ECE 385

Spring 2023
Experiment #2

Logic Processor

Michael Gamota
Aditya Adusumalli

Introduction

In this experiment, we built a 4-bit logic processor. Our processor has a synchronous design and the user can select from 8 logical operations (Figure 4) to perform on the contents of two 4-bit registers. The processor is capable of placing the result of the logic function in one of the two registers. In addition to those just mentioned, the user controls the initial values and loading of those values into each register as well as when the computation cycle starts. The processor will run the selected function once every time the user prompts it to do so.

Operation of the Logic Processor

To load data into either register the user must select the 4 but value using switches 1, 2, 3, 4 on the DIP switches at the top of the board where switch 4 dictates the least significant bit. “Open” signifies a 1 and “closed” signifies a 0. Once the data values have been set, the user then flips either switch 6 or 7 to load the set 4-bit value in register A or B, respectively. Then those steps are repeated for the other register starting with selecting a new 4-bit value if the user desires.

In order to compute a computation cycle, the user should first select the logical operation and the routing. The inputs/op codes for these are detailed in Figure 4 & Figure 5 and correspond to different logic operations and routing tracks. The user uses switches 1, 2, and 3(F2, F1, F0 respectively) to select the logic operations. The user uses switches 7 and 8 (R1 and R0) to route the signal from the computation unit to the desired location which may be register A, B or neither. Once the user has selected the logic operation and routing, the computation cycle begins when the user flips the “Enable” switch which is Switch 1 of the rocker switch array at the bottom of the board. The computation cycle will run only once (compute all 4-bits) regardless of the position of the “Enable” switch when the cycle continues. To run a new cycle, the “Enable” switch must move to the off position and then after the current cycle ends, back to the on position.

Written Description

The register unit includes two 4-bit bidirectional shift registers (74194 IC). One register was designated “Register A” and the other “Register B”. After looking at the datasheet, we decided to use the right-shift function, but the left-shift function would have also worked for our purposes. There are 3 functions for the register that we wanted to use: shift, parallel load, and halt each register. To do so, we devised logic which we detail in the “design steps” section. The inputs to our registers are the clock used by the whole processor, the “Load A” and “Load B” switches, a serial input from the routing unit, and a “Shift” signal generated by the control unit. The outputs from the register are the contents of the register which is visualized using an LED array block.

The computation unit was built using one NAND IC (7400), one XOR IC (7486), one NOR IC (7402), and one 8:1 multiplexer IC (74151N). The inputs for the logic ICs were the original A and B values or in some cases, a function of A and B that needed to be inverted (ie (A AND B) = NOT(A NAND B)). The 6 outputs of the logic units can be called F(A, B) and each F(A,B) was connected to a unique input on the 8 input multiplexer. The other 2 inputs on the multiplexor were either pulled high or low to implement a set or clear function, respectively. The select pins of the multiplexor received input from the DIP switch array box near the bottom of the board, specifically switches 1, 2, and 3 were used for F2, F1, F0, respectively. The output of the multiplexor was connected to 2 inputs on the dual multiplexer used in the routing unit.

The routing unit was built using a dual 4:1 multiplexer (74153N). The select pins on the IC operated both of the 4:1 MUXs. The output of the left side of the IC (MUX 1) was connected to the serial input of Register A, and the output of the right of the IC (MUX 2) was connected to the serial input of Register B. Both MUXs had the same inputs overall (A, B, F(A,B)) but connected to different respective input pins (detailed in Figure 11).

Lastly, the control unit used a 4-bit counter (74161), a D flip-flop (7474), a 3-input NOR IC (7427), a hex inverter (7404), an XOR IC (7486), and a NAND IC (7400). The inputs to the control unit were “Load A”, “Load B”, “Clock”, and “Execute”. The outputs of the control unit were connected to the S1 and S0 pins of the registers, which are responsible for the registers’ operations (Shift, load, halt). The logic ICs, flip flop, and counter were used to compute the output signals and the equations, layouts, and K-maps are detailed later in the report and would be hard to keep track of if described here in text.

