# ECE 385

Spring 2024

Experiment 2

# A Logic Processor

By Wendi Wang (wendiw2)

## 1) Introduction.

This bit-serial logic operation processor can operate on two 4-bit binary numbers, storing in register A and B respectively. This processor can calculate 8 different functions (f (A, B) = A AND B, A OR B, A XOR B, 1111, A NAND B, A NOR B, A XNOR B, 0000). After the calculation, it can route the result of those operations in 4 different ways: no storage of f (A, B), store f (A, B) into register B, store f (A, B) into register A, no storage of f (A, B) but swap A and B.

## 2) Operation of the logic processor.

### a. The sequence of switches to load data.

Flipping Load A switch to high voltage (keeping Load B switch and Execute switch at low voltage) can load D3-D0 (which are controlled by switches) into A register.
Flipping Load B switch to high voltage (keeping Load A switch and Execute switch at low voltage) can load D3-D0 (which are controlled by switches) into B register.
The sequence of flipping switches to load data:
Flip D3-D0 -> Switch on Load A -> Switch off Load A -> Flip D3-D0 -> Switch on Load B -> Switch off Load B

### b. The sequence of switches to initiate a computation and routing operation.

Switch off Load A -> Switch off Load B -> Flip F2-F0 -> Flip R1->R0 -> Switch on Execute (where Load A, Load B, F2-F0, R1-R0 and Execute are all controlled by switches)

## 3) Written description, block diagram and state machine diagram.

### a. Written description.

**The Register Unit:**

The Register Unit consists of two 4-bit shift registers to store values of Register A and Register B respectively. These registers are loaded through flipping switches D3-D0 and switching on Load A/Load B switches, and they are under the control of the control unit, especially controlled by the Shift bit (the output of the Control Unit), Load A switch, Load B switch, and the clock signal (shared between Register Unit and Control Unit). The output of the Register Unit is two 4-bit values, and the Register Unit will deliver exactly one bit from each of two 4-bit values (namely A and B) to the Computation Unit as inputs for further computation.
In the meanwhile, the Register Unit accepts two bits from the routing unit, namely A' (new A) and B' (new B). Since the Register Unit contains two 4-bit shifter registers Register A and Register B, we can name the 8 bits in this unit by <A3, A2, A1, A0> and <B3, B2, B1, B0> respectively. If A = A0, then when A' is fed into Register A, the value stored in Register A will become <A', A3, A2, A1> (right shift). If A = A3, then when A' is fed into Register A, the value stored in Register A will become <A2, A1,

A0, A'> (left shift). Similarly for Register B, if B = B0, then when B' is fed into Register B, the value stored in Register B will become <B', B3, B2, B1> (right shift). If B = B3, then when B' is fed into Register B, the value stored in Register B will become <B2, B1, B0, B'> (left shift). These processes will repeat exactly 4 times.

**The Computation Unit:**

The Computation Unit accepts 2 input bits A and B from the Register Unit (from Register A and Register B respectively). The function selection inputs F2-F0 in the Computation Unit are controlled by flipping switches. The Computation Unit outputs the logical function f (A, B) specified by <F2, F1, F0> and outputs the A and B inputs unchanged as well. All the outputs will be fed into the Routing Unit.

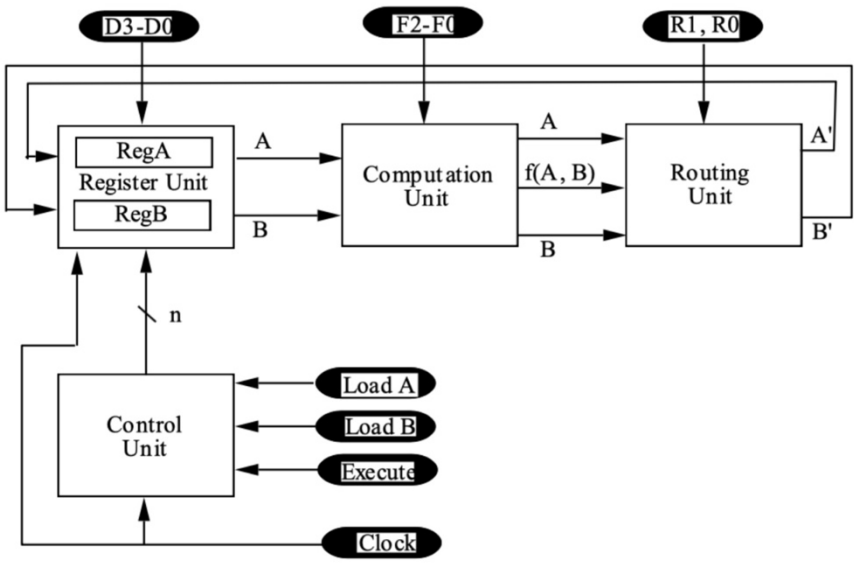
**The Routing Unit:**

The Computation Unit accepts 3 input bits f (A, B), A, and B from the Computation Unit. The routing selection inputs R1-R0 in the Routing Unit are controlled by flipping switches. The Routing Unit outputs A' and B' (new A and new B) specified by <R1, R0>, and both A' and B' will be fed into the Register Unit.

