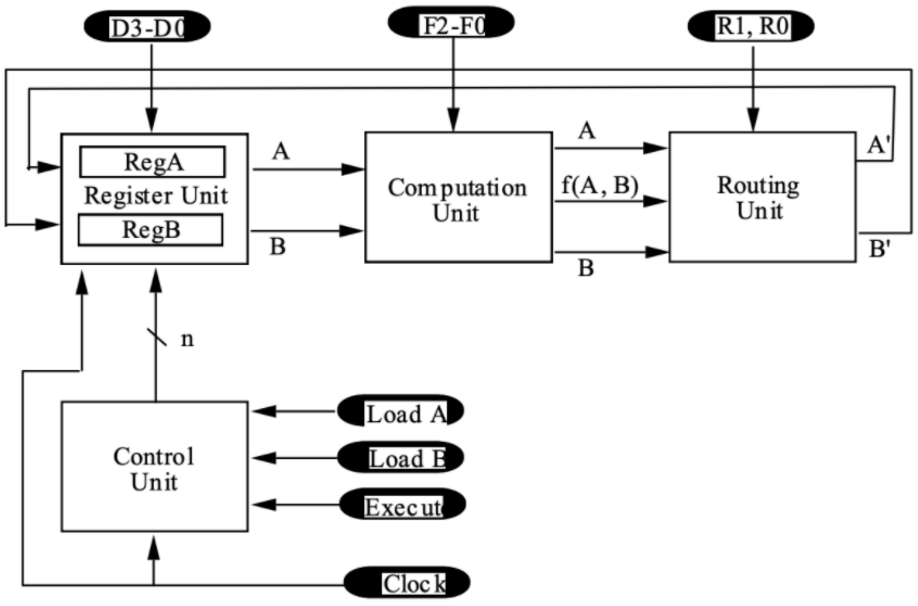


Figure 2: Block Diagram

State Machine Diagram of logic processor

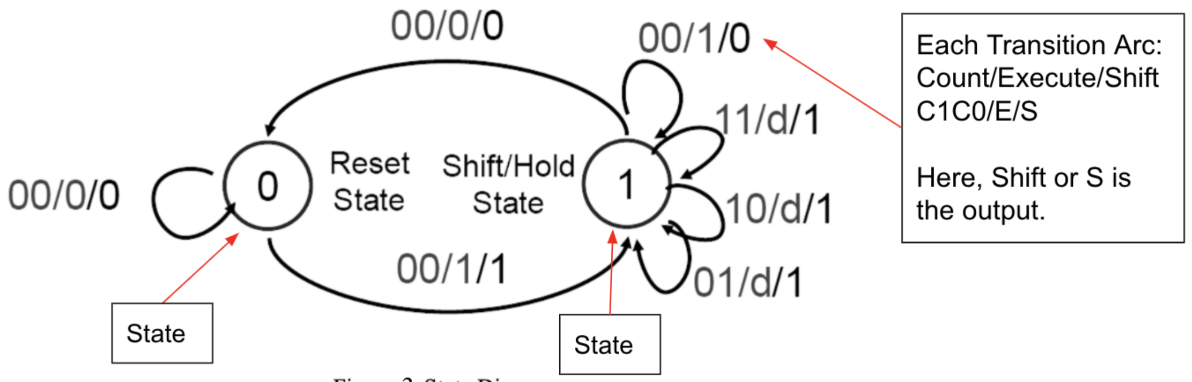


Figure 3: State Diagram

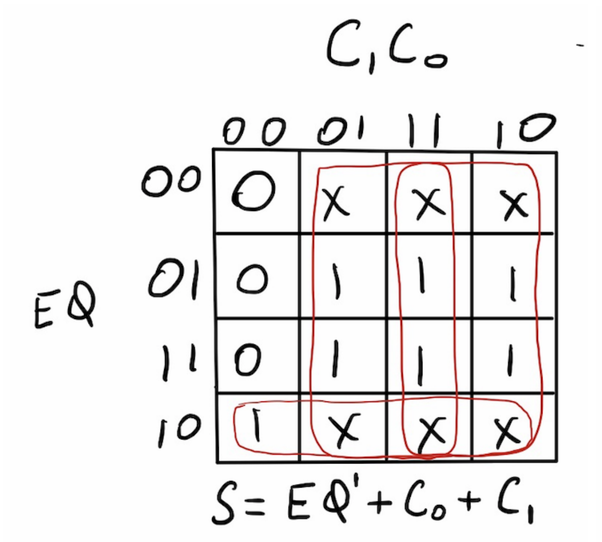
Transition Arc	Description of Arc	State
00/0/0	Counter is 0, Execute and S is Low	Hold in Reset State
00/1/1	Counter is 0, Execute and S is High	Shift & Compute Cycle 1
01/d/1	Counter is 1, Execute is don’t care and S is High	Shift & Compute Cycle 2
10/d/1	Counter is 2, Execute is don’t care and S is High	Shift & Compute Cycle 3
11/d/1	Counter is 3, Execute is don’t care and S is High	Shift & Compute Cycle 4
00/1/0	Counter is 0, Execute is High and S is Low	Hold State until E is Low
00/0/0	Counter is 0, Execute and S is Low	Return to Reset State

Table 3: Description of states in the Mealy State Machine Diagram (Figure 3)

Karnaugh Maps and Truth Tables

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

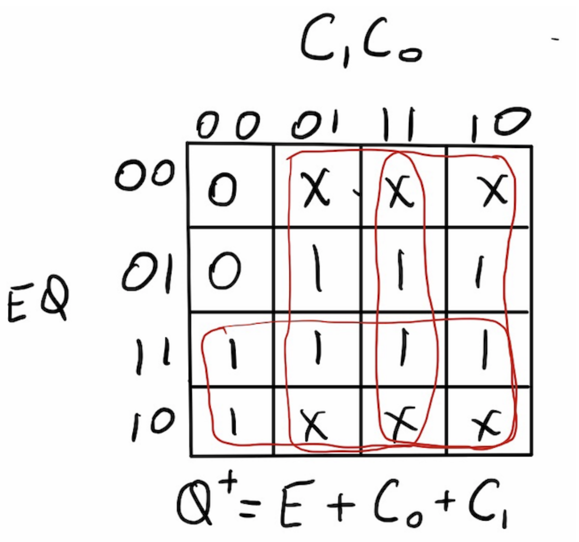
Table 4: Truth Table for Control Unit



$$\begin{array}{c|cccc} EQ \backslash C_1C_0 & 00 & 01 & 11 & 10 \\ \hline 00 & 0 & X & X & X \\ 01 & 0 & 1 & 1 & 1 \\ 11 & 0 & 1 & 1 & 1 \\ 10 & 1 & X & X & X \end{array}$$

$S = EQ' + C_0 + C_1$

Figure 5: K-Map for S



$$\begin{array}{c|cccc} EQ \backslash C_1C_0 & 00 & 01 & 11 & 10 \\ \hline 00 & 0 & X & X & X \\ 01 & 0 & 1 & 1 & 1 \\ 11 & 1 & 1 & 1 & 1 \\ 10 & 1 & X & X & X \end{array}$$

$Q^+ = E + C_0 + C_1$

Figure 6: K-map for Q^+

Circuit Design Choice

For Lab 2.1, in the computational unit, we decided to use Quad 2-input NAND gate chips to compute our 8 operations. We used one chip to represent the NAND operation along with the XOR operation because this would help us reduce the number of chips used. We also used an 8:1 MUX to select which operation to be computed because there were 8 operations that are represented by a 3-bit value of F2 F1 and F0. This made it easier for us to place the F2 F1 and F0 signals coming from the switches instead of using 2 4:1 MUX. Along with this, instead of representing each computation separately using gates, we only represented NAND, NOR, and XOR. To compute AND, OR and XNOR, we inverted the values of NAND, NOR, and XOR. This allowed us to use a lesser number of chips on our breadboard.