Block Diagram

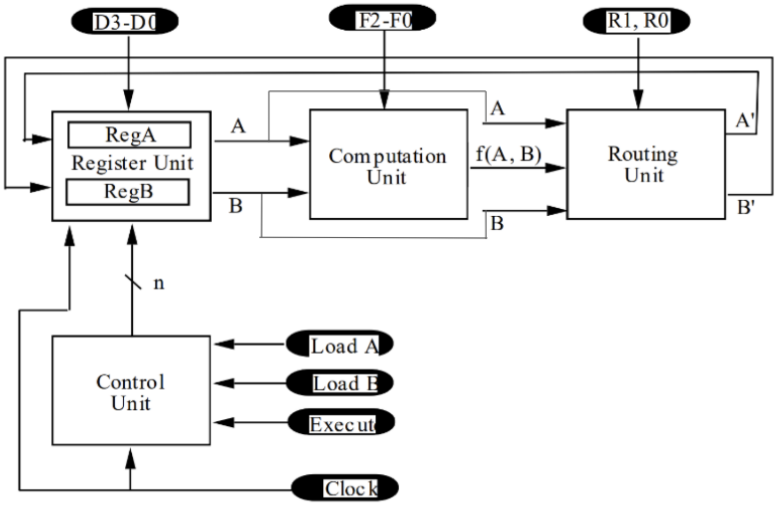


Figure 1: Block Diagram

State Machine Diagram

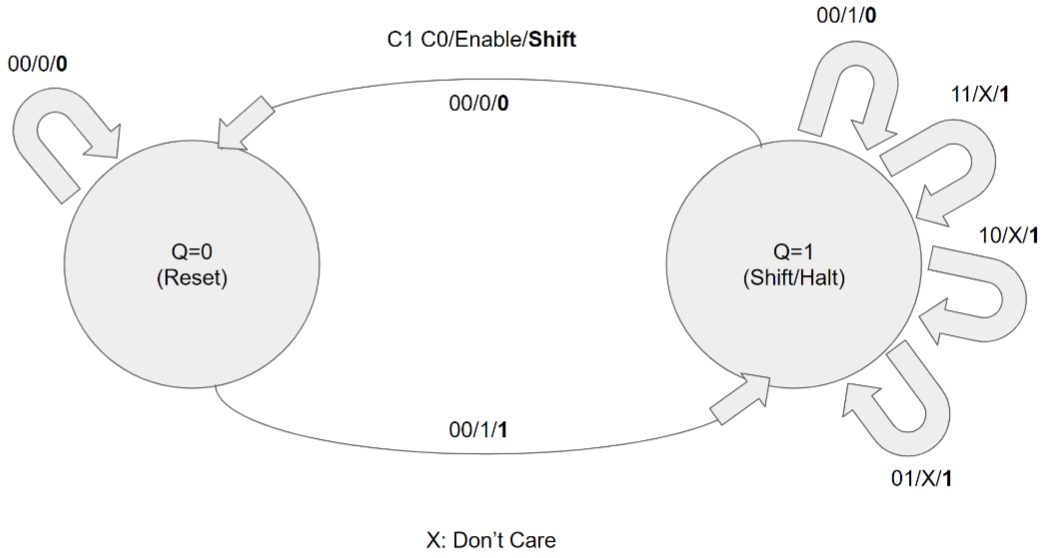


Figure 2: Mealy Machine State Diagram

Post Lab Q3: We used a Mealy machine because that is the method that seemed to be encouraged in the lab instructions. The benefit of using the Mealy machine is that there are fewer states, in our case, only 2.

Design Steps

Starting with the register unit, we decided to use two 4-bit bidirectional shift registers (74194). This would allow us to do both parallel loading as well as simultaneously shifting one bit out and a new bit in. We also needed to display the contents of the registers so we used the DIP LED arrays to display the contents of both registers. To set the bits for the parallel loading of the registers, we used 4 DIP switches because switch bounce would not be an issue. The controlling of the register unit functions is detailed in the control unit paragraph.

For the computation unit, we used an assortment of chips (detailed in "Written Description") to enable 8 bitwise logic operations: Set, Reset, XOR, XNOR, OR, NOR, AND, and NAND. We tried to minimize the number of chips, so we used the extra OR gates to invert the outputs of OR and XOR. This way we could save space on the breadboard for the more complicated control unit. We connected the outputs of each of these operations to a 8:1 multiplexer which had its 3 bit select bus connected to the F2, F1, and F0 DIP switches which we used because switch bounce would not be an issue. This way we would be able to select the logic operation performed on the inputs A and B. The table relating F2, F1, and F0 to the corresponding output is shown below.

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

Figure 4: Computation unit function table

For the routing unit, we needed to implement the following table:

Routing Selection		Router Output	
R1	R0	A*	B*
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

Figure 5: Routing unit function table

In order to do this, we used a dual 4:1 multiplexor with a shared select input. This meant that one MUX output would be A* and the other would be B*. The inputs R1 and R0 would be the select inputs coming from the DIP switch box which we used because we didn't have to worry about switch bounce.

