

# ECE 385 Experiment 2: Lab Report

---

**Spring 2023**

**Eshaan Tibrewala (eshaant2) and Shivam Patel (shivamp6)**

**Lab Section: SL Friday 4:00PM**

**TA: Shitao Liu**

## ECE 385 Experiment 2: Lab Report

Introduction

Operating the Logic Processor

Loading the Registers and Initializing Function and Routing

Table 1: Switch Assignment for Upper Switch Board

Table 2: Switch Assignment for Lower Switch Board

Initiating Computation and Routing

Logic Processor Components

Finite State Machine

Control Unit

Register Unit

Computation Unit

Table 3: Computation Unit Function Table

Routing Unit

Table 5: Routing Selection Table

Design Process of Logic Processor

Design Considerations

Design Steps

Table 6: Control Unit State Transitions

Breadboards and Circuit Schematics for Logic Processor

Logic Processor on FPGA

Summary of .SV Files

Table 7: .SV Modules for Logic Processor

Processor.sv

Code Block 1: module Processor

Control.sv

Code Block 2: module control

Compute.sv

- Code Block 3: module compute
- Reg\_8.sv
  - Code Block 4: module reg\_8
- Register\_unit.sv
  - Code Block 5: module register\_unit
- Router.sv
  - Code block 6: module router
- Synchronizers.sv
  - Code block 7: module sync, sync\_r0, sync\_r1
- Testbench\_8.sv
  - Code block 8: module testbench
- HexDriver.sv
  - Code block 9: module HexDriver
- RTL Top Level Block Diagram
- Processor Simulation
- SignalTap ILA Trace
- Troubleshooting Bugs and Glitches
- Lab Questions
  - Inverting a Signal
  - Modular Design
  - Moore vs. Mealy Finite State Machines
  - ModelSim vs. SignalTap
- Conclusion

## Introduction

---

Experiment 2 was a two week experiment, and consisted of designing and implementing a bit-serial logic processor and getting acclimated to Quartus, SystemVerilog and generating simulations. The first week was spent on the logic processor, which is capable of calculating 8 different bit-wise operations consisting of **A AND B**, **A OR B**, **A XOR B**, **1111(all high)**, **A NAND B**, **A NOR B**, **A XNOR B**, and **0000(all low)**. **A** and **B** are the two 4-bit data on which the operations are performed. The processor also has the capability of routing the operations four different ways, where **A** and **B** are routed back to their respective registers without being operated on, **A** is routed back to its original register while the result of the operation is routed to Register A, **B** is routed back to its original register and Register A receives the result of the operation, or **A** and **B** are simply routed to the opposite register without being operated on. The registers used were bidirectional shift registers, so that the operations can be conducted one bit at a time and the result can be shifted back into the correct register one bit at a time. The design of the

processor also utilized multiple multiplexers, D Flip Flops, and a counter. The second part of the lab was more of a simple introduction to SystemVerilog and Quartus, where we created a basic Quartus Prime project in order to synthesize, simulate, and extend our 4-bit logic processor to 8-bits.

## Operating the Logic Processor

---

In order to operate the logic processor, the user must use a series of switches which allow them to load Registers A and B, choose the desired operation to carry out, and finally determine the routing destination. The logic processor includes LEDs so that the user can keep track of their set up and follow the steps of the processor as it carries out the desired operation on the clock cycle. Bits D3-D0 are the initial parallel load inputs into Registers A and B, bits F2-F0 select the specific operation being performed, bits R1 and R0 determine the routing option, Load A loads data into Register A, Load B loads data into Register B and Execute starts the logic processor cycle.

### Loading the Registers and Initializing Function and Routing

Setting up the logic processor requires two 8 - DIP switches (Manufacturer PN: 76SB08T) and the DE10-Lite FPGA board, which our processor uses as a clock to slowly step through the process. While the clock could have also been simulated with the SPDT switches (Manufacturer PN: 76SC04T), we decided to go with the FPGA because it was already debounced, eliminating the chances of static hazards that come with using mechanical switches. We placed our two DIP switches in a line, such that the upper switch board contains Load A, Load B, Execute, F2-F0, and R1-R0 and the lower switch board contains D3-D0 for Register A and Register B. While the same four switches can be utilized to load both registers, we used all eight available switches to organize and simplify the set up process. We used Key 0 on the FPGA board because it functioned as a debounced switch, and could therefore be used to step through the clock.