In the register unit, we started off by using the 74195E 4-bit Bidirectional Universal Shift Register to store the data in values and to shift the bits. However, we soon realized that controlling the shifting of the bits on the 74194A/E 4-bit Bidirectional Universal Shift Register was more intuitive and hence, we used the same in our register unit.

In the control unit, we had 2 choices of counters: The Up Down Counter and the Binary Counter. We decided to use the Up-Down counter in our circuit because it gave us better control to start and stop the counter when we needed it to.

Detailed Circuit Schematic

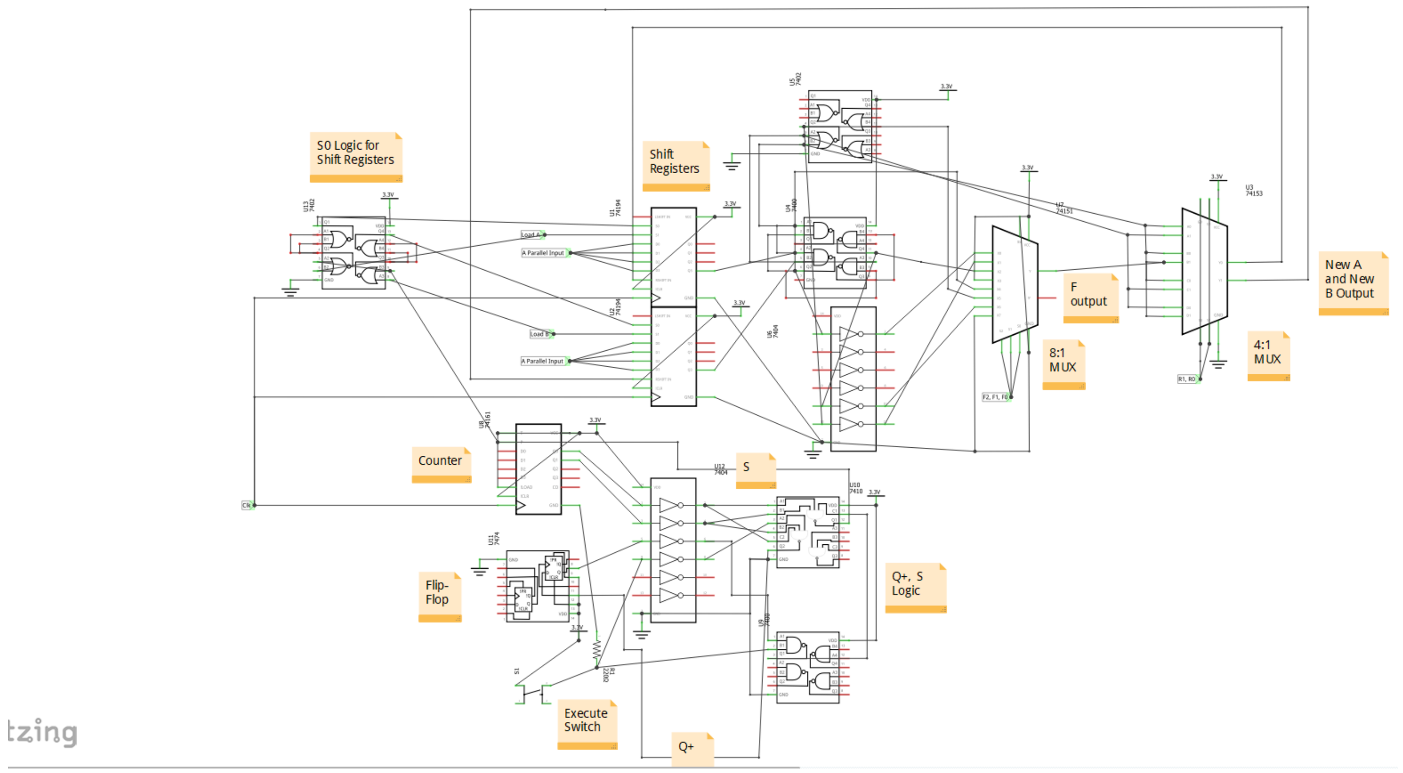


Figure 7: Circuit Schematic

Breadboard view

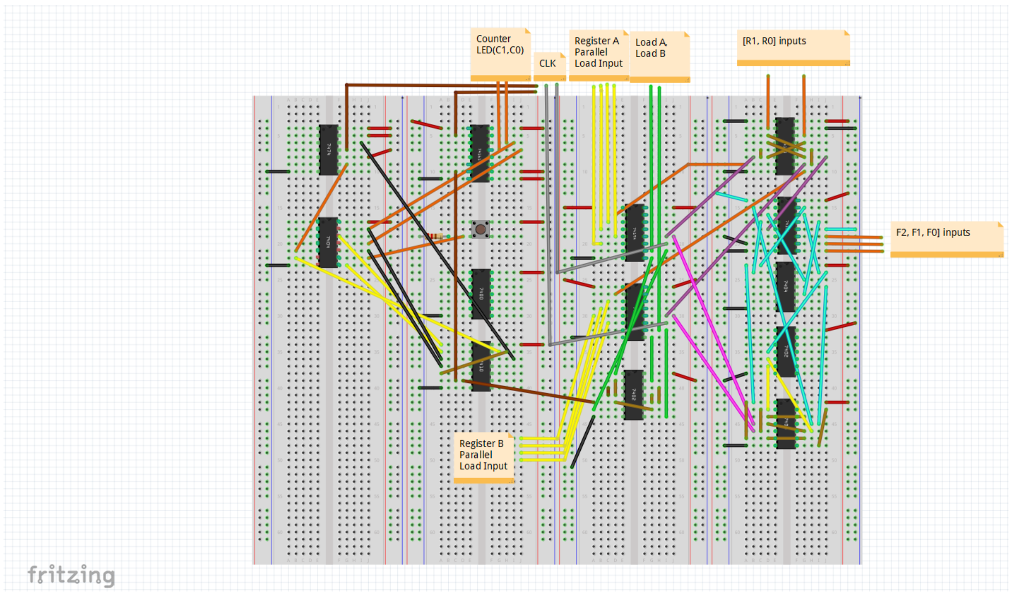


Figure 8: Breadboard view

8-bit logic processor on FPGA code alterations

- Processor
  - Input Din and outputs Bval and Aval extended from [3:0] → [7:0]
  - SignalTap logic:
    - logic [2:0] F; assign F = 3'b010;