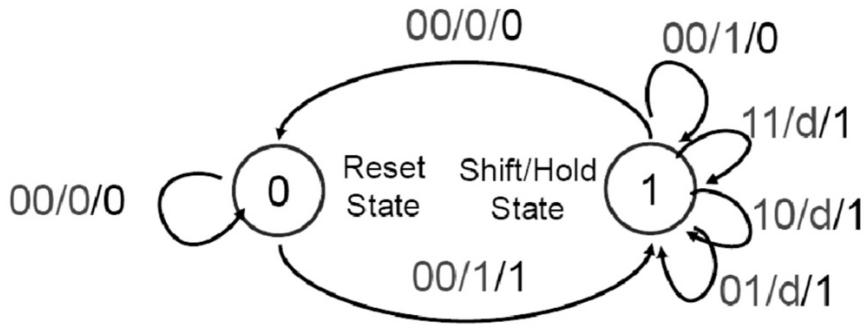
**The Control Unit:**

The Control Unit accepts 4 inputs: Load A, Load B, Execute, and the clock signal. Load A, Load B, and Execute are all controlled by flipping switches. The Load A and Load B will perform parallel loads from D3-D0 values into Register A and Register B respectively. Execute bit tells the Control Unit that the select switches <F2, F1, F0> and the values in Register A and Register B are ready for execution, and the Control Unit should begin the computation cycle if Execute bit is held high. The Control Unit then shifts the Register Unit exactly 4 times and halts automatically until next time Execute bit is held high or the counter starts counting again.

**b. High-level block diagram.**



**c. State Machine Diagram.**



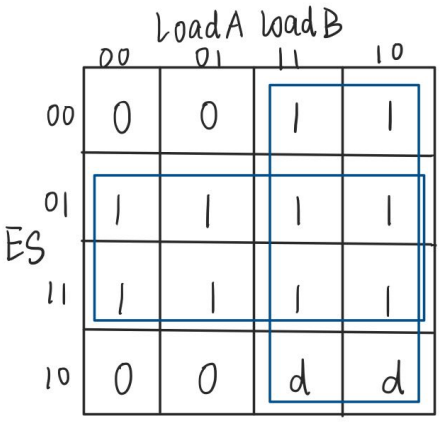

It is a Mealy state machine. There are 2 states in this Mealy state machine, namely Reset State (represented by 0) and Shift/Hold State (represented by 1). XX/Y/Z values on every arc has the meaning as follows: XX is the binary flip-flop value of the binary counter. Y represents Execute bit, and Z represents the output bit Shift. XX and Y are input bits, and Z is the output bit of the state machine.

When output bit Shift becomes high, the binary counter begins counting down. After 4 cycles (namely the counter counts following 00 ->01->10->11->00), if Execute bit is still held high, the output bit Shift will be held low, and the state machine stays at Shift/Hold State. If after 4 cycles the Execute bit is held low, the output bit Shift will be held low, and the state machine transfers to Reset State, staying at Reset State until Execute bit is held high or the counter starts counting again.

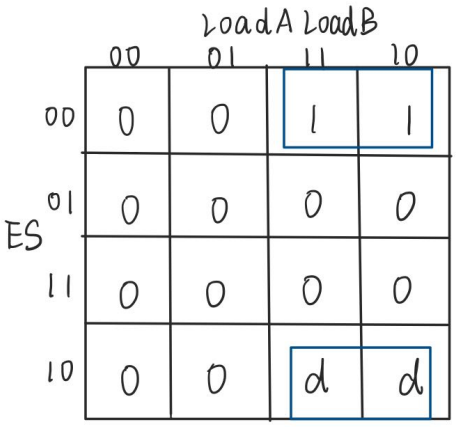
**4) Design steps taken and detailed circuit schematic diagram.**

