ECE 385

Spring 2024

Lab 2

Ziheng Li (zihengl5)

Xin Yang (xiny9)

Section AL1

Prof. Zuofu Chen

Introduction

For this lab, we design and build a bit-serial logic processor, which does operations on two 4-bit registers. In total, it can do 8 different operations: AND, OR, XOR, 1111, NAND, NOR, XNOR, 0000. For output, we have 4 different modes for displaying the computation results: 1. Hold AB 2. Hold A, show result in B, 3.Hold B, show result in A, 4. Exchange value between AB.

Operation of the Logic Processor

First, we have to flip the switches D3-D0 to choose the desired data to load. Second, we need to flip the Load A/Load B switch to choose which registers we want to load the data into.

In order to initiate a routing and computation operation, we have to flip R1R0 on routing units to choose the displaying mode. Then we have to flip F2-F0 to select the function that we want to operate between A and B. Once all the steps above have been completed, the user finally flips Execute to initiate a computation, and the result will be showed on the led.

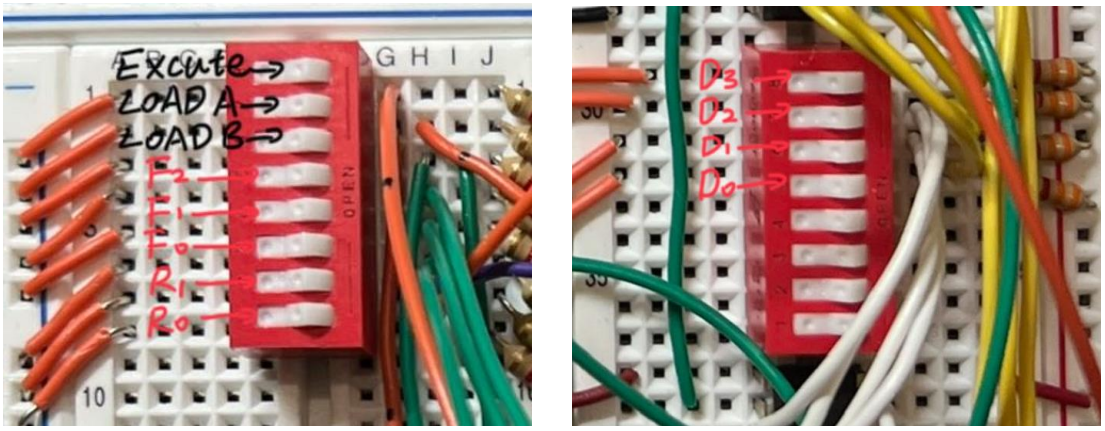


Fig.1 Switches needed to operate the processor

Description for Circuit & Block Diagram

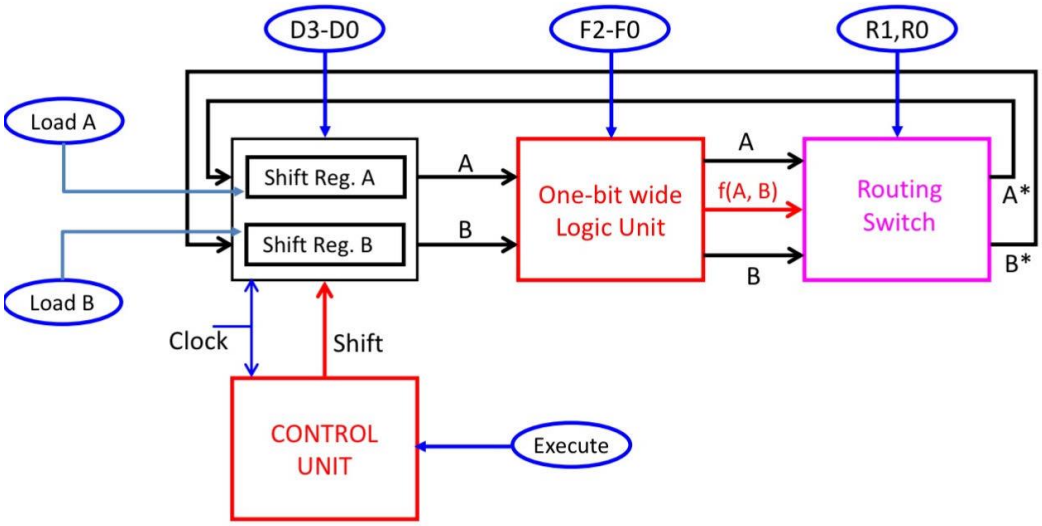


Fig.2 High-level block diagram

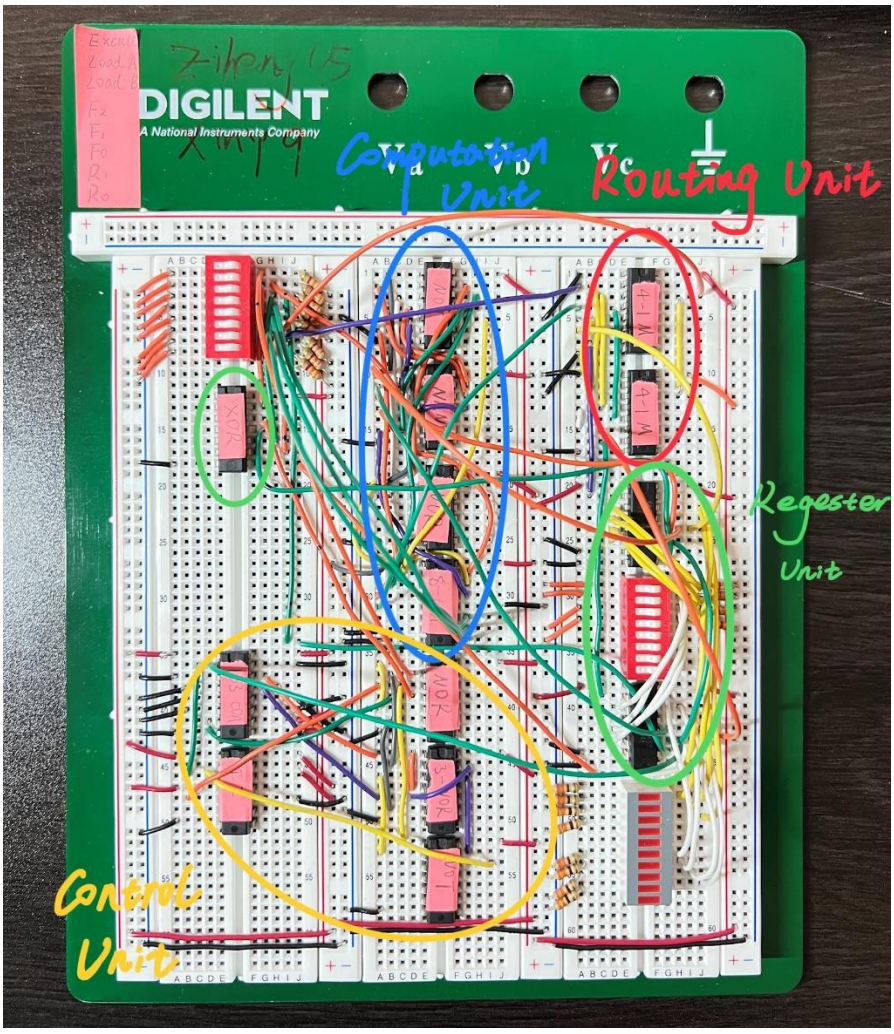


Fig.3 Actual layout with block diagram

Computation Unit (See Design Procedure for details)

There are 2 inputs, 3 outputs, and 3 control signals for the computation unit. By selecting F2, F1, and F0, we choose 1 operation from 8 to execute between A and B, finally produce the output f(A, B) and feed it into routing units along with A and B (both are remains the same).

Routing Unit (See Design Procedure for details)