    - logic [1:0]; assign R = 2'b10;
  - More HexDrivers to view upper nibble
    - .In0(A[7:4])
    - .In0(B[7:4])
  - Din_sync extended to [7:0]
- Register_unit
  - Input D and outputs A and B extended to [7:0]
- Control
  - curr_state and next_state extended to [3:0] to have states {{A, B, C, D, E, F, G, H, I, J} where J is the last state rather than F for 4 bits.
  - The finite state machine is extended to J as well
    - F :   next_state = G;
    - G :   next_state = H;
    - H :   next_state = I;
    - I :   next_state = J;
    - J :   if (~Execute)   next_state = A;
- Reg_4
  - input D and output data_out extended to [7:0]
  - Reset data_out extended to 8'h0 instead of 4'h0
  - Shift Enable data_out in the concatenation of Shift In and Data_Out extended to [7:1]
- Router, Compute, Synchronizers, HexDriver, and Testbench_8 sv files were unchanged

RTL block diagram

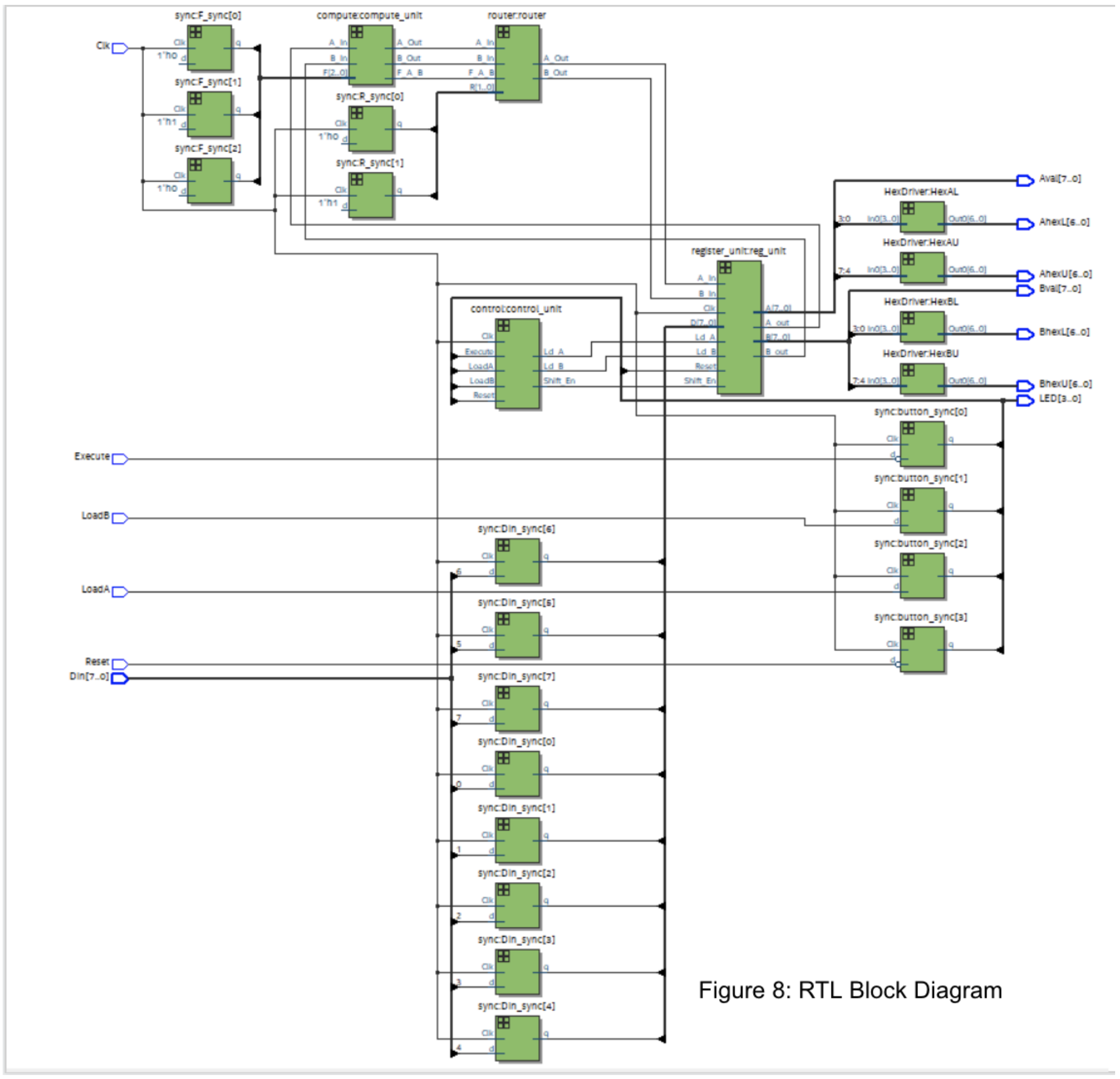
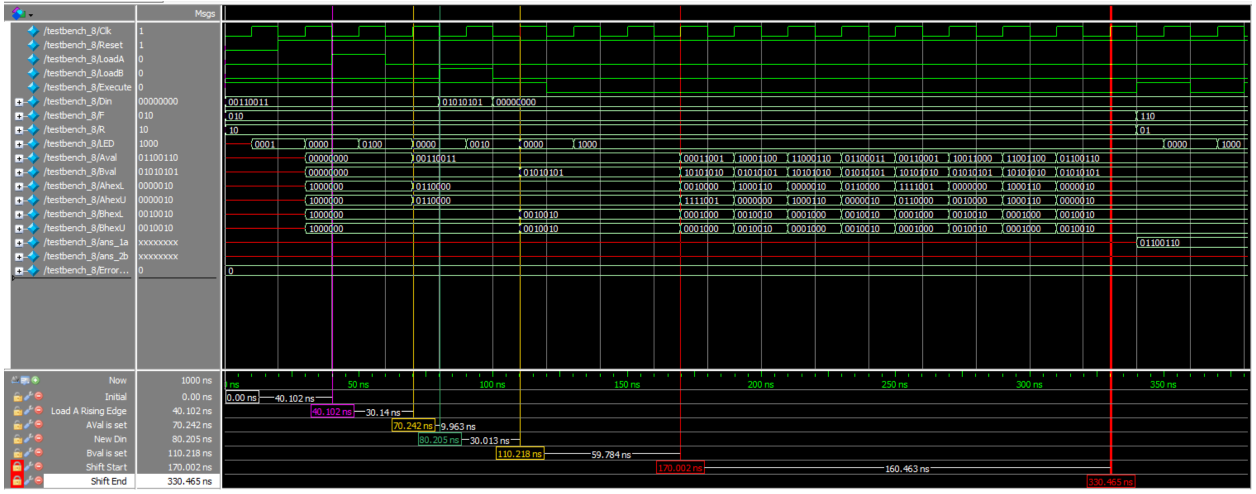