The shift registers have 2 inputs which control the function of the registers, S1 and S0. In order to devise logic that would turn the "Shift" signal from the control module into 2 different values to apply to S1 and S0.

Function	S1	S0
Hold	0	0
Shift Right	0	1
Parallel Load	1	1

Figure 6: Functioning of Bidirectional Shift Register

Based on this table, we noticed that the only time S1 should be high is when we are doing parallel loading, so we figured that we could hardwire the "Load A" and "Load B" inputs directly to S1 on their respective registers. Additionally, we decided that the "Shift" signal from

the control unit should be part of the logic expression for S0. Since we also needed S0 to be high for parallel loading, we made **$S0_A$ = Load A XOR Shift** and **$S0_B$ = Load B XOR Shift**. This can be considered as part of the control unit, so the rest of the control unit logic will be covered now. The control unit takes in 3 inputs, Load A, Load B, and Execute. Our implementation uses a Mealy Machine implementation shown with the below truth table. The value S is used to stop the processor after it has processed the original 4 bits in the register. The processor then waits for "Execute" to go low and then high again to process the new 4 bits.

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

Figure 7: Truth table for control unit

This truth table was turned into these K-maps and equations:

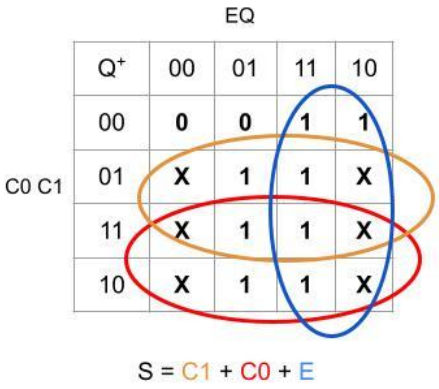


Figure 8: K-map and equation for Q (Sequential)

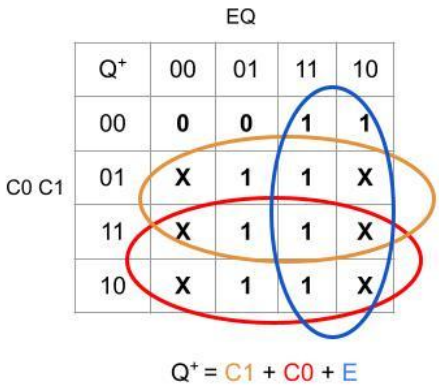


Figure 9: K-map and equation for S (Combinational)

In our implementation, we generate C1 and C0 using a 4-bit counter (74161). In order to stop the counter after completion of one processing cycle, or 4 clock cycles (one for each bit), we connected S to the enable input of the counter. To implement the sequential logic, we used a D flip-flop (7474). To implement the combinational logic we used a 3-input NOR IC (7427), a hex inverter (7404), an XOR IC (7486), and a NAND IC (7400). While this was not the most efficient design, we ran into problems when debugging and found that when we used the hex inverter instead of simplifying the data path and using the inverted output of the flip flop our design worked as intended. One important design consideration was the debouncing of the "Execute" switch. To do this, we used the rocker switch provided in the lab kit, and used the debouncing diagram in the General Guide. This ensured that when we flipped the switch, we would not get any glitches which could cause the processor to not run as intended.

