ECE 385

Fall 2023

Experiment #2

Logic Processor

Yunxuan Yang, Yuqi Wen
2023/9/10
TA: Tianhao Yu

1) Introduction

The logic processor developed for this lab performs basic arithmetic and logical operations on 4-bit data and can be extended to 8-bits. The operations supported are addition, subtraction, bitwise AND, and bitwise OR. The processor consists of two 4-bit registers, namely A and B, which can be manually loaded by the user for computation.

2) Operation of the Logic Processor

a. Loading Data into Registers

To load data into the A and B registers, the user must first flip the 4-bit input switches A/B0-A/B3 to represent the 4-bit data. After completing the setup, the user should activate the "LOAD A" switch, which triggers the register to simultaneously load the specified data. Turn off the switch once the operation is finished. For register B, use the same input switches to input the required values. To load them into the register, turn the "LOAD B" switch on. Turn it back off once the desired values are loaded. To confirm that the data has been properly loaded, check the corresponding LEDs that display the register's values.

b. Initiating a Computation and Routing Operation

To accurately configure the operation and define the routing paths, adjust the switches for F0-F3 and R0-R1 according to the values specified for the desired function, as outlined in the tables found in Lab Manual 2. After correctly setting the values for F0-F2 and R0-R1, along with the appropriate values for registers A and B, you can initiate the operation by toggling the "Execute" switch.

TABLE 1: Functions

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

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

3) Written description, block diagram and state machine diagram

- **Register Unit:** The task of this unit is to store the two inputs in register A and B and pass them to the computational unit. It is construct with two 4-bit Bidirectional Universal Shift Register (74194A/E), one for A one for B. When the Load A/Load B function is activated for each register, data bits D3-D0 are inputted into each register via parallel input pins 3 to 6. Both registers then display these bits on four LEDs each, which are connected from outputs QA to QD. Given that our computing unit processes data on a bit-by-bit basis, the initial inputs in the register, which are entered through parallel loading, need to be sequentially transferred into the computational unit. This transfer begins with the least significant bit, QD. As a result, we needed to execute a rightward shift of the bits. To achieve this, we linked Load A/Load B to S1 on both registers. Simultaneously, the Shift bit was connected to S0 on both registers. This Shift bit is determined by applying the OR function to Load A/Load B and S - the latter being the control unit's output. We made this connection because the shift action is triggered when S1 is in a Low state, and S0 is in a High state. After the computation is complete, the result might be saved in a register unit, determined by the R1 and R0 signals in the routing module.

- **Computation Unit:** This unit's primary function is to perform 8 distinct bitwise operations on a singular bit received from the register unit, specifically the least significant bit emerging from QD after a rightward bit shift. The unit's structure comprises an 8:1 Multiplexer, a Quad 2-input NAND(7400), a Quad 2-input NOR(7402), and a Quad 2-input XOR.(7486)
  An 8:1 MUX was incorporated due to the necessity of selecting one out of eight possible bitwise operations. The user determines this selection based on the F2, F1, and F0 signals, which are inputted into the S2, S1, and S0 ports of the chip.

  Several key operations are executed in this unit:
  The NAND operation is conducted using a single NAND gate.
  The AND operation is achieved by inverting the NAND gate's output via the NAND gate as inverter.
  The NOR function is realized with one NOR gate.
  The OR function is produced by inverting the NOR gate's output.
  The XOR operation uses XOR gates (as illustrated in figure 1).
  The XNOR operation is derived by inverting the output from the XOR mechanism.
  For the 1111 operation, the MUX's input was set to a high state.
  For the 0000 operation, the MUX's input was designated as low.
  On the integrated chip, the strobe bit remains in a low state.