Figure 9: RTL Block Diagram

RTL Simulation

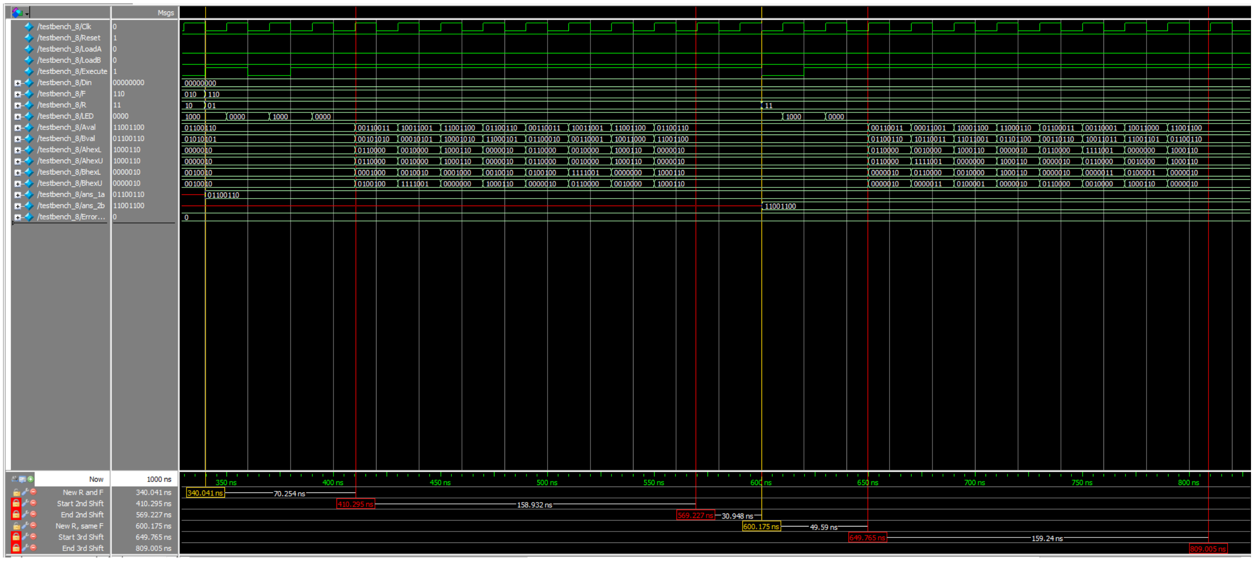


	Msgs
/testbench_8/Clk	1
/testbench_8/Reset	1
/testbench_8/LoadA	0
/testbench_8/LoadB	0
/testbench_8/Execute	0
/testbench_8/Din	00000000
/testbench_8/F	010
/testbench_8/R	10
/testbench_8/LED	1000
/testbench_8/Aval	01100110
/testbench_8/Bval	01010101
/testbench_8/AhexL	0000010
/testbench_8/AhexU	0000010
/testbench_8/BhexL	0010010
/testbench_8/BhexU	0010010
/testbench_8/ans_1a	xxxxxxxx
/testbench_8/ans_2b	xxxxxxxx
/testbench_8/Error…	0

Now	1000 ns
Initial	0.00 ns
Load A Rising Edge	40.102 ns
AVal is set	70.242 ns
New Din	80.205 ns
Bval is set	110.218 ns
Shift Start	170.002 ns
Shift End	330.465 ns

Figure 10.A: RTL Simulation Initially

- At the start of the simulation Din, R, and F are set, the Clk starts and Reset is triggered,
- Load A is then toggled (denoted by the purple cursor on the waveform), and at the next rising edge of the clock (given by the first orange cursor) Din is stored in Aval.
- The value of Din is changed (shows at the green cursor), after that Load B is toggled and similar to Load A, at the next rising edge of the clock (given by the second orange cursor) Din is stored in Bval.
- Now Aval and Bval have values and execute is triggered, the bits of Aval and Bval begin to shift directed by the R and F values, the shifting can be seen between the two red cursors
- After the shifting is over the result of the operation is stored in ans_1a



```
                              Msgs
/testbench_8/Clk              0
/testbench_8/Reset            1
/testbench_8/LoadA            0
/testbench_8/LoadB            0
/testbench_8/Execute          1
/testbench_8/Din              00000000
/testbench_8/F                110
/testbench_8/R                11
/testbench_8/LED              0000
/testbench_8/Aval             11001100
/testbench_8/Bval             01100110
/testbench_8/AhexL            1000110
/testbench_8/AhexU            1000110
/testbench_8/BhexL            0000010
/testbench_8/BhexU            0000010
/testbench_8/ans_1a           01100110
/testbench_8/ans_2b           11001100
/testbench_8/Error...         0

                  Now         1000 ns
New R and F                   340.041 ns
Start 2nd Shift               410.295 ns
End 2nd Shift                 569.227 ns
New R, same F                 600.175 ns
Start 3rd Shift               649.765 ns
End 3rd Shift                 809.005 ns
```

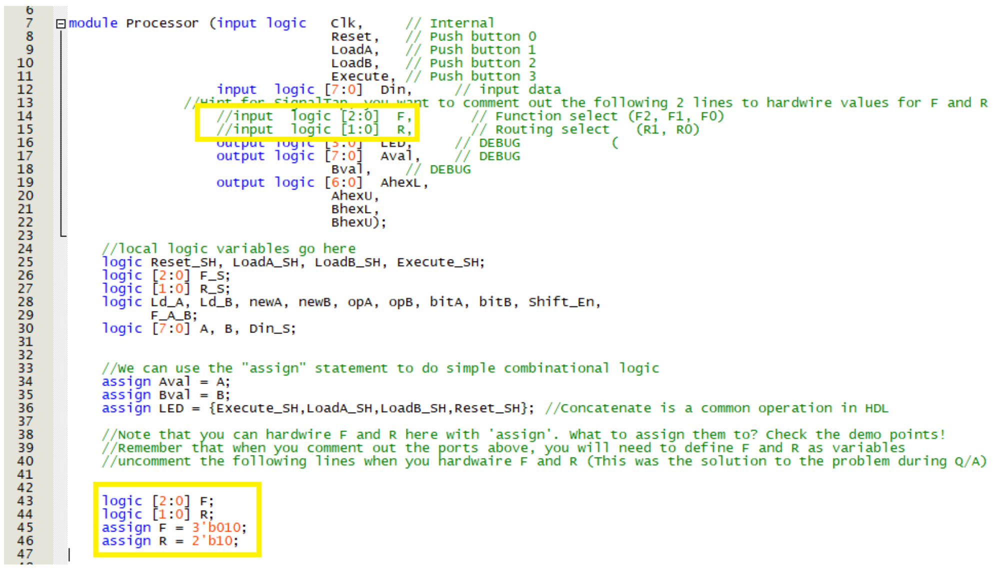
Figure 10.B: RTL Simulation after Execute is pressed

- After the first case in the testbench, the values of R and F are changed and Execute is toggled (denoted by the first yellow cursor),
- The bits of Aval and Bval begin to shift directed by the new R and F values (seen between the first two red cursors)After the shifting is over the result of the operation is stored in ans_2b
- Finally, R is changed to 11, which means the values in Register A and B should swap. (seen at the yellow cursor)
- Once again, after Execute is toggled shifting begins and the values of Aval and Bval have swapped as seen at the beginning and end of the shifting process (between the last two red cursors)

SignalTap ILA Trace Procedure

To produce a trace there are a few steps that need to be taken before opening SignalTap in Quartus.