Circuit Schematic Diagram

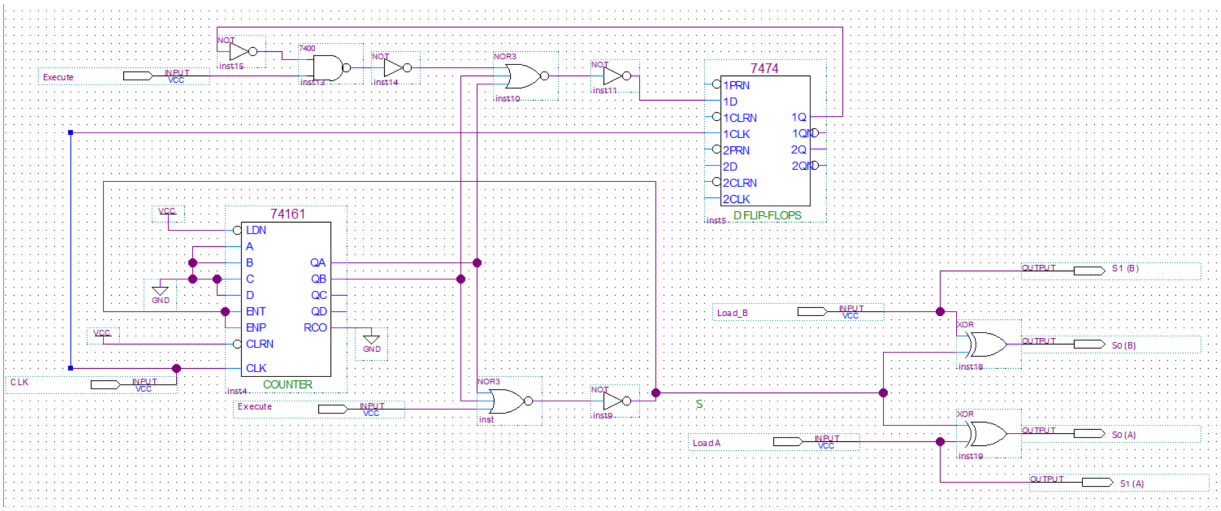


Figure 10: Control Unit Schematic

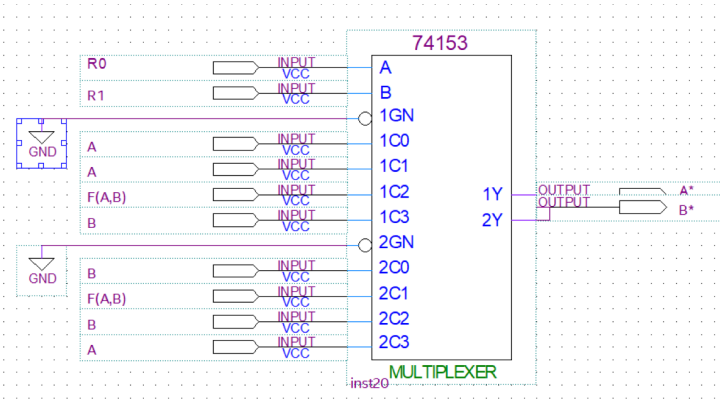


Figure 11: Router Unit Schematic

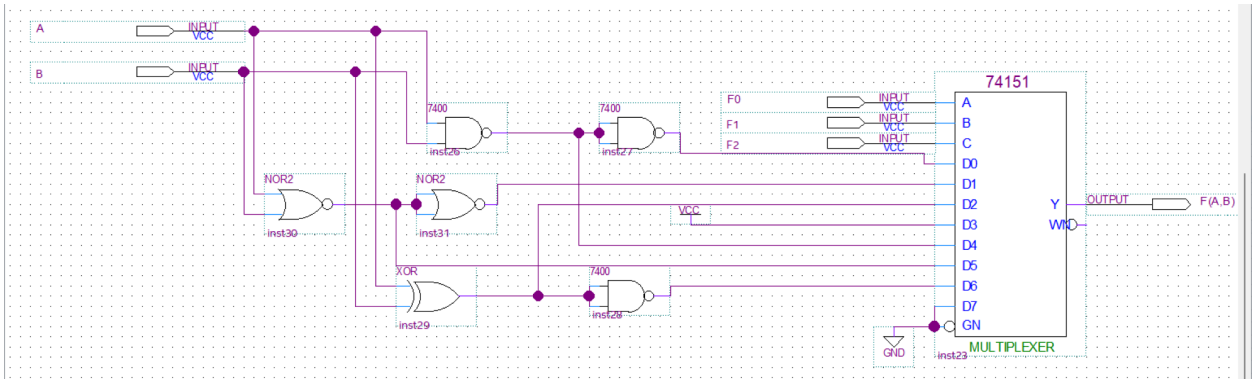


Figure 12: Computation Unit Schematic

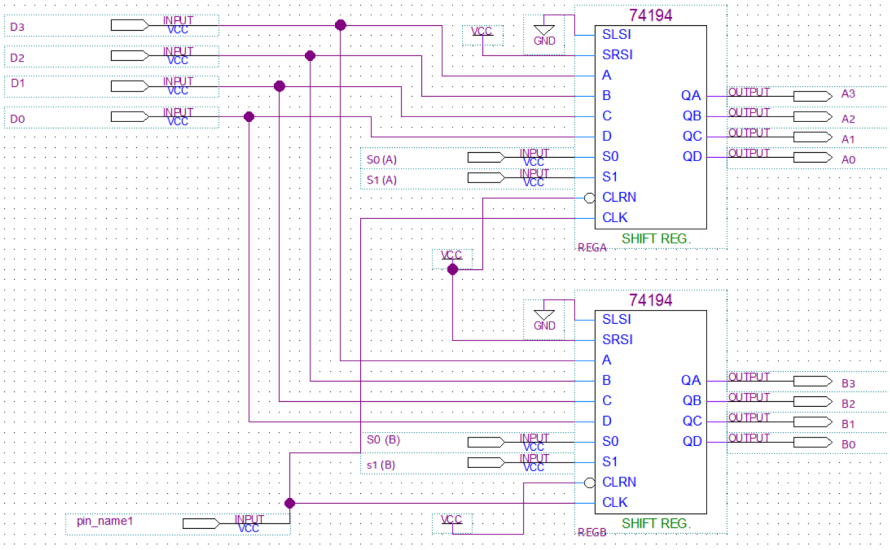


Figure 13: Register Unit Schematic

Breadboard view / Layout sheet

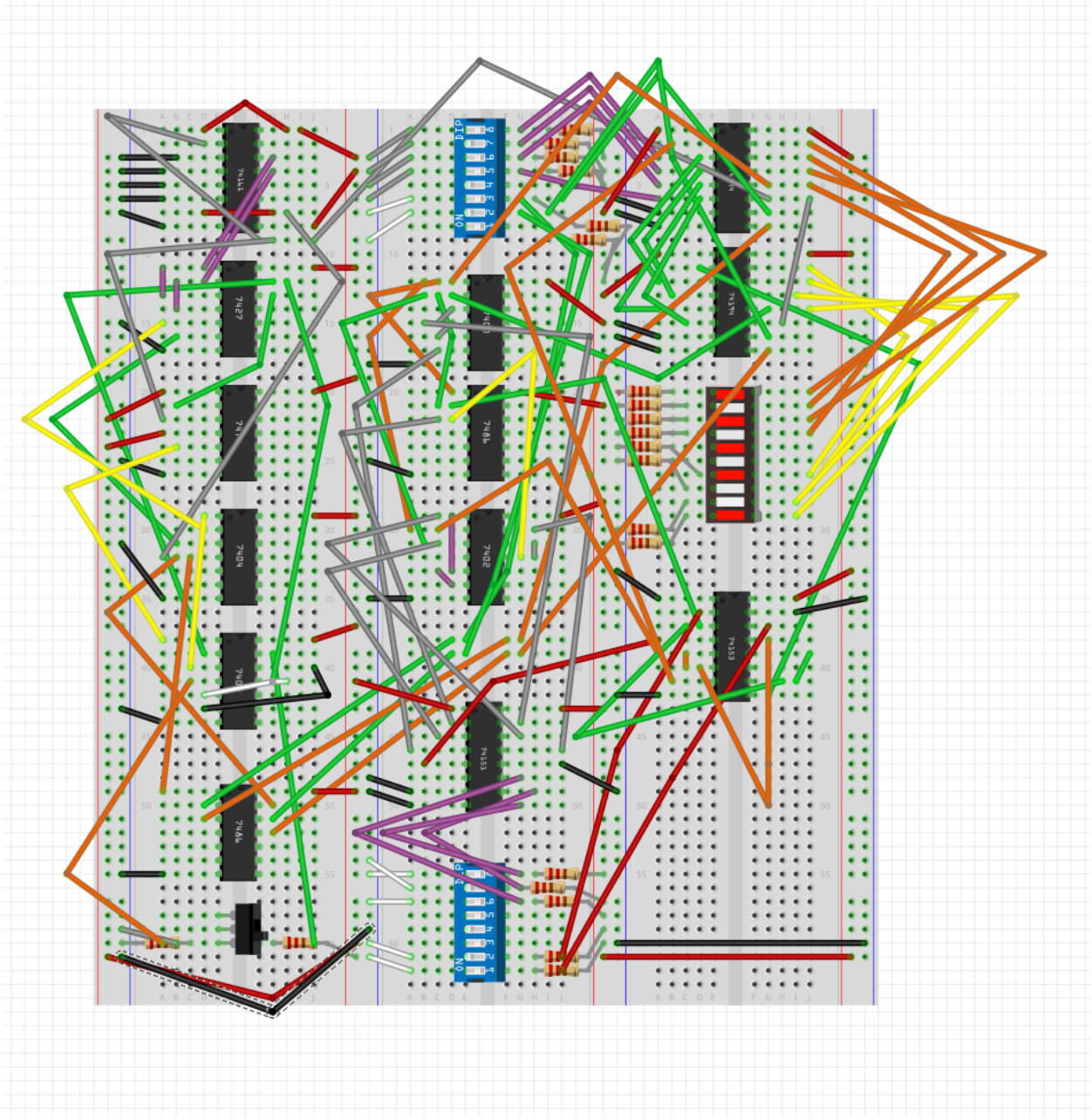


Figure 14: Fritzing Breadboard Layout

8-bit logic processor on FPGA

The 8-bit logic processor on the FPGA is an extension of Lab 2.1 as we implemented a logic processor with the same available logic operations and routing selections. The difference lies in the number of bits stored in the register and therefore the number of bits that are operated on in one operation cycle. To implement this we used 8-bit registers as well as increased the number of states in our FSM model. The FPGA implementation used a Moore state machine because it is relatively straightforward to do so in SystemVerilog. We added 4 extra states to account for the 4 extra bits.

1) Compute.sv
    a) Inputs: [2:0] F, A_In, B_In
    b) Outputs: A_Out, B_Out, F_A_B
    c) Description: This file uses the input logic to determine which logical operation to apply to A and B, including AND, OR, XOR, and Set(1111) as well as the opposite of those 4.
    d) Purpose: This file is responsible for the computation unit which uses one of the above combinational logic operations to compute f(A,B).
    e) Changes: None
2) Reg_4.sv
    a) Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D
    b) Outputs: Shift_Out, [7:0] Data_Out
    c) Description: This module uses a sequential “always_ff” block to store 8 bits of data and shift them out one at a time which shifting in simultaneously