The routing unit has 3 inputs, 2 outputs, and 2 control signals. R1R0 controls the 4 modes of output. AB is the same as AB feed into the computation unit. And f(A,B) is the result of the computation unit. We implement this part with two 4-1 multiplexers.

Control Unit (See Design Procedure for details)

Control Unit has only 1 input and 1 output. Inside the control unit, a FSM perform the state operation and produce the output SHIFT. Since the logic processor only has 4 bits, we need to control it so that it stop to produce shift signal after 4 cycles. To achieve it, we employ a counter that counts from 0 to 3.

Register Unit (See Design Procedure for details)

The shift register unit has 3 inputs, 2 outputs and 6 control signals. The output from the routing unit is the bit we want to store in the leftmost bit of the register. Control signal D3-D0 manipulate the value that we want to load into the register as a starting point. Combined logic between Load A/B and S controls whether we load the value from D3-D0 or shift the value on register. For this part, we use 2 four bits shift register and several logic gates.

State Machine

In out design, a Mealy state machine was used as the center of the control unit. To associate the binary flip-flop with the diagram, we have store the state Q in the flip flop: 0 is the Reset State, while 1 is the Shift/Hold State.

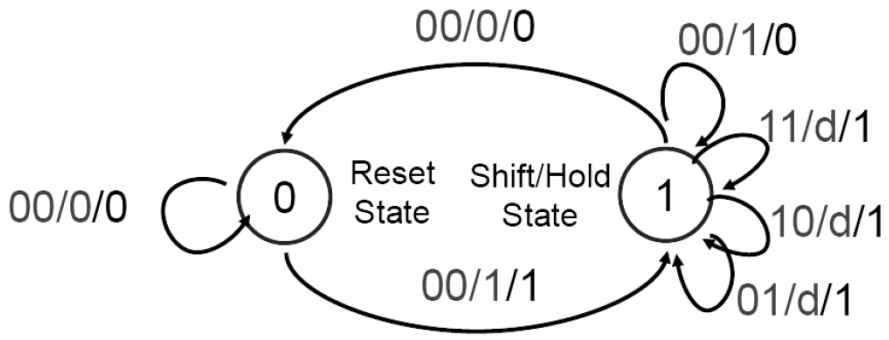


Fig.4 State Machine (Mealy) diagram

The Arc label is as follows:

- 00/0/0–Set/Reset
- 00/1/1–shift 0
- 01/d/1–shift 1
- 10/d/1–shift 2
- 11/d/1–shift 3
- 00/1/0–HOLD

C1C0 / E / S
Count / Execute / Shift-count Enable
(Input) (Input) (Output)

Fig.5 Meaning for arcs

Design Procedure & Circuit Schematic

We choose to start on the combinational logic part, the Logic Unit and Routing Switch, since both of them only utilize basic NAND/NOR/NOT gate, it will be easier to implement and test.