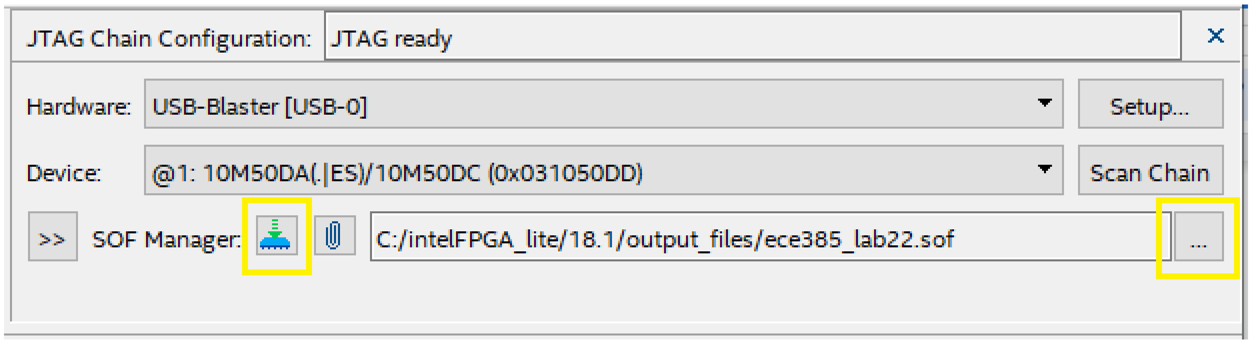
1. First you must edit the Proccessor.sv file and comment out both F and R as input logic from the module. Since there are not enough switches on the FPGA we will have to hardwire these values, for our demonstration F = 010, and R = 10. These changes are shown in the image below.



```
module Processor (input logic   Clk,     // Internal
                                Reset,   // Push button 0
                                LoadA,   // Push button 1
                                LoadB,   // Push button 2
                                Execute, // Push button 3
                  input  logic [7:0]  Din,      // input data
             //Hint for SignalTap, you want to comment out the following 2 lines to hardwire values for F and R
                  //input  logic [2:0]  F,        // Function select (F2, F1, F0)
                  //input  logic [1:0]  R,        // Routing select   (R1, R0)
                  output logic [3:0]  LED,      // DEBUG           (
                  output logic [7:0]  Aval,     // DEBUG
                                Bval,    // DEBUG
                  output logic [6:0]  AhexL,
                                AhexU,
                                BhexL,
                                BhexU);

        //local logic variables go here
        logic Reset_SH, LoadA_SH, LoadB_SH, Execute_SH;
        logic [2:0] F_S;
        logic [1:0] R_S;
        logic Ld_A, Ld_B, newA, newB, opA, opB, bitA, bitB, Shift_En,
              F_A_B;
        logic [7:0] A, B, Din_S;


        //We can use the "assign" statement to do simple combinational logic
        assign Aval = A;
        assign Bval = B;
        assign LED = {Execute_SH,LoadA_SH,LoadB_SH,Reset_SH}; //Concatenate is a common operation in HDL

        //Note that you can hardwire F and R here with 'assign'. What to assign them to? Check the demo points!
        //Remember that when you comment out the ports above, you will need to define F and R as variables
        //uncomment the following lines when you hardwaire F and R (This was the solution to the problem during Q/A)

        logic [2:0] F;
        logic [1:0] R;
        assign F = 3'b010;
        assign R = 2'b10;
```

Figure 11: Screenshot of the Processor.sv file highlights the lines of code to comment and uncomment to run signal tap

2. Next, compile your code and open the SignalTap Logic Analyzer in Quartus. Upload the .sof file of the project to the FPGA.



JTAG Chain Configuration: JTAG ready

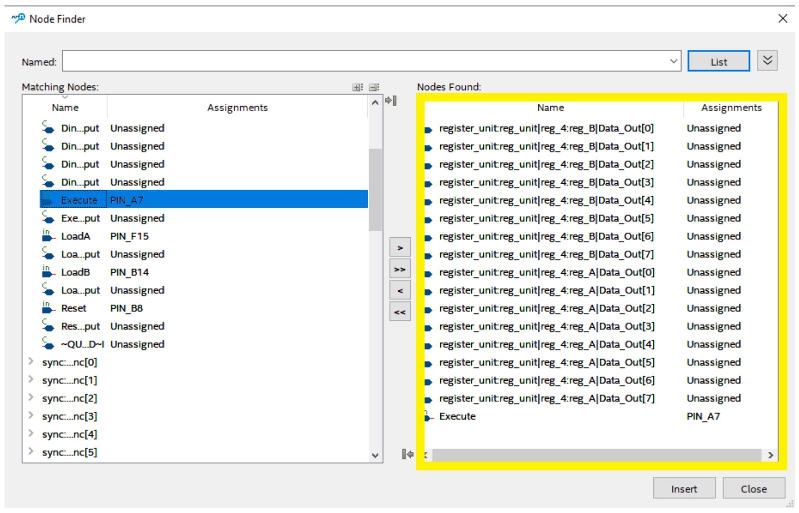
Hardware: USB-Blaster [USB-0]

Device: @1: 10M50DA(.|ES)/10M50DC (0x031050DD)

SOF Manager: C:/intelFPGA_lite/18.1/output_files/ece385_lab22.sof

Figure 12: Screenshot of .sof upload to FPGA within SignalTap

3. After that, to set up a SignalTap instance, open the node finder and add the Register output signals from both A and B, add the Execute signal, and add the Clk signal to the Signal Configuration in setup. You may optionally group the A and B outputs together.