**a. Written procedure of the design steps taken.**

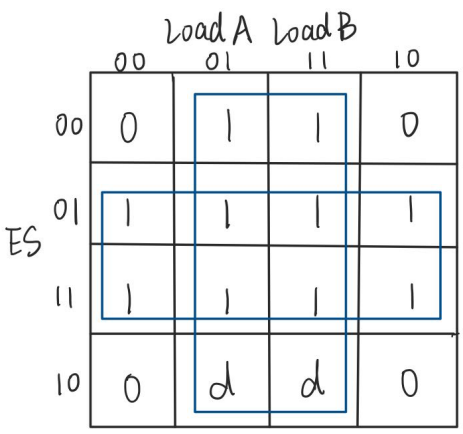
*K-maps:*




A_S0 = Load A + S
(Port 9 of Shift Reg. A)

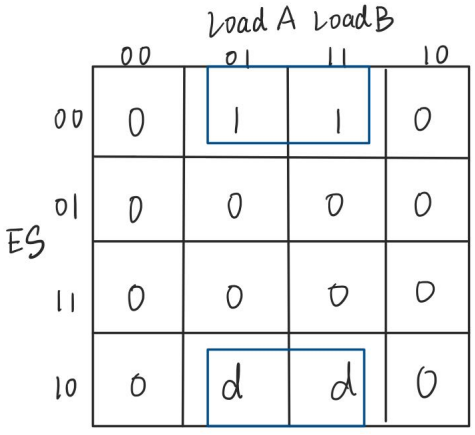



A_S1 = Load A · S'
(Port 10 of Shift Reg. A)



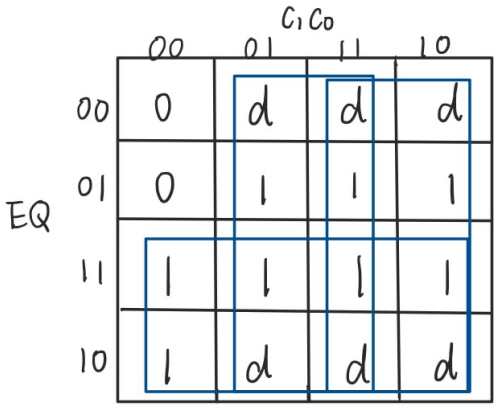

B_S0 = Load B + S

(Port 9 of Shift Reg. B)

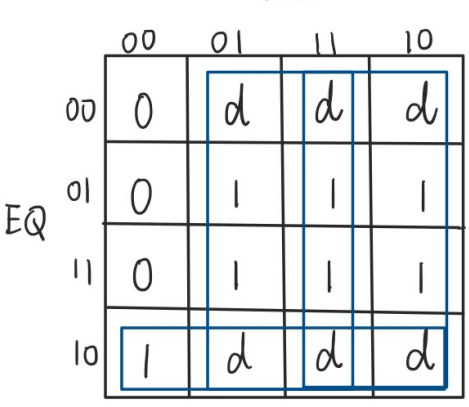



B_S1 = Load B · S'

(Port 10 of Shift Reg. B)




$Q^+ = C_0 + C_1 + E$




$S = C_0 + C_1 + EQ'$

*Truth Tables:*

| 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 |

| 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 |

| Execute (E) | Load A | Load B | S | A_S0 (Port 9 of Shift Reg. A) | A_S1 (Port 10 of Shift Reg. A) | B_S0 (Port 9 of Shift Reg. B) | B_S1 (Port 10 of Shift Reg. B) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | d | d | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | d | d |
| 1 | 1 | 1 | 0 | d | d | d | d |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

TRUTH TABLE (Shift Register A & B)

| OPERATING MODE | INPUTS | | | | | | | OUTPUT | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | CP | $\overline{MR}$ | S1 | S0 | DSR | DSL | $D_n$ | $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
| Reset (Clear) | X | L | X | X | X | X | X | L | L | L | L |
| Hold (Do Nothing) | X | H | l | l | X | X | X | $q_0$ | $q_1$ | $q_2$ | $q_3$ |
| Shift Left | ↑ | H | h | l | X | l | X | $q_1$ | $q_2$ | $q_3$ | L |
| | ↑ | H | h | l | X | h | X | $q_1$ | $q_2$ | $q_3$ | H |
| Shift Right | ↑ | H | l | h | l | X | X | L | $q_0$ | $q_1$ | $q_2$ |
| | ↑ | H | l | h | h | X | X | H | $q_0$ | $q_1$ | $q_2$ |
| Parallel Load | ↑ | H | h | h | X | X | $d_n$ | $d_0$ | $d_1$ | $d_2$ | $d_3$ |