**Table 1: Switch Assignment for Upper Switch Board**

<b>SWITCH NUMBER</b>	<b>SWITCH FUNCTION</b>
8	Load A
7	Load B
6	Execute

SWITCH NUMBER	SWITCH FUNCTION
5	F2
4	F1
3	F0
2	R1
1	R0

**Table 2: Switch Assignment for Lower Switch Board**

SWITCH NUMBER	SWITCH FUNCTION	CORRESPONDING BIT
8	D3 (Register A)	$a_3$
7	D2 (Register A)	$a_2$
6	D1 (Register A)	$a_1$
5	D0 (Register A)	$a_0$
4	D3 (Register B)	$b_3$
3	D2 (Register B)	$b_2$
2	D1 (Register B)	$b_1$
1	D0 (Register B)	$b_0$

In order to set up the logic processor, the user must first determine which bits to load into Registers A and B via the lower switch board. Switches 5-8 load in Register A, with switch 8 representing the most significant bit. Turning the switch **ON** makes the bit binary 1 while keeping the switch **OFF** makes the corresponding bit binary 0. Switches 1-4 function the same way, but load in Register B with switch 4 representing the most significant bit.

Once the lower switch board is set, the user then turns on switch 7 and switch 8 on the upper switch board so that Load A and Load B are **ON**. However the bits A and B are not loaded into their corresponding registers until the next rising edge, so the user must press Key 0 on the FPGA board to simulate one clock period. The user can then check if the load was successful by checking LED 1, which is placed right below the lower switch board on the breadboard. A red light on the LED represents logic 1 and no light represents logic 0. The bars on the LED are numbered 1-10 from top to bottom. LED bars 1-4 represent Register A with bit  $a_3$  being represented by bar 1. LED bars 7-10 represent Register B with

$b_3$  being represented by bar 7. LED bars 5 and 6 always remain **OFF** for organizational purposes and to make it easier to differentiate between Registers A and B.

## Initiating Computation and Routing

To start the logic processor, turn Load A and Load B back **OFF** and turn **ON** switch 3 on the upper switch board, which turns Execute **ON**.

Once Execute is **ON**, the logic processor will perform exactly only cycle, in which the bits will shift out of the register unit and into the computational unit, after which the new (or same depending on which function is chosen) bits will be routed back to the correct register as determined by bits R1-R0 in the routing unit. This cycle occurs once, and is not restarted until the Execute switch is turned **OFF** for a clock period, allowing the FSM to return to the initial state, at which point the logic processor can begin again as normal. It is important to note that the logic processor will still function as normal and complete the cycle even is the Execute switch is turned **OFF** in the middle of the computational cycle.

## Logic Processor Components

---

The logic processor is comprised of the control unit, register unit, computation unit, and routing unit. Each one is vital to the successful functionality of the processor, and combines together to form the high-level block diagram shown in Figure 1. As shown below, the control unit simply initiates the processor, and once executed, the rest of the processor functions between the register, computation, and routing units, until the clock cycle has been reset.