Logic Unit

We first designed the Logic Unit, as mentioned in the lecture, we did not build all 8 logics, instead, we built half of them and then carry out the other halves by simply adding an inverter. The four logics that we first build are NAND, NOR, XNOR, and 0000. We have the corresponding chips for NAND and NOR operation. 0000 is nothing but grounding the signal. As for XNOR, even though XOR is available, we thought it would be redundant to have an extra chip on the board while we can implement XNOR using NAND and NOT (as shown in Fig. 6). After half of the logic is implemented, we then negate all of them through an inverter and thus create the other half. Finally, for logic selection part, we employ a 8-1 MUX with 8 logics as inputs, and F2, F1, F0 as selector to selects the final logic operation we want to perform.

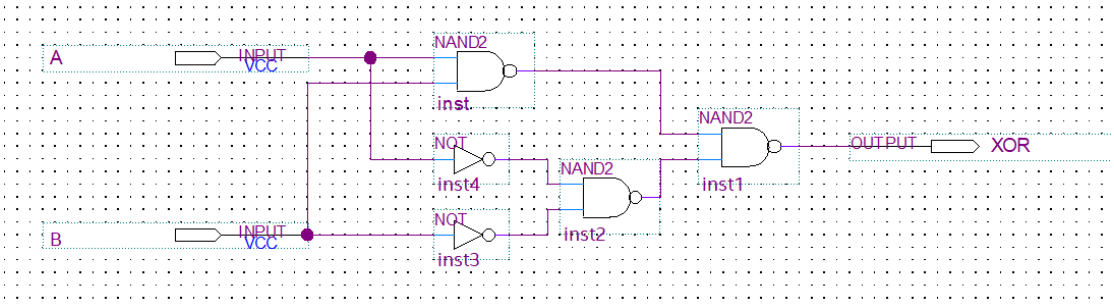


Fig.6 XOR using NAND and NOT

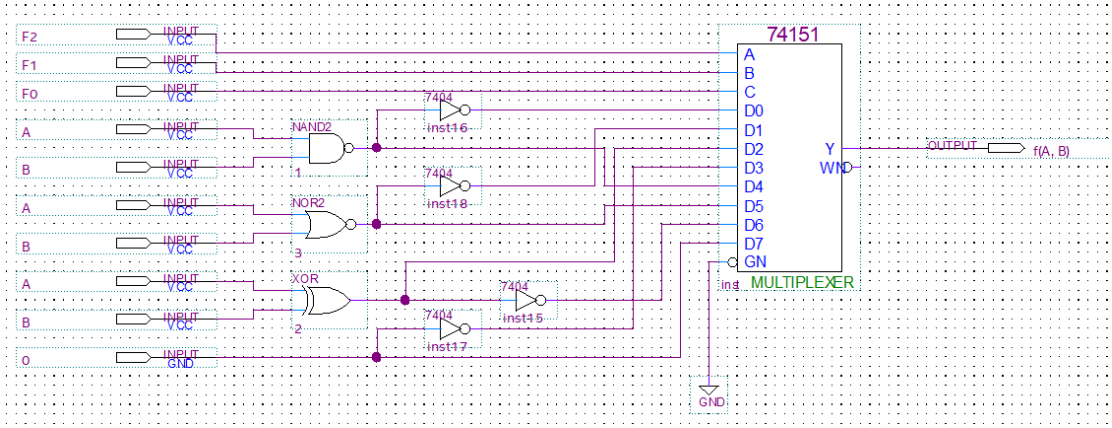


Fig.7 Overall layout for Computation Unit

Routing Unit

Second, the Routing unit is simpler compared to Logic units. All it requires are two 4-1 MUXes, and they share the same selector R1, R0.

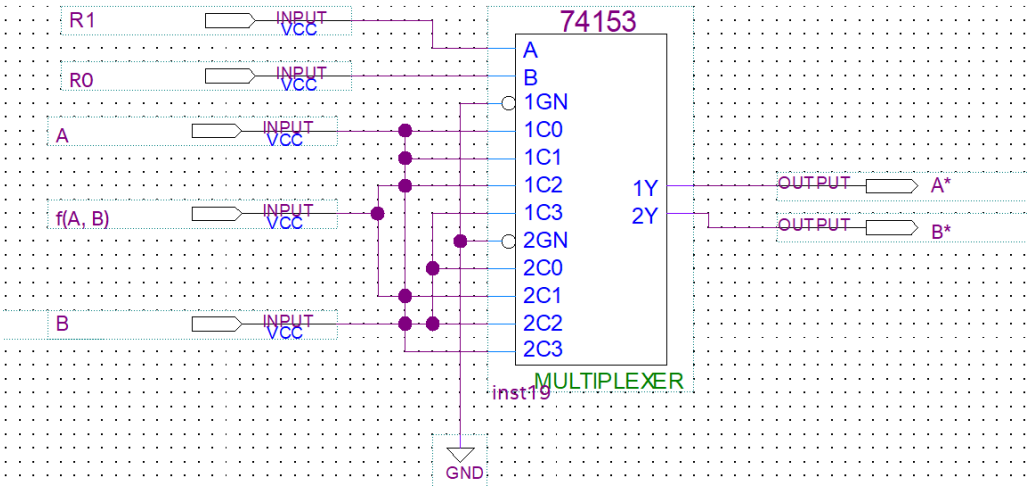


Fig.8 Overall Layout for Routing Unit

Control Unit

Third, we build the control unit. It only takes EXECUTE as its input and SHIFT signal as its output. As suggested in the lecture, if we implement the FSM using Moore State Machine, it would have more states and hence more binary digits to represent the state. Hence, we choose to use Mealy State Machine with only two states, shift and hold. Note that there is only four bits in the shift register; we need to make sure the FSM stays in HALT after four counts. As for the next state logic, we take advantage of a counter. The sequence of the state is always 00, 01, 10, and 11. Which matches with the behavior of a counter. Hence, the Reset vs. Shift/Hold state became the only signalfeed back to the FSM again. Moreover, a flip-flop is required to store its previous value as the next state's input. K-make for Q's next state logic and shift signal are following.