*Transformations of SOPs:*

For each of the SOPs stated above, to accommodate to NAND/NOR chips, we can re-write them as follows:

$$A_{S0} = Load\ A + S = NOT\ (Load\ A\ NOR\ S)$$

$$B_{S0} = Load\ B + S = NOT\ (Load\ B\ NOR\ S)$$

$$A_{S1} = Load\ A \cdot S' = NOT\ (Load\ A\ NAND\ S')$$

$$B_{S1} = Load\ B \cdot S' = NOT\ (Load\ B\ NAND\ S')$$

$$Q^+ = C_0 + C_1 + E = NOT\ (C_0\ NOR\ C_1\ NOR\ E)$$

$$S = C_0 + C_1 + E \cdot Q' = NOT\ \big(C_0\ NOR\ C_1\ NOR\ (E \cdot Q')\big)$$

$$= NOT\ (C_0\ NOR\ C_1\ NOR\ (NOT(E\ NAND\ Q')))$$

*Design considerations:*

To control the length of the computation cycle to be 4 each time Execute switch is pressed down, there must be some measures done to control the operations of the binary counter and shift registers.

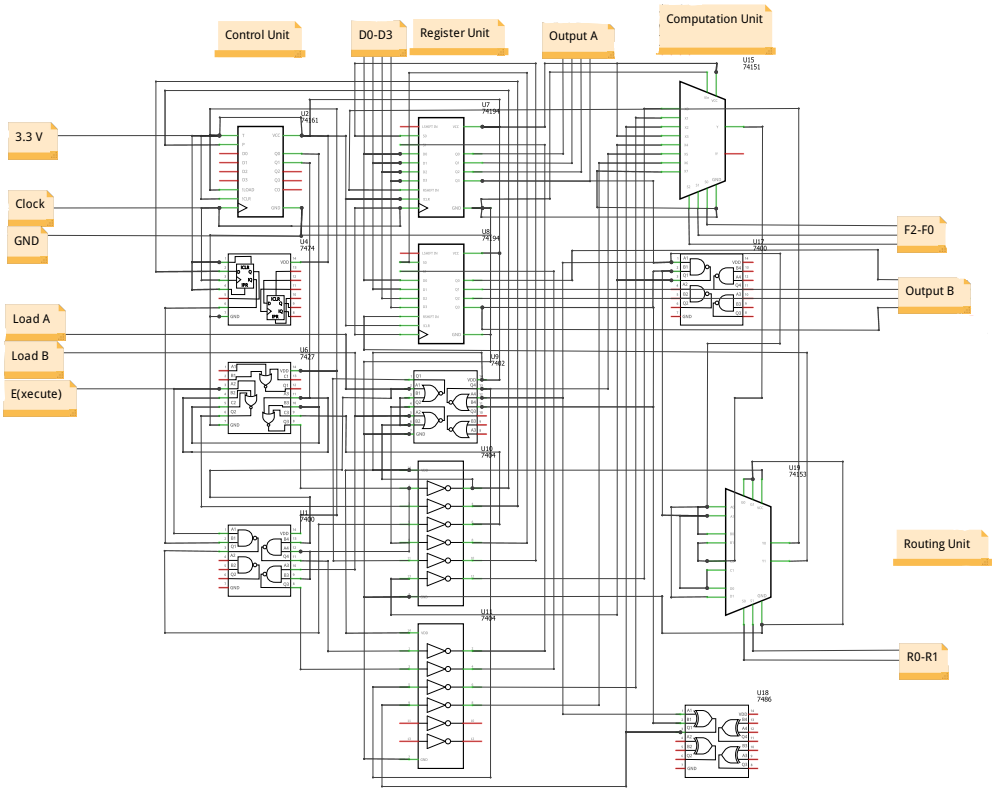
From the truth tables above, we know that if and only if Load A / Load B is held high and Shift bit is low, the value controlled by D3-D0 will be parallel loaded into Register A / Register B, requiring S1 and S0 values both held high.

After the loading completes, after Execute is held high, the computation cycle begins, requiring the binary counter starting to count and shift registers shifting right (according to our design) simultaneously. The processes above require Shift = 1 but not necessarily Execute = 1 since even if Execute bit is held low during a computation cycle, this cycle will continue running until finishing.