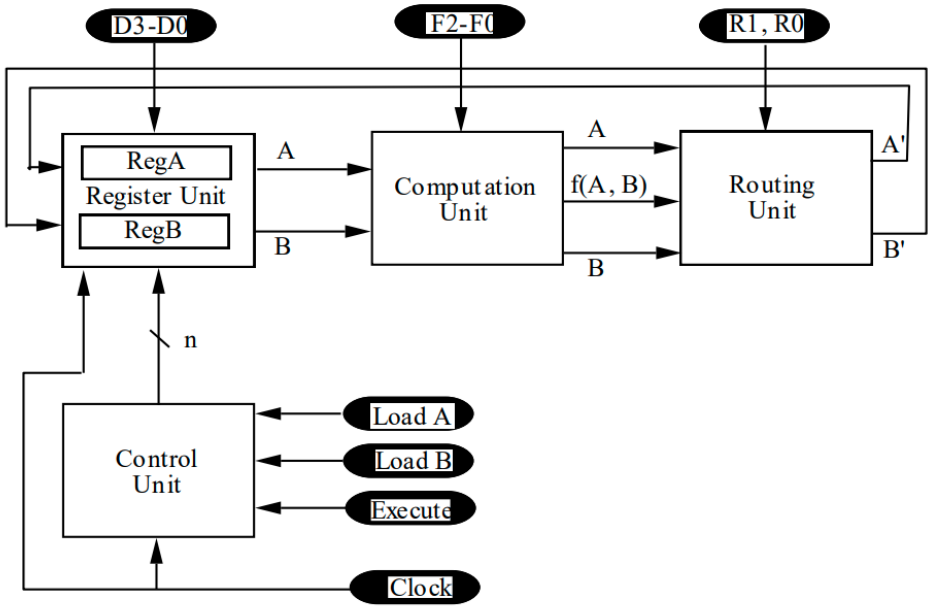
Function Selection Inputs			Computation Unit Output
F2	F1	F0	f(A, B)
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

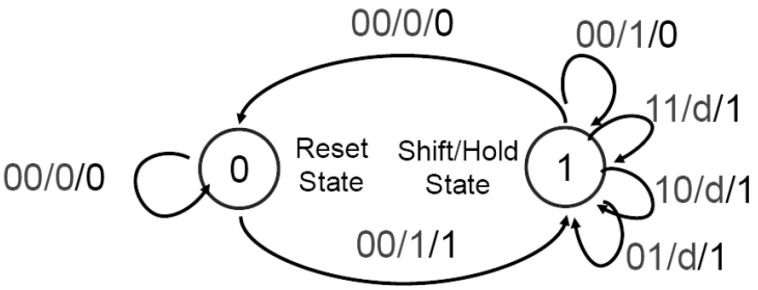
- **Routing Unit:** This unit receives inputs from R1 and R0 to determine which signal should be sent back to registers A and B. This section of the circuit features two 4-to-1 multiplexers. One multiplexer manages the signal being sent back to register A, while the other oversees the signal returning to register B. Each 4-to-1 multiplexer accepts three types of input: A, B, and f (A, B).

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

- **Control Unit:** The control unit receives three inputs: C1, C0, and EXEC. To build the control unit, a Mealy state diagram was drafted, indicating that the current state can influence the output signal 'Sh' (shift). The primary function of the control unit is to generate this 'Sh' signal at the appropriate time. Initially, we need to define the implementation for the current state, denoted as Q. The accompanying truth table shows that the next state Q+ relies on the inputs: Exec, Q, C1, and C0. Once Q was accurately configured, we began the implementation of the 'Reg. Shift' signal. This 'Shift' signal is then combined with the 'Load' signal to produce the signals S1 and S0, which are used to control the registers.

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





Each transition arc: C1C0/E/S

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

In Lab2-1, we employed a Mealy state diagram to design our control unit, which has just two states: "Reset State" and "Shift/Hold State."

In the "Reset State," the initiation of the first cycle of bitwise operations occurs when the 'Execute' signal transitions from 0 to 1. Once the operation is triggered, the computation unit continues to operate for four cycles, regardless of the state of the 'Execute' signal. During these four cycles, the count signals sequence through the states: 00, 01, 10, 11, allowing all four bits to be computed.

After completing the four cycles, the system enters the "Shift/Hold State," where it waits for the 'Execute' signal to transition back from 1 to 0. This holding mechanism is essential to avoid unintentionally triggering another set of four cycles if the 'Execute' button is held down too long, effectively keeping the signal at a constant 1.

Once the 'Execute' signal goes from 1 back to 0, the system returns to the "Reset State," preparing it for another operation cycle.