    d) Purpose: This module provides an 8 bit register structure used to store both A and B
    e) Changes: Extend register size from 4 bit to 8 bit by increasing size of D and Data_Out
3) Testbench_8.sv
    a) The testbench does not include modular inputs and outputs but instead has internal logic variables which correspond to the inputs and outputs of the Processor.sv file
    b) Description: The testbench file sets up logic variables and then sets values to input into the Processor.sv file to run the entire logic processor system
    c) Purpose: The testbench file allows us to simulate the logic processor in ModelSim.
    d) Changes: None
    e)

4) Synchronizers.sv
    a) Inputs: Clk, d, Reset
    b) Outputs: q
    c) Description: This file declares the synchronizers required to allow asynchronous inputs, in this case the switches and buttons, to be read into the FPGA
    d) Purpose: Since the processor relies on user input from the switches which do not line up with the clock pulses, we need to synchronize these asynchronous inputs
    e) Changes: None
5) Router.sv
    a) Inputs: [1:0] R, A_In, B_In, F_A_B
    b) Outputs: A_Out, B_Out
    c) Description: The module uses an “always_comb” block to route the incoming signals to the specified register
    d) Purpose: This allows the user to perform one of 4 operations, keep A and B the same, swap A and B, keep A and put f(A,B) in B, or keep B and put f(A,B) in A.
    e) Changes: None
6) Register_unit.sv
    a) Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D,
    b) Outputs: A_out, B_out, [7:0] A,[7:0] B)
    c) Description: This module is made up of two 8 bit register units. Load_A and Load_B are parallel load enable pins and A_In, B_In, and A_Out, B_Out are used during shifting for serial shift in and out