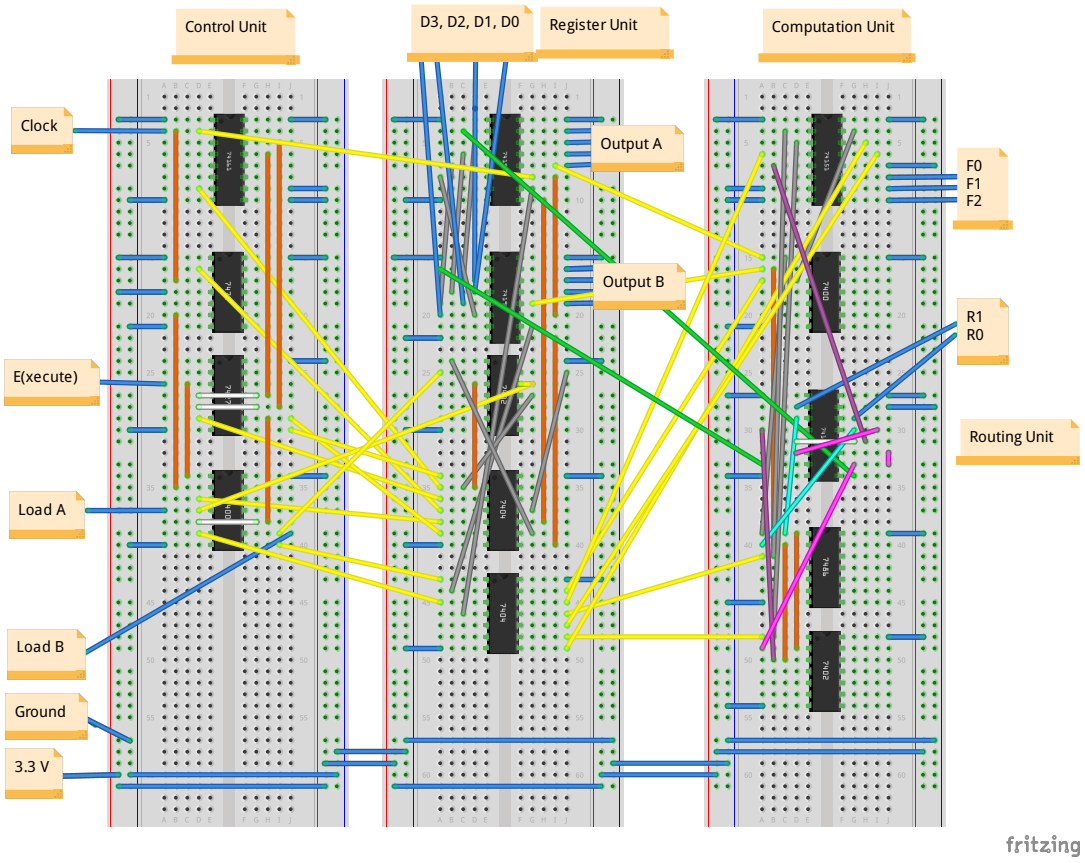
In the meanwhile, if Load A or Load B is held high during the computation cycle, we should finish the computation cycle first before we do another round of parallel

loading. The computation cycle will complete by itself when the binary counter counts to $C_1C_0$ = 00 again since Shift bit would be automatically held low according to $S = C_0 + C_1 + E \cdot Q'$ no matter $E = 1$ or $E = 0$ (during the computation cycle, $E \cdot Q'$ is always held low).

**b. Detailed Circuit Schematic.**




**5) Breadboard view / Layout sheet.**




## 6) 8-bit logic processor on FPGA.

### a. Summary of .sv files

*compute.sv:* This file contains a module of Computation Unit, with 8 bitwise functions implemented. Since extending a 4-bit logic processor to 8-bit only increases the length of the computation cycle but not the functions supported by the computation unit, no changes are done to this module.

*Control.sv:* This file contains a module of Control Unit, with a Moore Machine implemented. Since extending a 4-bit logic processor to 8-bit only increases the length of the computation cycle, the number of states also increases from 4 to 8. We add 4 more states to the Moore Machine to support 8-bit inputs' computations.

*HexDriver.sv:* This file contains a module of visualizing outputs. To be specific, it contains a module of demonstrating output values on LEDs using 7-segment method. An LED digit on the Urbana Board is divided into 7 segments, with different segment combinations we can express numbers from 0 to F in hexadecimal. Since extending a 4-bit logic processor to 8-bit doesn't change the way LEDs displaying output values, no changes are done to this module.

*Processor.sv:* This file contains the top-level module of the logic processor. Since extending a 4-bit logic processor to 8-bit increases the digits of values in Register A and Register B, we change the input values and output values in this top-level module

from 4 to 8. Furthermore, we add 2 more hex drivers to let more digits displayed on LEDs on Urbana Board.

*Reg_4.sv:* This file contains the module of Register Unit. This module can reset, hold, shift, or parallel load the value in a module instantiation according to the values of its inputs. Since extending a 4-bit logic processor to 8-bit increases the digits of values in Register A and Register B, we change the number of digits stored in module from 4 to 8.