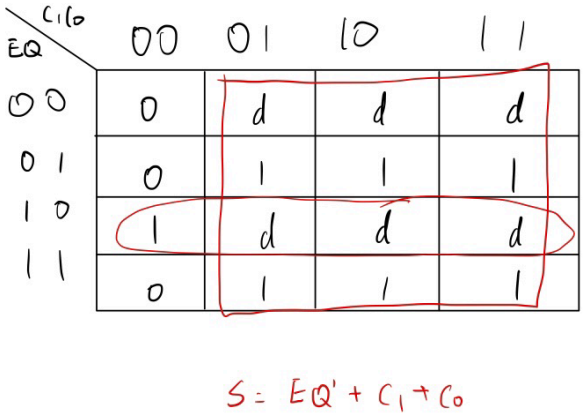
4) Design Steps and Detailed Circuit Schematic

a. Design Procedure

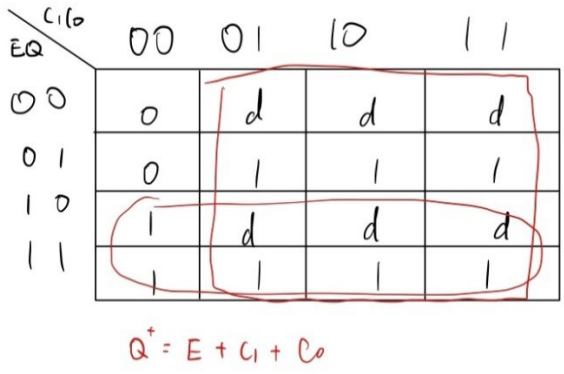
We used K-maps to derive the next state logic in the control unit. All logic expressions were transformed to use NAND/NOR gates because the discrete logic chips in the kit only support NAND/NOR operations.

K-Maps for Control Unit Output Signals to Register Unit

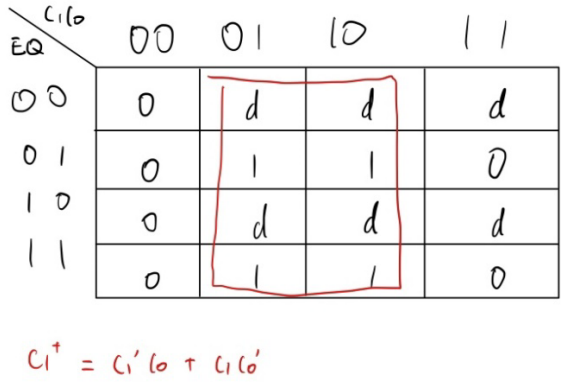
Exec. Switch ('E')	Q	C1	C0	Reg. Shift ('S')	Q^+	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	D
0	0	1	0	d	d	d	D
0	0	1	1	d	d	d	D
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	D
1	0	1	0	d	d	d	D
1	0	1	1	d	d	d	D
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0



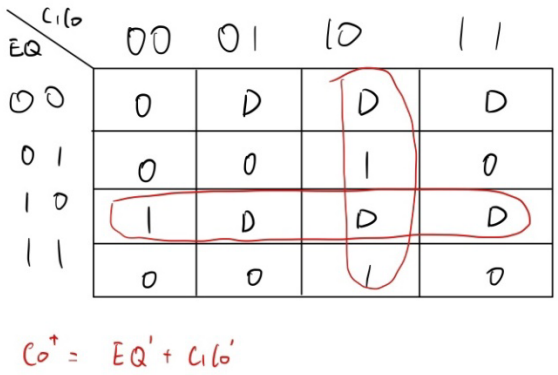
We used a two-input NAND gate to calculate $\overline{\bar{E} + Q}$, then used three-input NOR gate to get $\overline{\overline{\bar{E} + Q} + C_1 + C_0}$. After that, we applied an inverter to get S.



We used a three-input NOR gate to get $\overline{E + C_1 + C_0}$, then we applied an inverter to get Q^+.



We used two two-input NAND gates to calculate $\overline{\overline{C_1}C_0}$ and $\overline{C_1\,\overline{C_0}}$ separately, then applied a two-input NOR gate along with an inverter.



We used two two-input NAND gates to calculate $\overline{E\bar{Q}}$ and $\overline{C_1\,\overline{C_0}}$ separately, then applied a two-input NOR gate along with an inverter.

Design Considerations

The first consideration is the control unit. Since we have a large K-map, and we need to separate it to four parts. We also need to optimize the Boolean expression of each small K-map so that we might use less gates in our design, making the design less complicate. After implemented the Boolean expressions, we need to consider what chips to choose. We tried three-input and four-input to find the best way for the circuit. Also, after comparing the 4-bit Sync Counter (74163E) with the 4-Bit Binary Counter (74161N), the Binary Counter was identified as the optimal fit, offering the most suitable modes and functionalities.