Node Finder

Named: | List

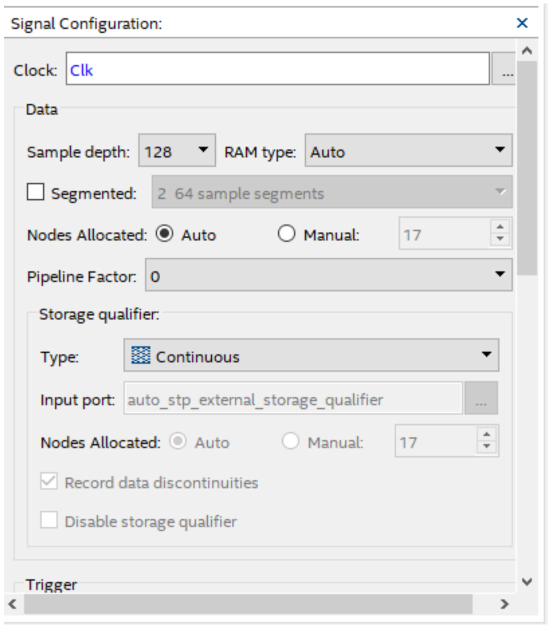
Matching Nodes:

Name	Assignments
Din...put	Unassigned
Din...put	Unassigned
Din...put	Unassigned
Din...put	Unassigned
Execute	PIN_A7
Exe...put	Unassigned
LoadA	PIN_F15
Loa...put	Unassigned
LoadB	PIN_B14
Loa...put	Unassigned
Reset	PIN_B8
Res...put	Unassigned
~QU...D~I	Unassigned
sync:...nc[0]	
sync:...nc[1]	
sync:...nc[2]	
sync:...nc[3]	
sync:...nc[4]	
sync:...nc[5]	

Nodes Found:

Name	Assignments
register_unit:reg_unit|reg_4:reg_B|Data_Out[0]	Unassigned
register_unit:reg_unit|reg_4:reg_B|Data_Out[1]	Unassigned
register_unit:reg_unit|reg_4:reg_B|Data_Out[2]	Unassigned
register_unit:reg_unit|reg_4:reg_B|Data_Out[3]	Unassigned
register_unit:reg_unit|reg_4:reg_B|Data_Out[4]	Unassigned
register_unit:reg_unit|reg_4:reg_B|Data_Out[5]	Unassigned
register_unit:reg_unit|reg_4:reg_B|Data_Out[6]	Unassigned
register_unit:reg_unit|reg_4:reg_B|Data_Out[7]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[0]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[1]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[2]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[3]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[4]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[5]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[6]	Unassigned
register_unit:reg_unit|reg_4:reg_A|Data_Out[7]	Unassigned
Execute	PIN_A7

Insert | Close


Signal Configuration:

Clock: Clk

Data

Sample depth: 128 RAM type: Auto

Segmented: 2 64 sample segments

Nodes Allocated: [✓ Auto] Manual: 17

Pipeline Factor: 0

Storage qualifier:

Type: Continuous

Input port: auto_stp_external_storage_qualifier

Nodes Allocated: Auto Manual: 17

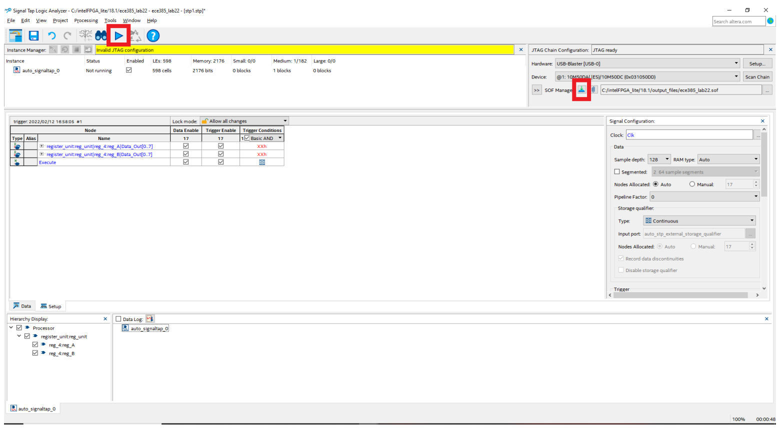
Record data discontinuities

Disable storage qualifier

Trigger

Figure 13: Node Selection for SignalTap of Lab 2.2

4. Compile from the SignalTap window once again and load the sof to the FPGA once again.


Signal Tap Logic Analyzer - C:/intelFPGA_lite/18.1/ece385_lab22 - ece385_lab22 - [stp1.stp]*

Instance Manager: Invalid JTAG configuration

Instance	Status	Enabled	LEs: 598	Memory: 2176	Small: 0/0	Medium: 1/182	Large: 0/0
auto_signaltap_0	Not running	✓	598 cells	2176 bits	0 blocks	1 blocks	0 blocks

JTAG Chain Configuration: JTAG ready

Hardware: USB-Blaster [USB-0]

Device: @1: 10M50DA(.|ES)/10M50DC (0x031050DD)

SOF Manager: C:/intelFPGA_lite/18.1/output_files/ece385_lab22.sof

trigger: 2022/02/12 16:58:05 #1 Lock mode: Allow all changes

Type	Alias	Name	Data Enable 17	Trigger Enable 17	Trigger Conditions 1 Basic AND
		register_unit:reg_unit|reg_4:reg_A|Data_Out[0..7]	✓	✓	XXh
		register_unit:reg_unit|reg_4:reg_B|Data_Out[0..7]	✓	✓	XXh
		Execute	✓	✓	▨

Signal Configuration:

Clock: Clk

Sample depth: 128 RAM type: Auto

Segmented: 2 64 sample segments

Nodes Allocated: [✓ Auto] Manual: 17

Pipeline Factor: 0

Storage qualifier:

Type: Continuous

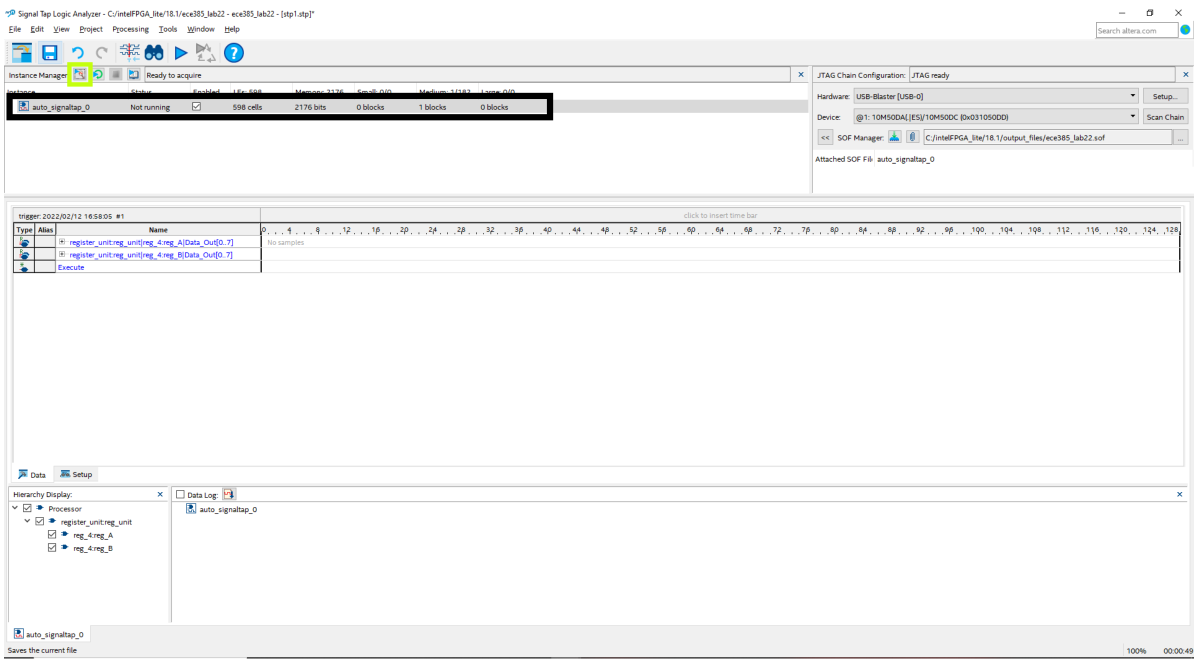
Input port: auto_stp_external_storage_qualifier