K-Map for Q+			C1 C0			
			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	
		POS = E+C1+C0				

K-Map for S			C1 C0			
			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	
		POS = (E+Q)(Q'+C1+C0)				

Fig.9 K-map for Q+ and S

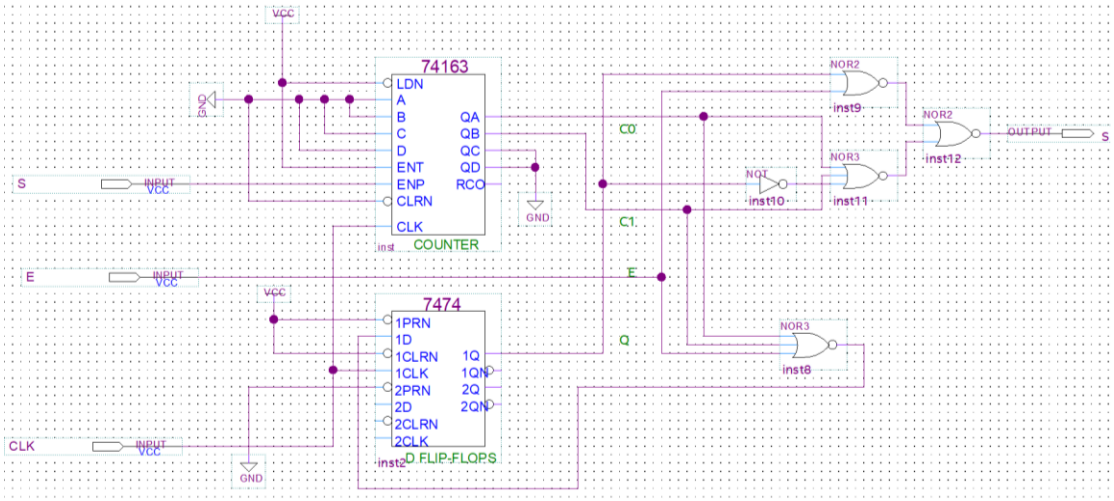


Fig.10 Overall Layout for Control Unit

Register Unit

Fourthly, we build the shift register unit. This unit takes in the output of the routing unit, D3-D0, Load A/B, and S (from control Unit) as inputs. We find that S1S0 controls the operating mode of the shift register chip. Based on the logic of design, we would only have 4 modes used: Reset, Hold, Shift Right, and Parallel Load. In addition, since two registers share D3-D0, we do not want to activate both load modes at the same time. In this case, we build a truth table to exam the relationship between Shift, Load, and S1, S0. Finally, we found that S1 (A/B) is aligned with Load A/B, and S0 (A/B) is aligned with S XOR Load A/B (Fig. 11). Now we can load them separately and shift at the same time.

Shift	LoadA	LoadB		S1(A)	S0(A)	S1(B)	S0(B)						
0	0	0		0	0	0	0						
0	0	1		0	0	1	1						
0	1	0		1	1	0	0						
0	1	1		1	1	1	1						
1	0	0		0	1	0	1		S1(A) = LoadA				
1	0	1		x	x	x	x		S1(B) = LoadB				
1	1	0		x	x	x	x		S0(A)=Shift XOR LoadA				
1	1	1		x	x	x	x		S0(B)=Shift XOR LoadB				