After that, we went on to design computational unit and routing unit. We chose an 8 to 1 MUX (74151N) since this is the most intuitive decision we came up with. We first implemented the 8 results of computational unit separately. With inputs F0-F3, we can directly get the output from computational unit. Then we connected the output from the 8 to 1

MUX to a 4 to 1 MUX to complete the routing unit together with input R0 and R1. This might not be the easiest way to design the circuit, but it is the most straightforward one.

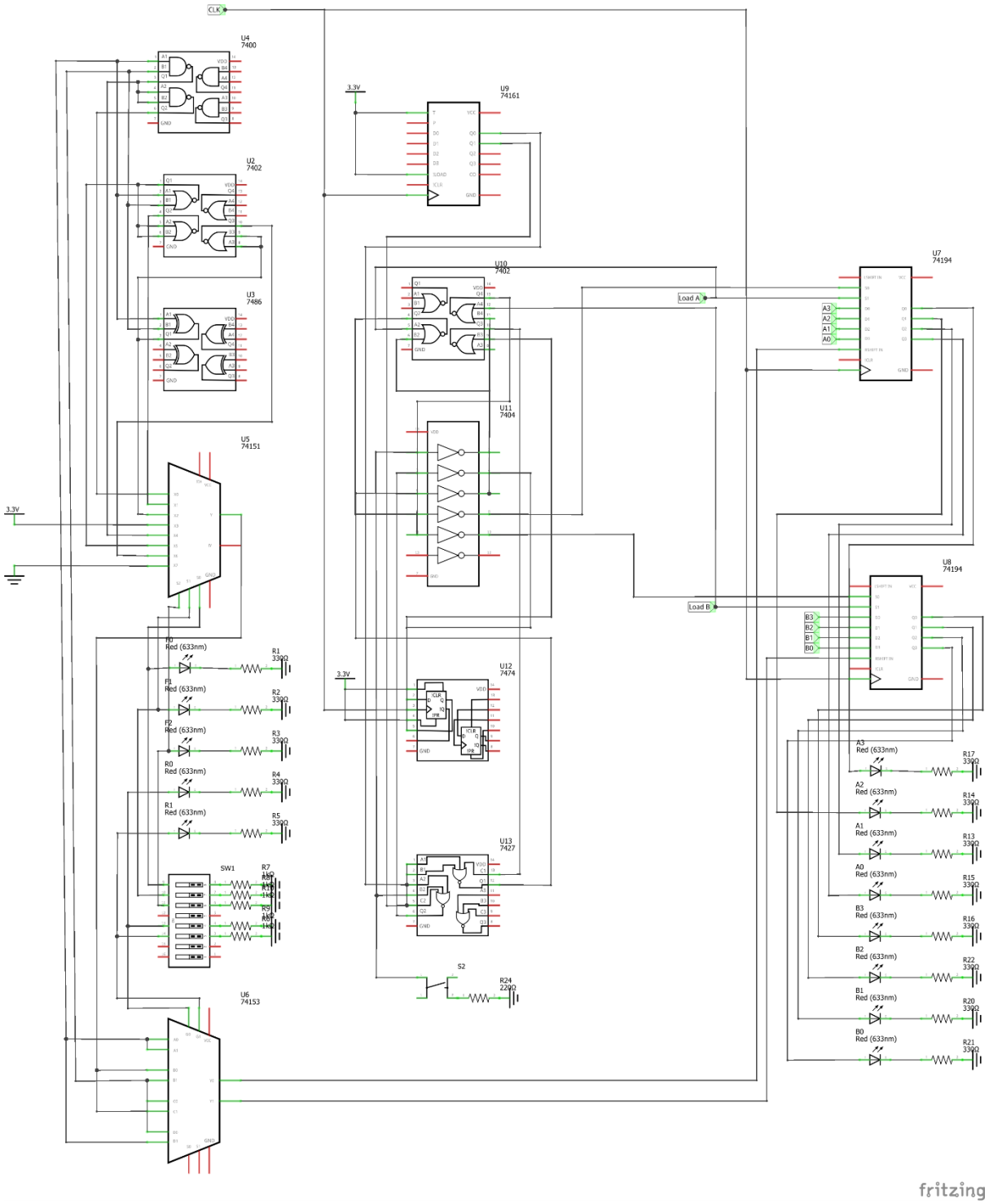
For the register unit, we were confused about how to assign values to two 4-bit register. If we used 8 switches, that would be too complex. We then realized that we could use the same four switches to assign data since we load the data to each register separately. With this design, we successfully make our register unit denser.