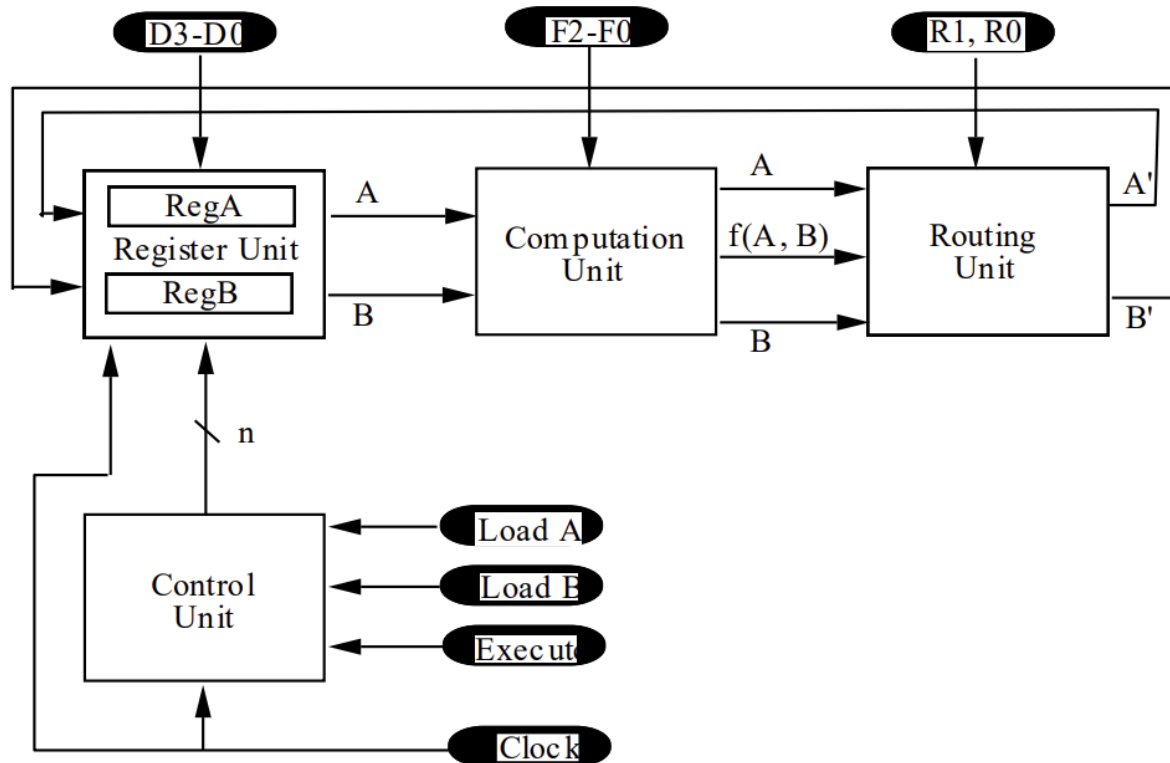


Figure 1: High-Level Block Diagram of Logic Processor

## Finite State Machine

The entire functionality and design of the control unit, and therefore the rest of the logic processor, depends on the finite state machine. The FSM is responsible ensuring that logic processor performs exactly one operation on the 4-bit data, and that the clock cycle does not repeat itself until Execute is turned back off for at least one clock pulse, before being turned on again to perform another operation. To build a functioning FSM, we first had to decide between a Moore machine or a Mealy machine. While both are viable options, we implemented the Mealy machine because it depends on both the current state and current inputs of the FSM, allowing it to function with fewer states than a Moore machine.

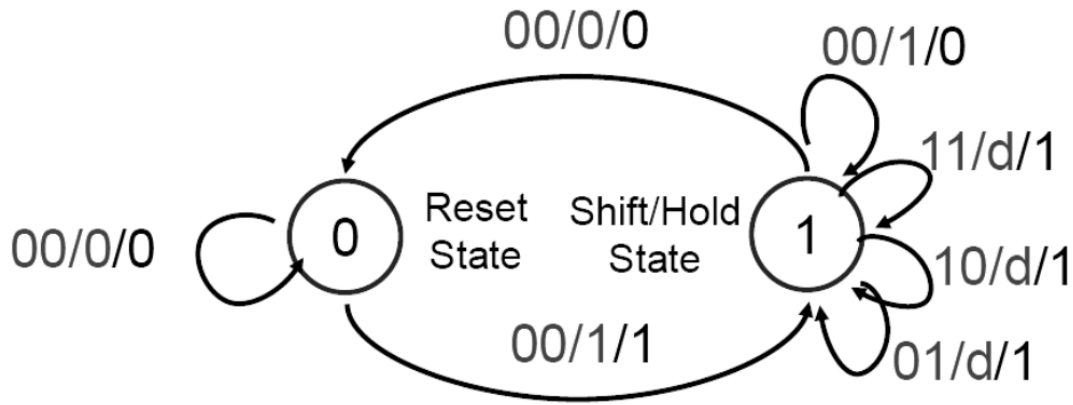


Figure 2: Mealy State Machine for Control Unit

Figure 2 shows the Mealy machine we used to drive the control unit and control the register unit operation. We used the FSM design given in the lab document since it was the simplest implementation of the Mealy machine and we knew we would be able to successfully implement it in our control unit. The Mealy machine inputs are 1-bit Execute signal  $E$ , 1-bit state representation  $Q$ , and a 2-bit count  $C1C0$ . Execute starts the computational cycle,  $Q$  represents the two states of the Mealy machine, which are the reset state and the shift/halt state. The count bits represent the current number of shifts, and counts up to 3 before returning to 0. The counter is held in the shift/halt state, and helps to reduce the total number of states required to implement the FSM. The outputs of the Mealy machine are the register shift bit  $S$ , the next state bit  $Q^+$ , and the 2-bit next count  $C1^+C0^+$ .