Fig.11 Logic between Shift, Load, S1, and S0

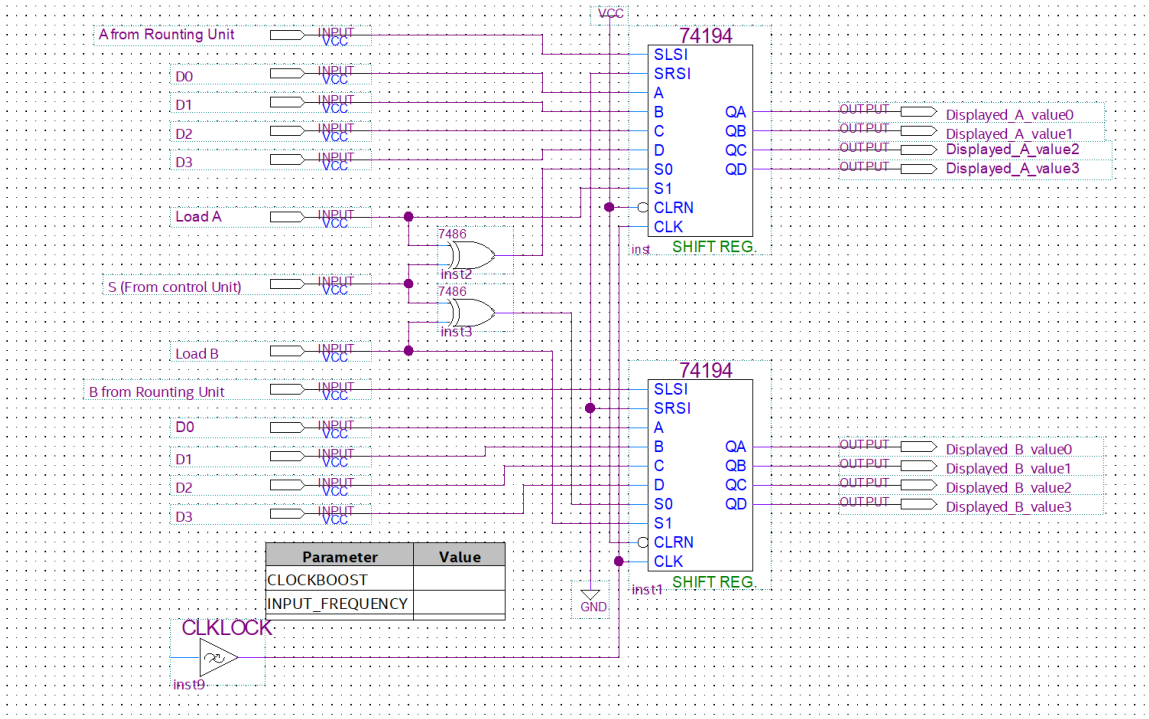


Fig.12 Overall layout for Register Unit

Breadboard View

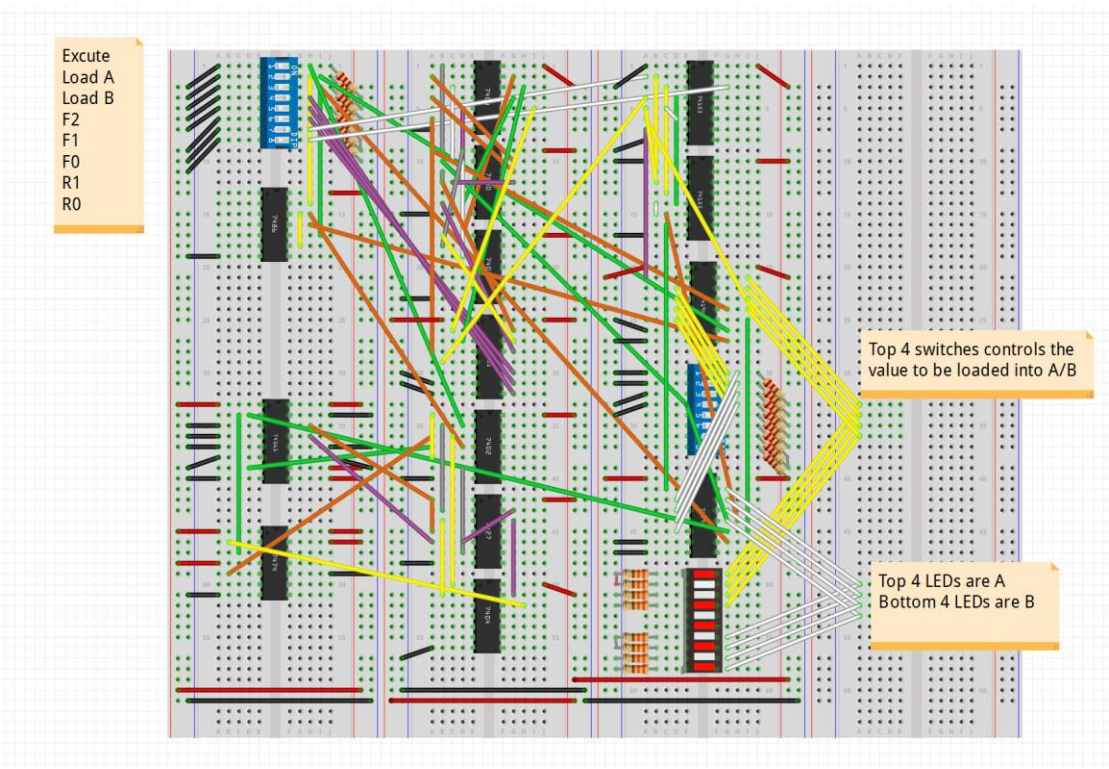


Fig.13 Breadboard view for whole circuit

8-bit Logic Processor on FPGA

For provided modules except ShiftReg8.sv and Control.sv, we keep them the same. For compute unit, we make no change since it functions computation bit-wise which means extending from 4 to 8 won't affect its way of design. For HexDriver, this is the provided driver to simulate LED on board for displaying register values. For Router Unit, it decides ways to load results which is also a bit-wise operation. Thus, it does not need to be modified. For Synchronizer.sv, it contains a debouncer circuit that eliminates the effect of contact bounce of switches as described in lab1.

To extend the 4-bit logic processor into 8-bit, we first need to change Register Unit. Originally, it can stores 4 bit, but we set two 4-bit registers in parallel by connecting the output of the first to the input of the second register, so that it can stores 8-bits information, as shown below.