b. Circuit Schematic

Computation Unit

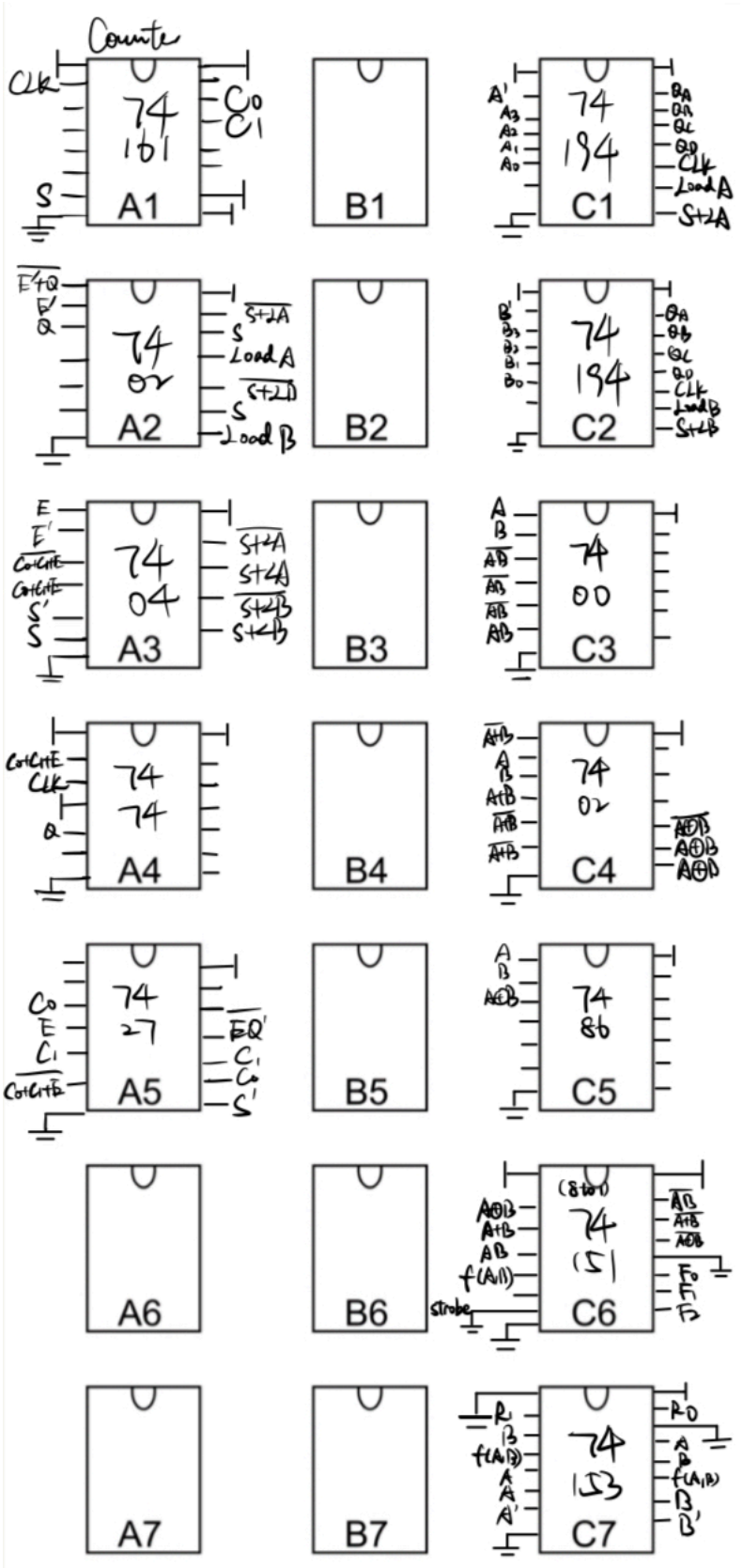
Control Unit

Register Unit



Routing Unit

5) Breadboard view / Layout sheet



6) 8-bit logic processor on FPGA

a. Summary of .SV modules and changes

Module: Reg_4.sv:

Changes:

- Change D and Data_out from 4 bits [3:0] to 8 bits [7:0]
- Reset function sets Data_out to eight 0s instead of 4
- Shift function sets Data_out to be {Shift_in, Data_out [7:1]}

Module: Register_unit.sv

Changes:

- Change input D from 4 bits [3:0] to 8 bits [7:0]
- Change Register A, B from 4 bits [3:0] to 8 bits [7:0]

Module: Control.sv

Changes:

- Since it needs to execute 4 more cycles, we added 4 additional operating states: G, H, I, J, placed after the reset state and before the hold state.