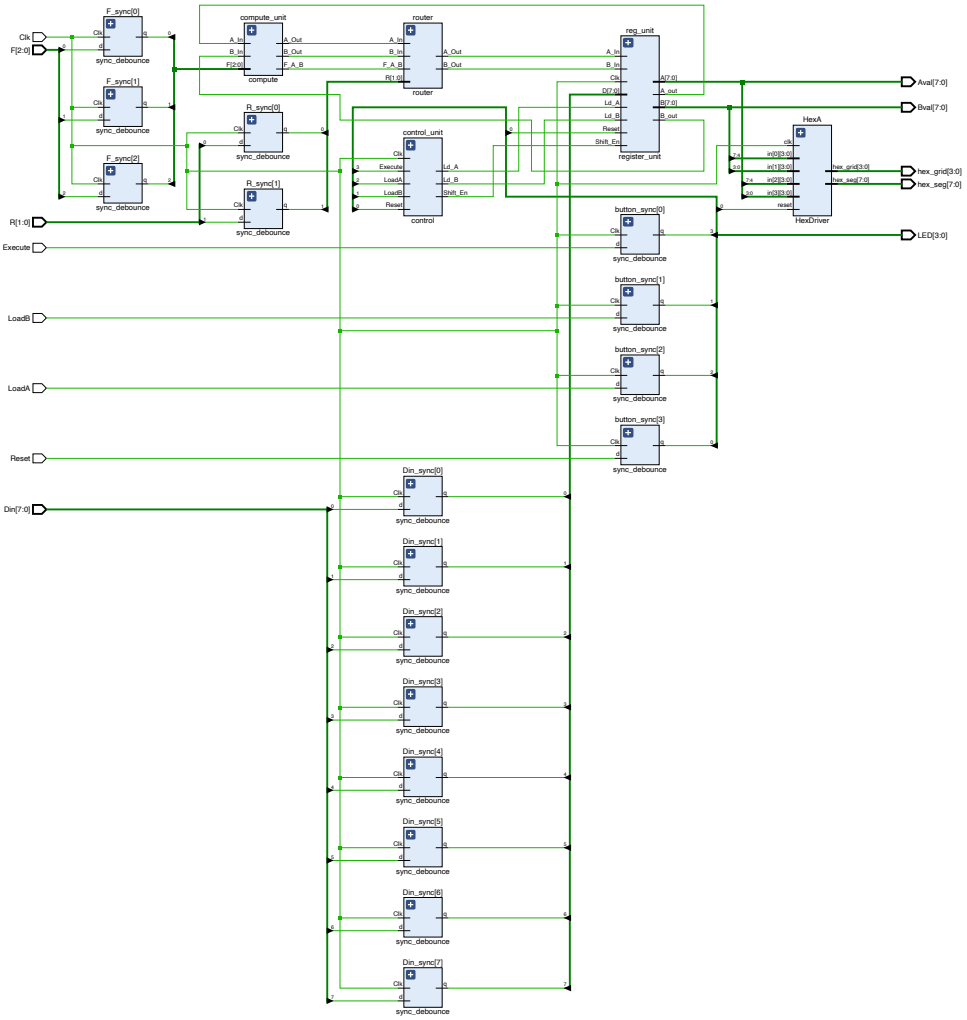
*Register_unit.sv:* This file contains the module of Register Unit. This module can reset, hold, shift, or parallel load the value in a module instantiation according to the values of its inputs. Since extending a 4-bit logic processor to 8-bit increases the digits of values in Register A and Register B, we change the number of digits stored in module from 4 to 8.

*Router.sv:* This file contains the module of Routing Unit. This module sends new A and new B to Register Unit according to the values of its inputs. Since extending a 4-bit logic processor to 8-bit only increases the length of the computation cycle but not the ways of routing supported by the routing unit, no changes are done to this module.