Hierarchy Display: Processor, register_unit:reg_unit, reg_4:reg_A, reg_4:reg_B

Data Log: auto_signaltap_0

Figure 14: Rapid Recompile, reupload to FPGA

5. Select the SignalTap Instance you have created and run analysis.
6. On the FPGA give Register A and B some values, we gave 33 and 55, with R = 10, and F = 010, we should expect that register A should hold the value of A XOR B, and register B should hold its own value. Once we have the values in the registers on the FPGA running execute should end the trace and give us the signals under the data tab.



Signal Tap Logic Analyzer - C:/intelFPGA_lite/18.1/ece385_lab22 - ece385_lab22 - [stp1.stp]*

Instance Manager: Ready to acquire

Instance	Status	Enabled	LEs: 598	Memory: 2176	Small: 0/0	Medium: 1/182	Large: 0/0
auto_signaltap_0	Not running	☑	598 cells	2176 bits	0 blocks	1 blocks	0 blocks

JTAG Chain Configuration: JTAG ready

Hardware: USB-Blaster [USB-0]

Device: @1: 10M50DA(.|ES)/10M50DC (0x031050DD)

SOF Manager: C:/intelFPGA_lite/18.1/output_files/ece385_lab22.sof

Attached SOF File: auto_signaltap_0

trigger: 2022/02/12 16:58:05 #1

Type	Alias	Name
		register_unit:reg_unit|reg_4:reg_A|Data_Out[0..7]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[0..7]
		Execute

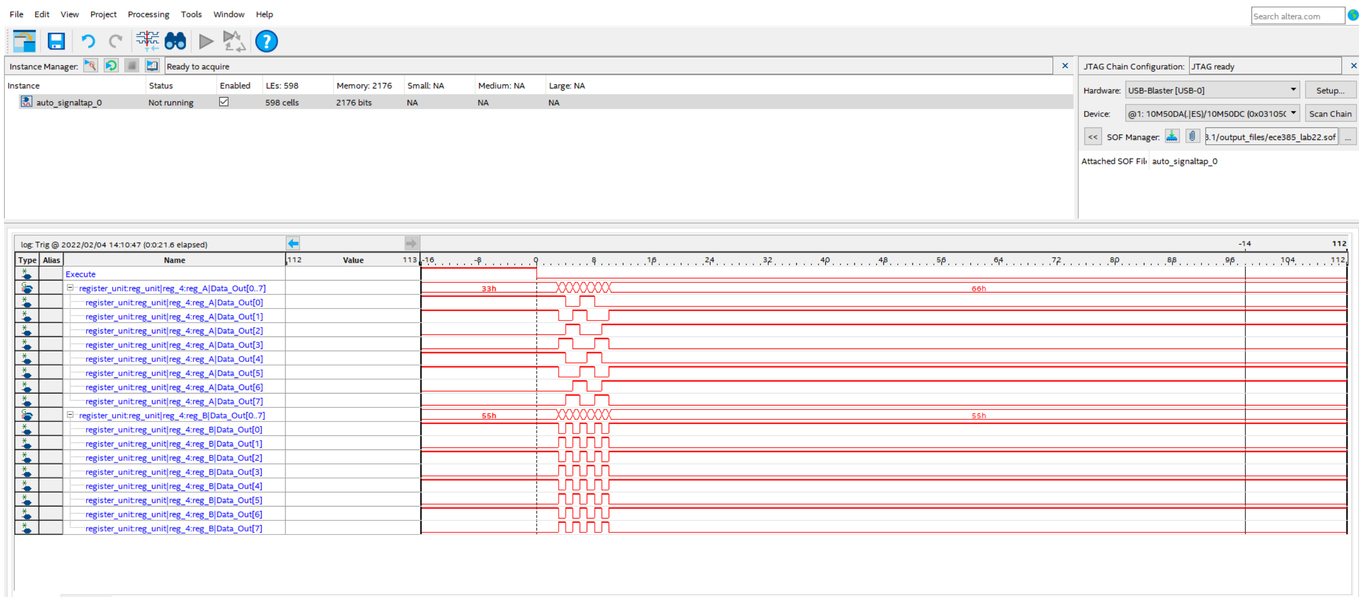
No samples

Hierarchy Display:
- Processor
  - register_unit:reg_unit
    - reg_4:reg_A
    - reg_4:reg_B

Data Log: auto_signaltap_0

Saves the current file

Figure 15: Screenshot of SignalTap instance and run analysis



Instance Manager: Ready to acquire

Instance	Status	Enabled	LEs: 598	Memory: 2176	Small: NA	Medium: NA	Large: NA
auto_signaltap_0	Not running	☑	598 cells	2176 bits	NA	NA	NA

JTAG Chain Configuration: JTAG ready

Hardware: USB-Blaster [USB-0]

Device: @1: 10M50DA(.|ES)/10M50DC (0x03105C

SOF Manager: 3.1/output_files/ece385_lab22.sof

Attached SOF File: auto_signaltap_0

log: Trig @ 2022/02/04 14:10:47 (0:0:21.6 elapsed)

Type	Alias	Name
		Execute
		register_unit:reg_unit|reg_4:reg_A|Data_Out[0..7]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[0]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[1]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[2]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[3]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[4]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[5]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[6]
		register_unit:reg_unit|reg_4:reg_A|Data_Out[7]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[0..7]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[0]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[1]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[2]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[3]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[4]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[5]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[6]
		register_unit:reg_unit|reg_4:reg_B|Data_Out[7]

Figure 16: Screenshot of data collected during analysis of the signal tap during 1 execution cycle

Post-Lab Questions

Q1. 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.