Module: Processor.sv

Changes:

- In the 4-bit implementation, the Hexdriver function is called twice: once for Register A and once for Register B. To extend this to an 8-bit system, Hexdriver will need to be

invoked four times in total. Specifically, it will be called twice for each register: once for the higher-order bits [7:4] and once for the lower-order bits [3:0]. This will ensure that both 8-bit registers, A and B, are fully utilized and managed by the Hexdriver function.

- Change number of bits for Din, LED, Aval, Bval, A, B, Din_S.

Module: Compute.sv (no changes)

Module: Router.sv (no changes)

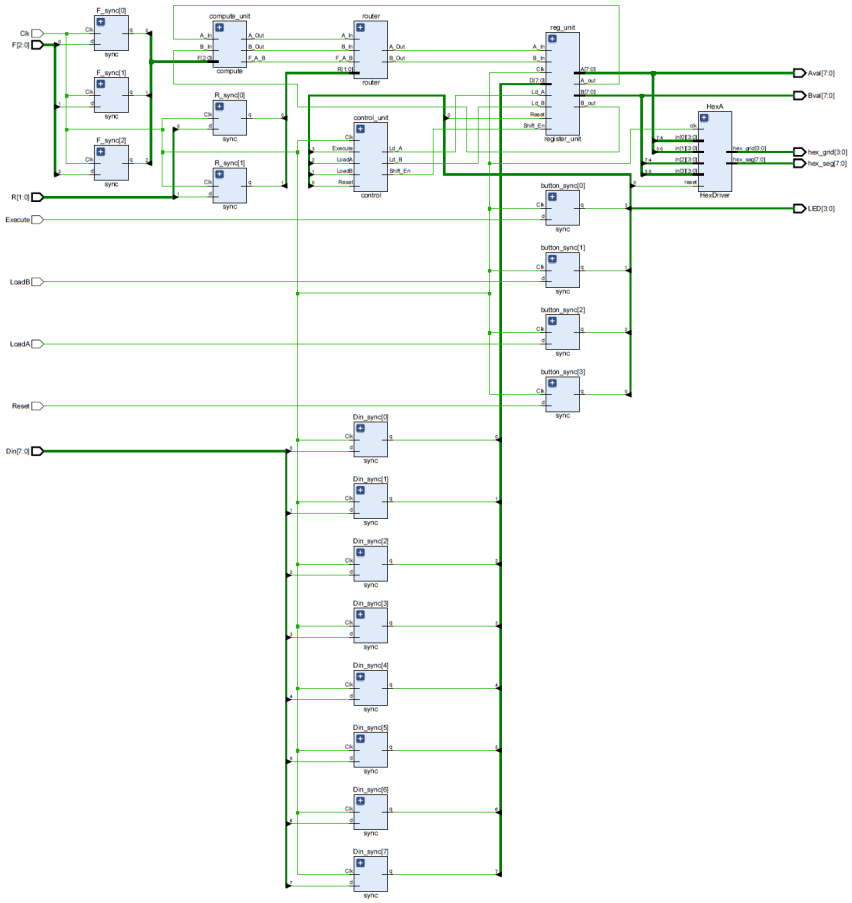
Module: Hexdriver.sv (no changes)

Module: Sychronizers.sv (no changes)