    d) Purpose: This allows the user to store A and B in one unit which makes the function of the whole processor easier and more abstracted
    e) Changes: Extended D, A, and B to 8 bits
7) Processor.sv
    a) Inputs: Clk, Reset, LoadA, LoadB, Execute, [7:0] Din,
    b) Outputs: LED, output logic [7:0] Aval, Bval, [6:0] AhexL, AhexU, BhexL, BhexU
    c) Description: The processor module is the top-level entity meaning that it has all the inputs and outputs needed to run the entire 8-bit logic processor. It instantiates all the other modules which are necessary for the processor to function as a whole: HexDriver, button_sync, Din_sync, F_sync, R_sync, control, compute, router, and register unit
    d) Purpose: This module creates the actual logic processor by tying all the individual units together

e) Changes: Increase the size of local variables A, B, and Din_S as well as input variables Din, I hardcoded the values for F and R, and increased the size of output LED, Aval, Bval to 8 bits

8) HexDriver.sv
   a) Inputs: [7:0] In0,
   b) Outputs: [6:0] Out0
   c) Description: This file uses a 4 bit input to create the corresponding output in hex on a 7 segment LED display
   d) Purpose: Allows data to be visualized in hex, but unused in this lab
   e) Changes: Extended In0 to 8 bits
9) Control.sv
   a) Inputs: Clk, Reset, LoadA, LoadB, Execute,
   b) Outputs: Shift_En, Ld_A, Ld_B
   c) Description: This file determines the output based on the current state and transitions from the current state to the next state within a always_comb block.
   d) Purpose: The control file creates a state machine for the control unit to ensure the processor only processes 8 bits at a time regardless of if “Execute” is still depressed.
   e) Changes: We had to add 4 more states: G, H, I, and J to allow the processing of the new 4 bits