Answer: A circuit that performs the XOR operation is the smallest two input, one output circuit that can optionally invert a signal. XOR can keep a 0 a 0 or invert it to a 1, and, keep a 1 a 1 or invert it to 0. This would've been useful for the lab if we had chosen to use a 4:1 MUX instead of an 8:1 MUX within our computational unit, by doing so we can provide 4 of the 8 logic operations then simply invert them when needed without additional logic, however during our lab we used an 8:1 MUX and an inverter, though it is less efficient we found it to be more straightforward.

Q2. Explain how a modular design such as that presented above improves testability and cuts down development time.

Answer: It allows the developer to build and test each unit/component individually. This in turn results in faster debugging since one can address the problems that arise on a controlled level. It is much easier to debug one small circuit that has one job rather than a fairly large circuit that should be doing 50 things. This also improves testability by allowing the developer to check the output of each unit to ensure that each outcome is appropriate rather than having to verify if the overall output is correct then backtracking through the circuit to debug if something has gone wrong.

Q3. Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

Answer: As mentioned earlier in the report, our circuit design was based on the Mealy State machine instead of the Moore State Machine. In a Mealy machine, the output values are determined by the current states and current inputs. In a Moore machine, the output values are only determined by the current states. A Mealy machine is a better option to use for the circuit design was because the number of states to represent this machine is significantly less as compared to a Moore machine. Due to this, the number of gates and flip-flops to design this will be relatively few. The output in a Mealy machine reacts faster to change in inputs as compared to a Moore machine. However, a Moore machine can be easier to design because of the many states and the transitions.

Q4. What are the differences between ModelSim and SignalTap? Although both systems generate waveforms, what situations might ModelSim be preferred and where might SignalTap be more appropriate?

Answer: The main difference between SignalTap and ModelSim is that SignalTap requires a hardware device such as an FPGA as compared to ModelSim which can be simulated using a testbench file in SystemVerilog. Since ModelSim uses a test bench to simulate all values, the inputs and written down in this file, along with the desired output. In ModelSim, the files such as the processor.sv file in our case is run using the input values provided in the testbench file to compute the outputs. However, since SignalTap requires hardware such as an FPGA, the user must manually turn on the switches to change the input. Due to this, the pin assignment between the FPGA and SignalTap is extremely crucial.

Description of all bugs encountered, and corrective measures taken

Bugs	Corrections
Implementing calculations of S and Q+. For Q+ we did not invert the signal of 1 of our inputs before NAND-ing all three inputs together. Also we incorrectly implemented the 3 input NAND IC by mixing up where the inputs to each NAND were, which resulted in an incorrect S.	When converting our SOP equations from Figure 5 and 6 we did not invert one of the signals on the board before passing it to the three input NAND before passing the signal on to the flip flop. We realized our 3 input NAND was incorrect after hooking LEDs up to see what our output was and we saw that we were not getting a signal from
Wiring the Up/Down Counter- wired the ENP and ENT wrong. Due to this, the counter would start the counter even if we did not press execute.	Signal S was passed in as ENP and the ENT is set to high.
Inefficient Logic for Compute Unit- represented all the logic operations separately using NAND and NOR gates. This caused us to use a lot more chips than needed.	Only made NAND, NOR, and XOR using separate chips and computed AND, OR, and XNOR by inverting the signals from NAND, NOR, and XOR. This resulted in fewer chips on our board.

Logic for S0 S1 on shift registers. We found that for the bit shift registers the inputs S1, S0 drive the register to load, shift or halt. When S1,S0 are (0,0) the register halts, when it is (1,1) the register loads, when it is (0,1) the register shifts.	To drive the registers our solution was to drive S1 with Load AB, since it was only 1 when the register loads. And S0 needs to be 1 when the register loads and is shifting otherwise it is 0. Our solution was to compute LoadAB OR S and provide that signal to S0
Increasing states in the state machine- In lab 2.2, while converting the logic processor into an 8-bit processor, we changed all the bits to account for the 8 bits but we didn't realize that we needed to add 4 additional states to account for 4 additional shifts of the bits.	Added 4 additional states in the enum logic of file control.sv extending from [3:0] → [4:0] to allot for a total of 10 states rather than 6.

Table 5: Circuit design bugs are corrections for the same

Conclusion

This lab gave us a good mix for circuit design and SystemVerilog. Lab 2.1 made us utilize several Multiplexers, gates, and other integrated circuits that we learned about in ECE 120 for the purpose of computing bitwise operations. It allowed us to think of an efficient and optimal circuit design, one example of this was the decision to use a Mealy finite state machine over a Moore state machine since it would be easier to implement. Lab 2.2 allowed us to get familiar with Quartus Prime and understand the fundamentals of SystemVerilog such as modules, blocking vs non-blocking assignments, and pin assignments on the FPGA. It helped us explore the important parts of Quartus Prime such as RTL Simulation and SignalTap. Overall, the purpose of the lab, to compute 8 different bitwise logic operations, was successfully completed.

Appendix A:

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 module calls all of our other modules

Purpose: This module, depending on the comments, can prime the software for either SignalTap or ModelSim, and instantiates the other modules. Without it our FPGA or testbench would not do anything.

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 module is comprised of two 4 bit shift registers, constructed from reg_4.sv, it defines the values to shift into the register, when in the load state, and shift out in the shift state as well as the least significant bit that gets passed on to the rest of the code.

Purpose: This sets up our A and B registers that will right shift, and load correctly

Module: router.sv

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

Outputs: A_Out, B_Out

Description: Dual 4:1 MUX, that gives A_Out and B_Out the values of A, B or F

Purpose: Pass the correct new value to the shift registers,

Module: reg_4.sv

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

Outputs: Shift_Out, [7:0] Data_Out

Description: If the state is in reset at the rising edge of the clock the registers are set to 0, if the state is load, then the registers are set to the value given by the input D, and finally if the state is Shift_En then the registers are set to the value of Shift_In and Data_Out concatenated together.

Purpose: Construction of a shift register that is called a register unit, for shifting loading and resetting our registers.

Module: hexdriver.sv

Inputs: [3:0] In0,

Outputs: [6:0] Out0

Description: Takes a nibble or 4 bits and converts it to a 7 bit equivalent that is displayable on the HEX# output LEDs of the FPGA.

Purpose: To allow the LED's to display what the input and output of our code is.

Module: compute.sv

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

Outputs: A_Out, B_Out, F_A_B

Description: 8:1 MUX that performs the combinational logic for the chosen operation of the logic processor (AND, OR, XOR, 1111, NAND, NOR, XNOR, 0000).

Purpose: To perform 1 of the 8 logic operations that the processor should and pass the value of A and B on to the next step.

Module: control.sv

Inputs: Clk, Reset, LoadA, LoadB, Execute,

Outputs: Shift_En, Ld_A, Ld_B

Description: Finite State Machine with 10 states, 1 Load, 8 shifts, and 1 Reset State. When loading shift enable is 0 and LoadA and LoadB are 1, when shifting and resetting LoadA and LoadB are 0 and shift enable is 1 and 0 respectively

Purpose: To keep the shift registers from shifting too many times per operation / actuation of execution.