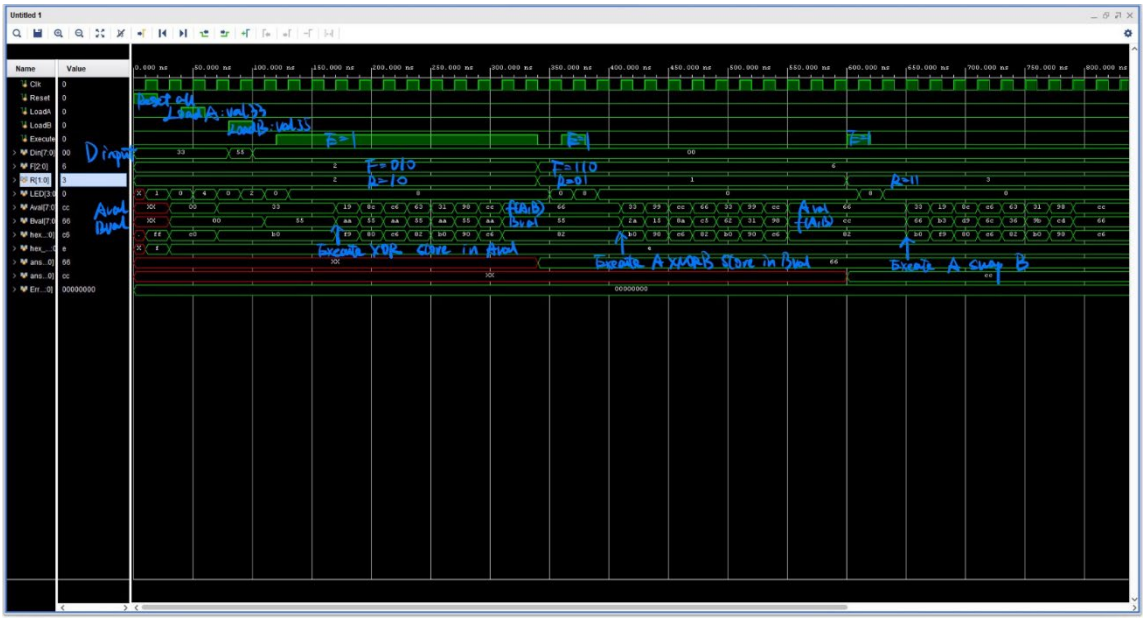
Module: Testbench.sv (no changes)

Module: Testbench8.sv (no changes)

b. RTL block diagram



c. Simulation of the processor



d. Steps to generate Vivado Debug Core trace

1. Under the pull-down menu of “Open Synthesized Design”, choose “Set up Debug”

2. Add Nets that we want to show in the waveform, here we choose Execute, Aval, and Bval. Also, assigned Probe Type, setting Execute to Trigger, Aval and Bval to Data

3. Click Finish and go on to generate Bitstream. This will run implementation as well.

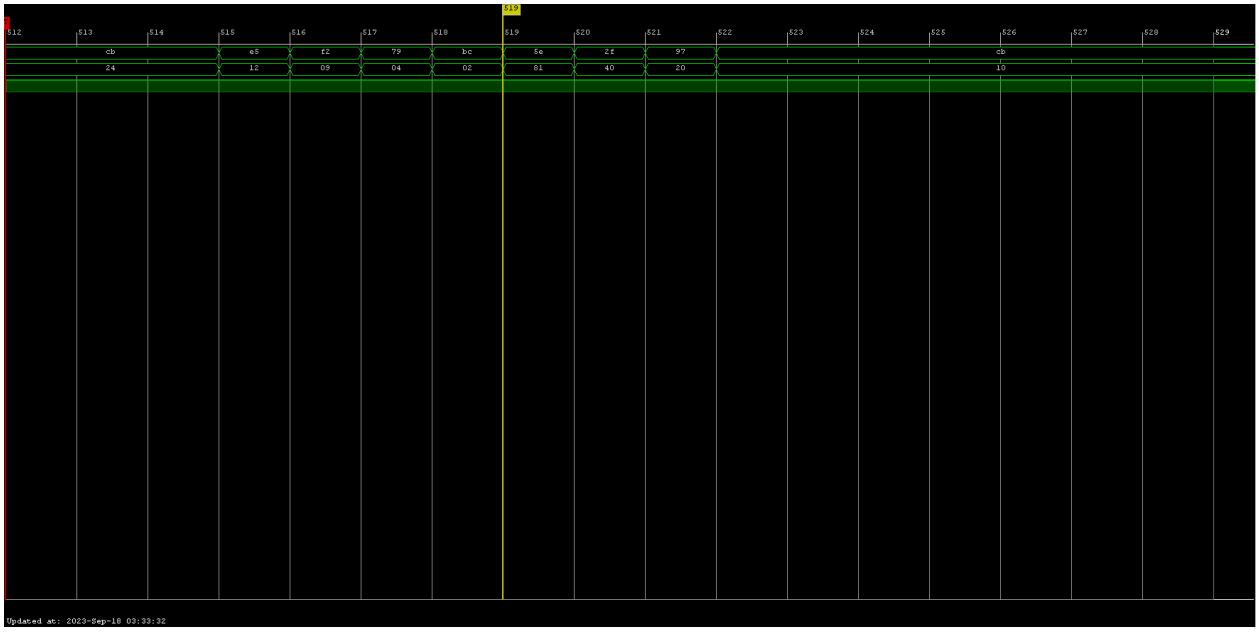
4. We then program the device with the Debug probes file. When finished, a waveform will pop up