RTL Diagram

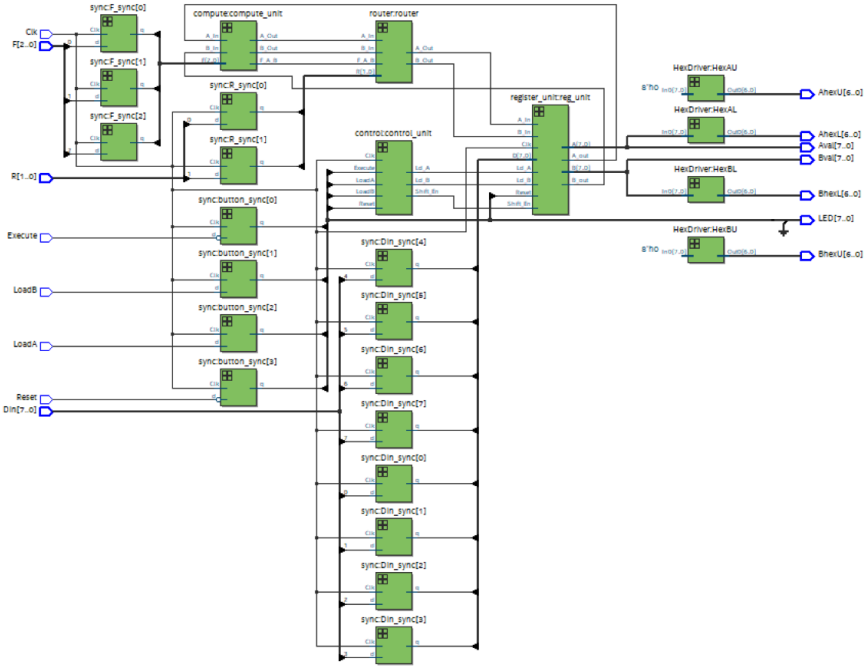


Figure 15: RTL Diagram

ModelSim Trace Annotations

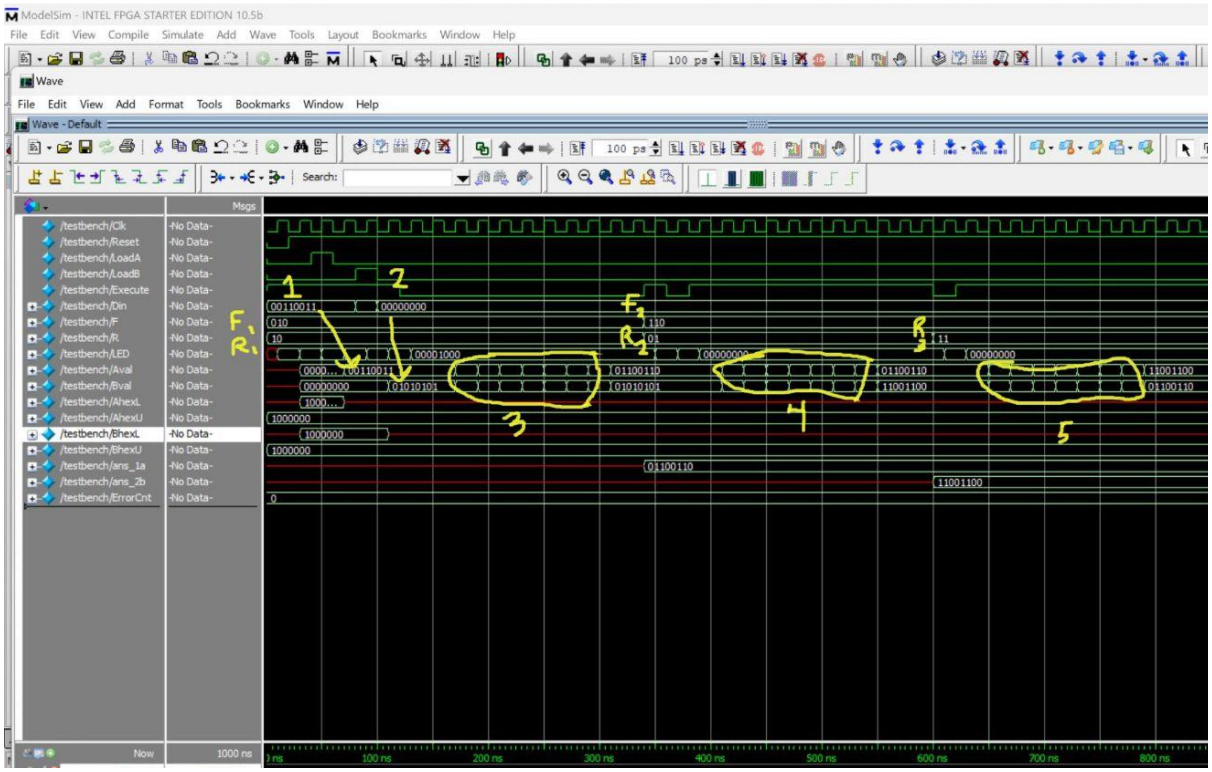


Figure 16: ModelSim Annotations

F1: First logic operation, XOR (010)
R1: First router instruction, Put f(A,B) in A, keep B the same (10)
1: Load A goes high and Din is placed in Aval on falling edge of Load A
2: Load B goes high and Din is placed in Bval on falling edge of Load B
3: Logic operations are performed one bit at a time and the registers shift to allow this
F2: New logic operation, A XNOR B
R2: F(A,B) is to be placed in B and A is to be kept the same (01)
4: Logic operations are performed one bit at a time and the registers shift to allow this
R2: A and B are swapped (11)

8.d) In order to use SignalTap, we first had to assign pins to all of the top level (Processor.sv) inputs and outputs according to the lab instructions. After doing this, we can select the “clk” signal to be the clock, then we add nodes to

analyze. We chose to view “Execute” and “Data_out” for registers A and B and set the data type to binary. We set the trigger to be the falling edge of the “Execute” signal because the button is pulled high. That way the data starting when the button is pushed is captured. After recompiling and programming the chip, we can view the real measured values of the specified signals in SignalTap.

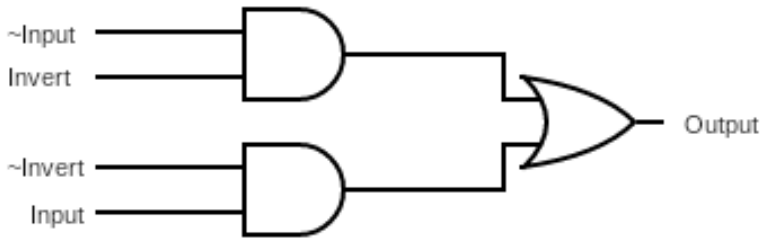
Bugs

The only bug that we ran into occurred when we built and tested the circuit initially with a 1 Hz clock and used a DIP switch for the “Execute” signal. While the switch bounce was not enough to cause problems with the 1 Hz clock, when we changed the clock to 1 kHz, we noticed that the computation cycle would run twice due to the switch bouncing. To fix this bug, we followed the debouncing switch diagram in GG.34. This debounced switch worked as intended for our execute signal and we no longer had issues with multiple computation cycles.