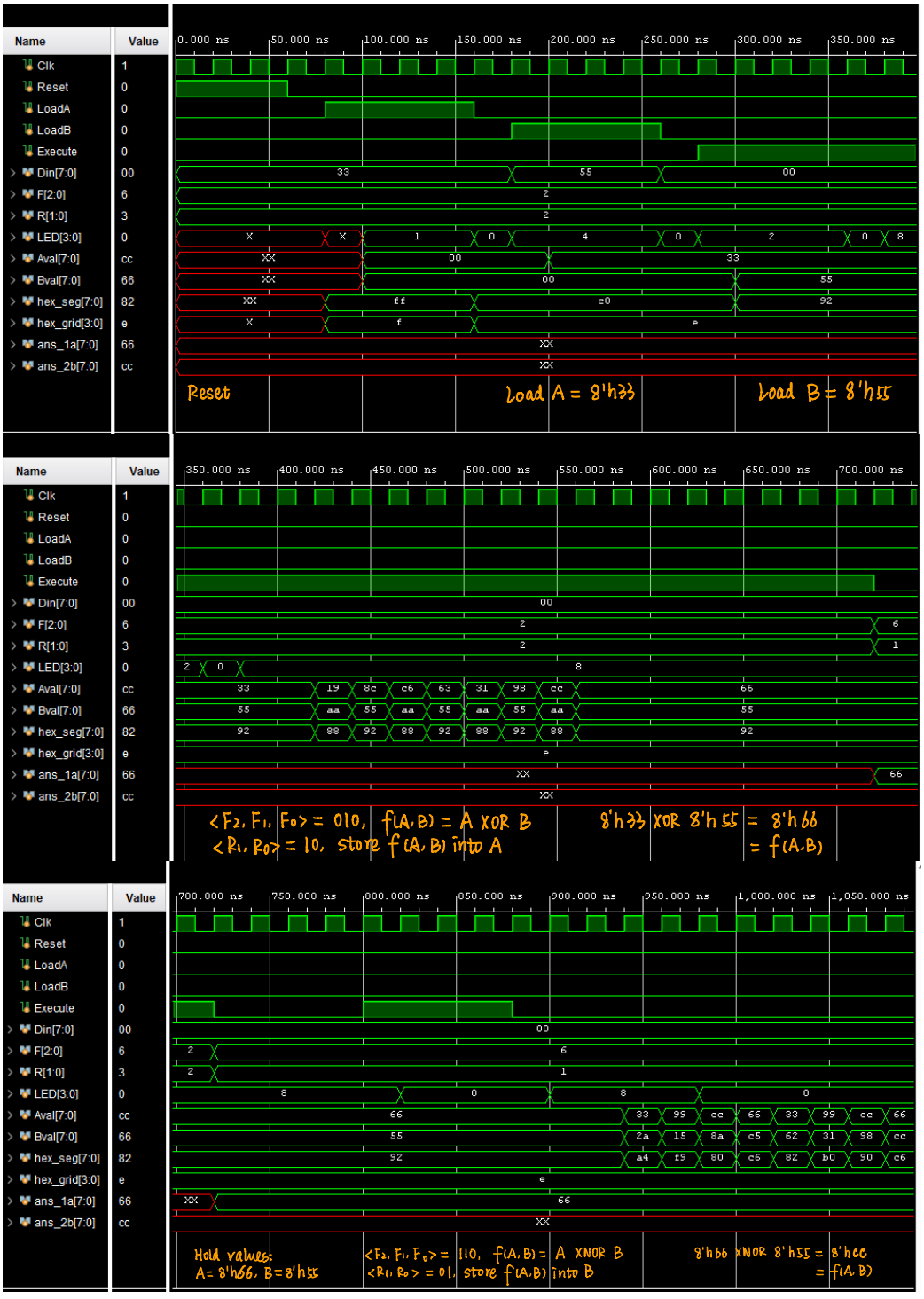
*Synchronizers.sv:* This file contains a module implementing a denouncer circuit and synchronizer. It serves as a synchronizer for pushbutton and switch inputs, addressing potential instability by detecting stable input cycles using a counter. The choice of counter width is conditional, using a larger width for synthesis to optimize performance. Since this file has nothing to do with the number of digits supported by the logic processor, no changes are done to this module.

*testbench_8.sv:* This file contains 3 test cases of this 8-bit logic processor, namely Reg A XOR Reg B storing the result in Reg A, Reg A XNOR Reg B storing the result in Reg B, and swapping Reg A and Reg B. Since extending a 4-bit logic processor to 8-bit only increases the length of the computation cycle but not the ways of routing supported by the routing unit nor the functions supported by the computation unit, no changes are done to this module.

## b. RTL Block Diagram



## c. A Simulation of the Processor




## d. Vivado Debug Core Generation Trace

Step 1: Open synthesized design and set up a debug core using Aval, Bval, and LED as “data and triggers”. After setting up a debug core, re-run synthesis and implementation, generate a new bitstream, open hardware design, auto connect the device, and then program the device using both the bitstream and the debug core.