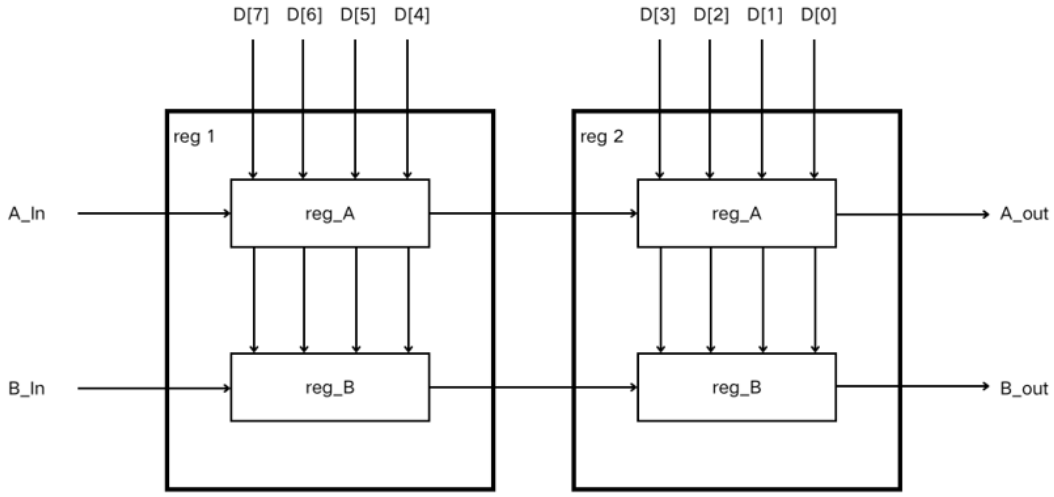


Fig.14 8-bits shift register using two 4-bits

The second thing we need to modify is the Control Unit. More specifically, the counter inside the unit. We employ the counter as a mechanism to generate signals used for shifting the register. The original 4-bit design only count down from 3 to 0, hence it can produce at most 4 shifting instructions. We modify the counter so that it can count down from 7 to 0. Other logic inside the Control Unit is kept unchanged.

Module descriptions

Module: compute.sv
Inputs: A_In, B_In, [2:0] F
Outputs: A_Out, B_Out, F_A_B
Description: This is the computation unit, it can perform 8 bitwise operation between A and B based on F values, and the result was stored in F_A_B, while A_Out, B_Out remains unchanged from A_In and B_In.
Purpose: Perform desired operation between A and B

Module: control.sv
Inputs: Clk, Reset, LoadA, LoadB, Excute
Outputs: Shift_En, Ld_A, Ld_B
Description: This is the control unit for the whole circuit. It mainly contains a Mealy State Machine and a flip flop. The first always_comb block determines the outputs based on the current state. The second always_comb block defines the logic for transitioning to the next state based on current state. Finally, the always_ff updates the current states based on the reset signal.
Purpose: After pressing the Execute bottom, the control unit will produce exactly 8 shifting signal to shift the register later on.

Module: HexDriver.sv
Inputs: clk, reset, in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: The module uses a loop to instantiate four nibble_to_hex converters, mapping each of the 4-bit inputs to their corresponding 7-segment display patterns. Then choose the hex_grid to determine which of the four LEDs is active.
Purpose: Convert binary into hexadecimal and then display them on the LEDs.

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, [7:0] hex_seg, [3:0] hex_grid
Description: This is the top-level module for the processor. When LoadA/B is high, the value of Din will be loaded into the register module for A/B. When Execute is high, the processor will do bitwise operations on each rising edge of Clk for 8 times. F chooses the computation type, and R chooses the output mode.
Purpose: This module is used as a combination of all other modules in the project for an 8-bit processor.

Module: Reg_4.sv
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [3:0] D
Outputs: Shift_Out, [3:0] Data_Out
Description: When the Load signal is high, the register loads the 4-bit input D into the register, which is an intermediate signal representing the next state of Data_Out. If Shift_En is high, the register shifts right by one position, taking Shift_In as the new leftmost bit, and shift rightmost bit into Shift_Out. If both Load and Shift_En are low, then the register keeps its original value. Finally, Reset will set Data_Out to 0000 and hence clean the register.
Purpose: The shift register load in data from Shift_In and output Shift_Out to compute unit. Finally, show the stored data through Data_Out.

Module: register_unit.sv
Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [3:0] D
Outputs: A_out, B_out, [3:0] A, [3:0] B
Description: It use two Reg_4 instances to store both A and B values. Both register share the same Clk and Reset botton. Ld_A and Ld_B are identical with each other.
Purpose: Same as Reg_4, expect this can store two sets of 4-bits value.

Module: Router.sv
Inputs: [1:0] R, A_in, B_in, F_A_B,
Outputs: A_Out, B_Out
Description: This module decides the output mode by two 4-to-1 mux to output A_Out and B_Out synchronously. It takes the R as a selector for each multiplexer. Data is chosen from A_in, B_in and F_A_B.
Purpose: This module is used to choose the routing selection and passed into register A and B.

Module: ShiftReg8.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: It put two instances of register_unit in parallel by connecting the output of the first to the input of the second register, so that it can stores 8-bits information, as shown in Figure 14.
Purpose: Same as register_uniy, expect this can store two sets of 8-bits value.

Module: Synchronizers.sv
Inputs: Clk, d
Outputs: q
Description: This module takes the signal d from the fpga button and switch. And outputs q for other modules to use. Firstly, it flops the input d twice. And then change the button when 2^COUNTER_WIDTH stable input cycles are recorded.
Purpose: It is used as a debouncer and a synchronizer for push buttons and switches.