Conclusion

In this lab, we utilized DIP ICs to implement a 4-bit logic processor. Our logic processor is synchronous, has 8 possible logic operations, and stores the result in one of two registers. The processor takes in user input and runs exactly one computation cycle every time the user flips the “Enable” switch to the ON position. Our implementation was based on a Mealy state machine. The user had control over 12 inputs: Enable, Load A, Load B, D3:D0 (register parallel load), F2:F0 (computation unit control), and R1:R0 (routing unit control). The processor was tested at clock speeds of 1 Hz and 1 kHz where it passed all tests.

Post Lab Q1, Q2:



This is useful in the design of the lab because there is at least one place where we want to be able to select an inverted signal: the logic unit. NAND, NOR, and XOR are all the inversion of AND, OR, and XNOR, respectively, so being able to selectively output these in the logic unit would be one possible implementation.

Modular designs like the one above are a great way to add abstraction to a system and allow for easier debugging.

Post Lab Q4

ModelSim is purely a simulation, it uses the RTL design of the FPGA to simulate functions outlined in a testbench file. SignalTap is used to measure actual signals on the board, meaning the FPGA must be plugged into Quartus. If the programmer wants to see if they have set up their logic correctly, using ModelSim doesn't require them to program the FPGA and do a full compile. However, if after programming the FPGA, there is still an issue, the user may want to use SignalTap to analyze the signals, particularly the manual inputs like switches and buttons which can only be measured/debugged using SignalTap.