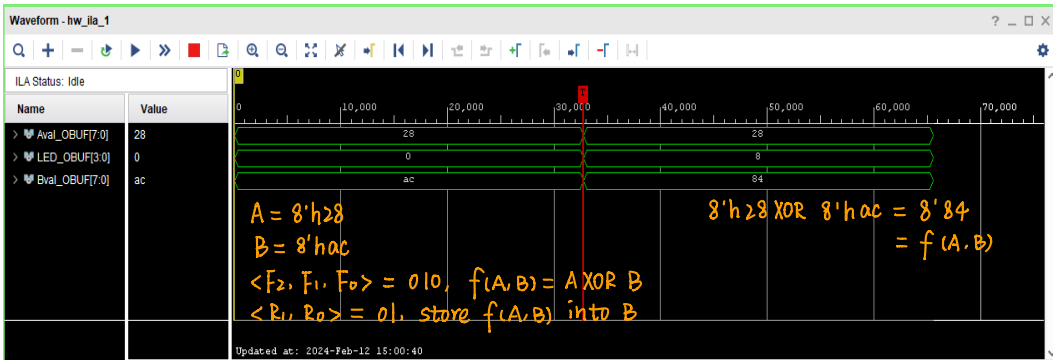
Step 2: Set LED as trigger and set the trigger condition as “RXXX” in bits.

Step 3: Load values to Reg A and Reg B respectively using Load A and Load B buttons on Urbana Board. Set <F2, F1, F0> and <R1, R0> switches on Urbana Board.

Step 4: Click “Run” on the menu of the debug core waveform interface, and then press Execute button on Urbana Board.

Step 5: A waveform of the contents in Reg A and Reg B before and after the computation would appear.

## e. The Output of the Debug Core




## 7) Description of bugs and corrective measures taken.

*Main Bug: The binary counter continued to count after 4 cycles.*

Measures: We didn't connect "S = 0" to terminating the counter, thus the counter receives no signals of stopping itself after 4 cycles. We corrected this bug by connecting Shift bit to one of the ports of the counter chip to determine whether the counter should stop or not.

## 8) Conclusion.

### a. Summary of this lab

This lab is about the design of a logic processor operates on two 4-bit binary numbers, performing 8 different functions and offering versatile result routing options. Data loading involves flipping Load A or Load B switches while maintaining Execute switch low. For computation initiation and routing, Load A and Load B are turned off, followed by flipping switches for function selection (F2-F0) and routing (R1-R0), and turning on Execute. The processor comprises a Register Unit with 4-bit shift registers (A and B), a Computation Unit for logic operations, a Routing Unit for result routing, and a Control Unit overseeing the entire process. This modular design ensures efficient data manipulation and computation cycles.

### b. Answers 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.*

The simplest circuit that can optionally invert a signal is an XOR gate with 2 inputs and 1 output. One of the inputs is the signal, and the other input serves as a chip select. This circuit can be used to simplify the construction of this lab. For example, it can simplify 8-to-1 MUX into a 4-to-1 MUX with an extra XOR gate.

We use <F1, F0> as the select bits of the 4-to-1 MUX and we also keep the functions defined by <F1, F0> = 00, 01, 10, 11 as AND, OR, XOR and 1. F2 is the chip select input of the XOR gate.

If F2=0 (we want the result of AND/OR/XOR/1), and the result of 4-to-1 MUX is 1, then the output of the XOR gate will be 1, corresponding to the operation result of AND/OR/XOR/1.
If F2=0 (we want the result of AND/OR/XOR/1), and the result of 4-to-1 MUX is 0, then the output of the XOR gate will be 0, corresponding to the operation result of AND/OR/XOR/1.
If F2=1 (we want the result of NAND/NOR/XNOR/0), and the result of 4-to-1 MUX is 1, then the output of the XOR gate will be 0, corresponding to the operation result of NAND/NOR/XNOR/0.
If F2=1 (we want the result of NAND/NOR/XNOR/0), and the result of 4-to-1 MUX is 0, then the output of the XOR gate will be 1, corresponding to the operation result of NAND/NOR/XNOR/0.
Therefore, it can simplify 8-to-1 MUX into a 4-to-1 MUX with an extra XOR gate.

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

Different parts in a modular design are always relatively independent, with a few information or values interchanging. For example, it is easier to verify the correctness of computation unit and routing unit without completing the function of control unit in a modular design. Therefore, modular designs enhances testability of different portions of the design, and they help us focus more on the most complicated and difficult parts in the design, thus cutting down development time.

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

Mealy Machines' output depends on both states and inputs, while Moore Machines' output only depends on the state. We choose to implement the logic processor using a Mealy Machine because it has only 2 states, thus we can use fewer flip-flops to store intermediate values than Moore Machine. The tradeoff lies in that when we use fewer states and flip-flops, the logical expressions of intermediate values and outputs are more complicated than those of Moore Machine, thus increasing the number of logic gates used in our 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 runs the simulation of our logic processor using testbench file we provide, while Vivado Debug Cores use the information and values (buttons or switches) we provide by the Urbana Board. In testing and modifying stage of developing a processor or another project, using vSim helps us find bugs and is convenient for us

to promote our design using small test cases. However, Vivado Debug Cores are more efficient when we want to run many test cases to verify the correctness of our design.