RTL Block Diagram

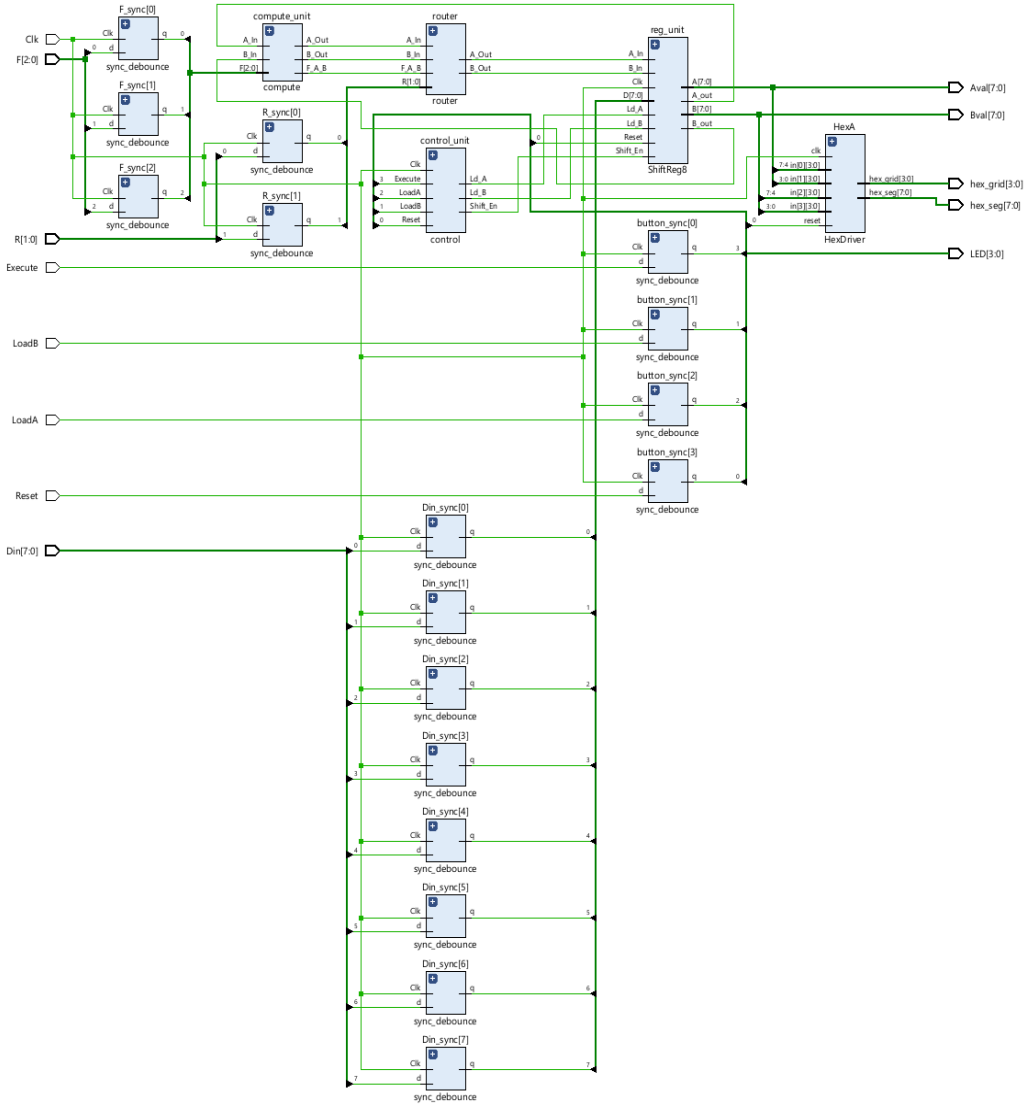


Fig.15 RTL block diagram

Simulation of Processor

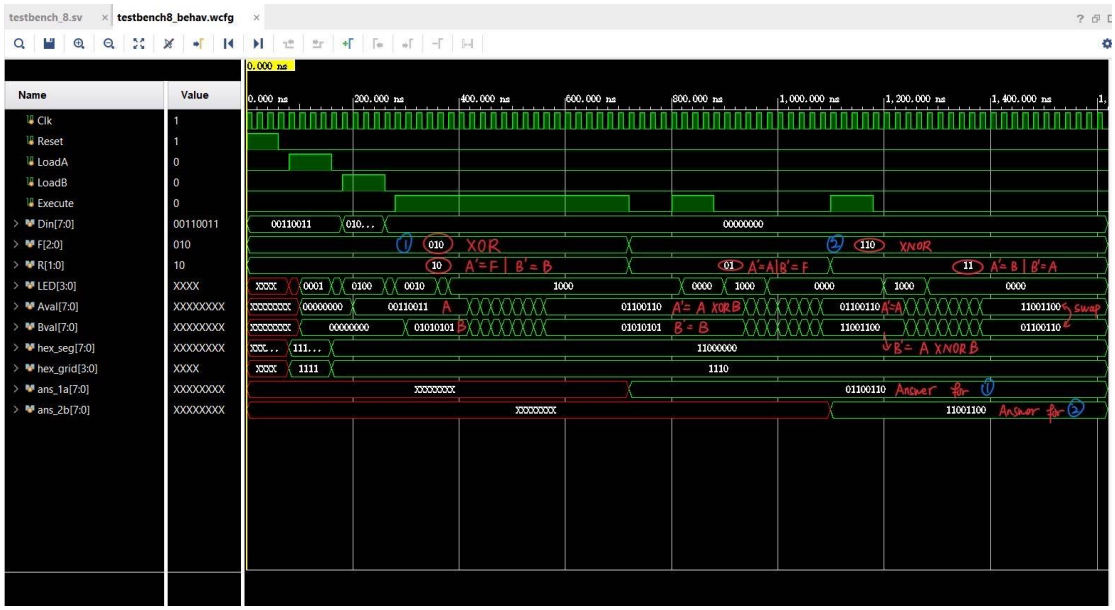


Fig.16 Simulation of processor

First, we load 00110011 into A, and 01010101 into B. Then we perform an XOR operation between A and B, and routing selection is A stores the result while B kept the same. The result is 00110011 XOR 01010101 = 01100110, which was stored in A as shown, and the result matched with ans_1a.

Second, we perform an XNOR operation between A: 0100110 and B: 01010101, and the routing selection is A stores the original value while B stores the result. The result should be 0100110 XNOR 01010101 = 11001100. And the result match with the ans_2b.