5. Setup trigger, we set the value of Execute to “B” both transition

6. Then we click run trigger to wait for execution

7. Once Execute button hit, the resulting waveform will be showed on the screen

e. output of the debug core (cb NOR 24)



7) Description of all bugs encountered

Lab 2.1:

1. Wrong boolean expressions from K-map. Correct: Rewrite the expressions.
2. Serie switch with LED and register, causing the voltage too low to function correctly. Correct: Changed the design to parallel connection.

Lab 2.2:

1. Not setting top level correctly. Change: Set correct top level.
2. Didn’t assign the pins with correct voltage, causing all pins not functioning after programmed. Correct: Followed the pin assignment table to assign pins.

8) Conclusion:

a. Summary

Lab 2.1 asks us to create a system capable of performing bitwise calculations. The most challenging component to design is the control unit, which includes signals such as Q, Exec, C1, and C0. It's crucial to have a deep understanding of the Mealy state diagram for this task.

In Lab 2-2, we are introduced to the Vivado software and FPGA boards. The primary objective here is to comprehend the existing code and make minor adjustments to it. Understanding the function of the testbench is also vital to ensure we are on the right track.

b. Answer to post-lab questions

1. 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.

An XOR gate is the simplest two-input, one-output circuit capable of optionally flipping a signal. It can either preserve a 0 or turn it into a 1 and can keep a 1 unchanged or flip it to 0. For our lab, had we opted for a 4:1 MUX rather than an 8:1 MUX in our computation module, this characteristic would have been handy. With this approach, we could execute 4 out of the 8 logic functions and just invert them as necessary without adding extra logic.

However, for clarity in our lab, we went with an 8:1 MUX accompanied by an inverter, which, though less efficient, was more intuitive for us.

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

It enables the developer to create and assess each module or component separately. This facilitates quicker problem-solving as issues can be tackled at a segmented level. Debugging a simple circuit with a single function is considerably easier than troubleshooting a more complex circuit responsible for multiple tasks. This approach also enhances testability, as the developer can evaluate the results of each individual module to confirm they're correct. This is in contrast to having to determine the accuracy of the entire system's output and then retracing steps within the circuit if there's a discrepancy.

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

As noted previously in our report, we chose to base our circuit design on the Mealy State machine instead of the Moore State Machine. In the Mealy setup, output values are shaped by both the current state and the present inputs. Conversely, in the Moore setup, output values depend solely on the current state. We found the Mealy machine more suitable for our design since it requires fewer states than the Moore machine. This translates to a reduced need for gates and flip-flops in the design. Furthermore, the Mealy machine responds more promptly to input changes compared to the Moore machine. On the other hand, the Moore machine, with its numerous states and transitions, can sometimes be simpler to design.

4. 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?

vSim is a part of the ModelSim suite and is primarily a simulation tool. vSim allows you to simulate your HDL (Hardware Description Language) designs, visualizing how your digital logic will behave under different conditions without needing the physical hardware.

Vivado Debug Cores are hardware debug tools provided by Xilinx for their Vivado Design Suite. The debug cores are embedded into the FPGA fabric and allow for real-time monitoring and capturing of signals within a design that is actually running on an FPGA.

Situations where vSim might be preferred:

1. Early in the design cycle, when you're testing the basic functionality of your digital logic.

2. When you do not have access to the physical FPGA hardware.

3. For designs that are not targeted for any specific FPGA but are more conceptual.

Situations where Vivado Debug Cores might be more appropriate:

1. After deploying your design to an FPGA and you need to monitor or debug its behavior in real-time.

2. When validating the interaction of your design with other components on the board or external devices.

3. When you want to observe phenomena like metastability, timing issues, or other real-world effects that are hard to capture in simulation.

c. Unclear part:

1. We are provided with chip 74161 as 4-bit up/down counter while on the data sheet there is only layout of 74169chip which is also 4-biy up/down counter, it should be clarified in the documentation.