Finally, we make the routing 11 so that the value stored in A, B were switched, as shown above.

Vivado Debug Core

1. Make sure the synthesis is up to date.
2. Click on “Set Up Debug” and drag the signals that you want to monitor from “Schematic” into nets. In our case, we choose to view signals A, B, and Execute.
3. Generate Bitstream and program it into FPGA.
4. Finally view the waveform in debug core, as shown below.

Fig.17 Both A, B are 00000000

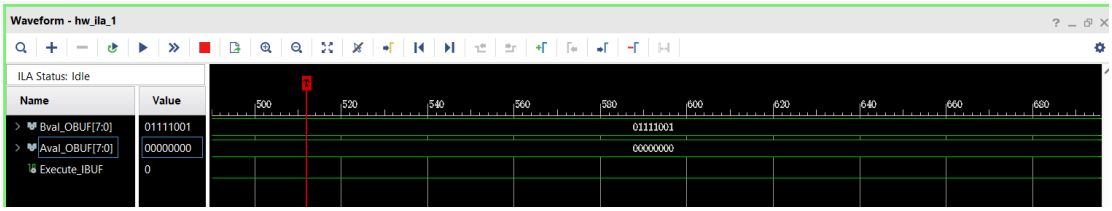


Fig.18 Load 01111001 into B

Fig.19 Load 10011101 into A

Fig.20 Perform XOR between A, B and stores result into B

Bugs Encountered

1. We implemented the Vcc switch and input pin in series. Although we want that when the switch is closed the data-in is 1 and the data-in is 0 when the switch is open, there appears to be that the input signal is sometimes at an intermediate value which is neither 1 or 0. This causes the output to differ from the precision based on the design circuit. We solved this by implementing a pull-up design with every switch on the breadboard.

2. In the first version of routing unit design, we want to implement two 4-to-1 mux on the same chip. However the output of the second chip is always bouncing. This is the result that we did not connect did not connect the probe pin to ground for the second 4 to 1 mux on the same chip. Then we take a closer look on the datasheet of the chip and then connect both probes to ground. The bug is finally solved.

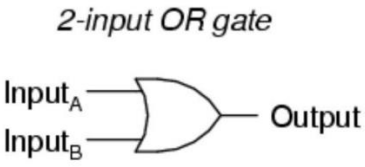
3. This one is the most awful bug we ever encountered. After testing one of the bidirectional chips on another empty breadboard, we implemented another two same chips on the breadboard that has other units. However, both of them show unexpected value when load and shift signals are passing in. We checked every pin's voltage using the multimeter and found that one of the output pins is high when it is supposed to be 0. We solved this by replacing them with register chips from a new set of lab kits. Finally we find out that 3 out of 4 shift register chips have some hardware issues which is very unlikely to happen.

4. When testing the control unit, we plan to see the output S signal regardless of other units as if we test other parts of the circuit. Then we disable the flip-flop, and only enable the counter and logic unit for S and Q+. However, the output is not the same as we expected. We solved this problem by disconnecting all the wires from the test part to the flip-flop. We find that those nodes will drain a lot of voltage while the chip is disabled.

Post Lab Question

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.

The simplest circuit that can optionally invert a signal which takes in two inputs is the XOR gate. The truth table for XOR gate is shown below, we found that one input can always be inverted/same by carefully chosen the other signal.



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Fig.21 XOR truth table

We use this design in the shift register unit. In this case, it is capable of controlling the input signal to S0 by XOR the signal S and LoadA/B.

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

A modular design is such a powerful tool in the development time and makes it more practical to test. First, we get to know the working progress of the processor without going deep into the concepts. Additionally, the design process of the circuit can be separated to different people so they can manage the process of each part at the same time. Meanwhile, the developers can choose to start on the easiest unit. In our case, it goes like this: ALU unit, routing unit, control unit, and then register unit. Most significantly, the developers are able to test the functionality of each part independently

before they wire the circuit all together. For the situation we met in this lab, we first debug all the Units separately to make sure they worked properly before putting them together. Surprisingly, the whole processor circuit worked at the first time when we connect each individual block together without further debugging.

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

We start the design of the state machine by a Moore machine because it is easy to see that we have 6 states: Reset, Shift3-0, and Halt. Now here comes the greatest difference between Mealy and Moore machine: the Moore machine only depends on the current state and the Mealy machine depends on a combination of the current state and current inputs. In other words, we can use less state to build the FSM with Mealy machine and in this case only one flip-flop is needed to store the state which indicates the progress of the circuit.

What are the differences between vSim and Vivado Debug Cores? Although both systems generate waveforms, what situations might vSim be preferred and where might debug cores be more appropriate?

The main difference between vSim and Vivado Debug Cores is that vSim generates waveforms based on those sv files in the computer. While Debug Cores generates waveforms based on the hardware implementation on the Urbana board. In addition, it provides a tool for real-time debugging that we can see the output while operating the FPGA board. vSim is preferred in the situation that we want to take a quick look at some design to see if it works out correctly. While debug cores are preferred when designers finish some complex hardware design and want to test its behavior on the real environment.

Conclusion

The lab mainly focusing on designing and building a bit-serial logic operation processor. The processor integrates two 4-bit shift registers, multiple logic gates, a counter, and a finite state machine as a control unit. It is capable of executing 8 different logical functions and routing the outcomes in four ways. From this lab, we have an overall review of knowledge from previous courses and gain better comprehension of shift registers, bitwise logical operations, and the state machine, more importantly, the significance of modular design as a engineer.