The transition arcs are labeled  $C1C0/E/S$ , where  $C1C0$  is the current count input,  $E$  is the execute input, and  $S$  is the register shift output. The FSM starts in the reset state, where  $EQC1C0 = 0000$ . The FSM will remain in this state until Execute is turned **ON**, starting the cycle. While Execute is low, the register unit and the next state also remains the same, where  $SQ^+C1^+C0^+ = 0000$ . The next state does not change until the next clock edge after Execute is flipped high, where  $EQC1C0 = 1000$ . The resulting next state is then  $SQ^+C1^+C0^+ = 1101$ , where  $S = 1$  because shift is enabled,  $Q^+ = 1$  because the next state is the shift/halt state, and  $C1^+C0^+ = 01$  since the count has been enabled so the FSM begins counting to decimal 3. Once four shifts are completed, the FSM remains in the shift/halt state, where  $SQ^+C1^+C0^+ = 0100$  until Execute is turned back off for one clock edge, at which point it returns to the reset state and  $SQ^+C1^+C0^+ = 0000$ . This represents one full complete cycle of the logic processor, and waits in the reset state to perform another cycle when Execute is switched back **ON**.

## Control Unit

The control unit houses the Mealy machine described in the *Finite State Machine* section of the report and the Load A and B signals. When Load A or Load B is switched high, the control unit tells the register unit to accept the values of bits  $D3D2D1D0$  to the respective register. The control unit also relies on the clock as an input to drive the FSM and the counter chip. The only output of the control unit is the shift enable bit  $S$ , which tells the register unit whether or not to shift the data. In one complete cycle, this bit is high for four shifts, during which it shifts the registers four times, before returning to low until the next FSM cycle begins.

## Register Unit

The register unit is made up of two 4-bit bidirectional shift registers, Register A and Register B.  $D3 - D0$  are the shift register inputs, and parallel load the data into the registers before a computational cycle is initiated by the Execute signal. However, the registers are only loaded into their respective registers when Load A or Load B are switched on, before the start of the cycle. During a computational cycle, the only inputs into the register unit come from the routing unit, which output two different 1-bit signals  $A'$  and  $B'$  to be shifted into their corresponding registers. The register unit also has two 1-bit outputs, which are  $A$  and  $B$ . These 1-bit signals are the least significant bits of the data stored in the register, and serve as inputs for the computation unit where the bitwise operations are performed. In a given cycle, the register unit is shifted four times until each register holds the desired result of the logic processor computation.

## Computation Unit

The computation unit is where the bitwise operations are performed. Bits  $F2 - F0$  are the inputs into a 4-1 mux which determine the operation to be performed. We were able to use one 4-1 MUX to perform eight operations by using bits  $F1 - F0$  to drive to multiplexer, and then driving the result through a XNOR gate with the  $F2$  bit. This simplified implementation was possible since the last four operations are just the inverse of the first four operations. A complete list of the function select inputs and its corresponding output are shown Table 3. Our computation unit works such that the register unit shifts out two 1-bit signals A and B into the computation unit. Based on the select bits  $F1 - F0$ , the corresponding operation is performed. The resulting bit is then driven through an XNOR, where it is inverted if  $F2$  is low, or remains unchanged if  $F2$  is high. The resulting bit, along with the original signals A and B, are then sent to the routing unit so that they can be shifted back into the registers. Since we only had NAND,



XOR, and inverter chips, we determined it would be easier to compute the inverted functions (NAND, NOR, XNOR, 0000).

$$(1) A \text{ NAND } B \rightarrow \overline{(A \bullet B)}$$

$$(2) A \text{ NOR } B \rightarrow \overline{(\overline{A} \bullet \overline{B})}$$

$$(3) A \text{ XNOR } B \rightarrow \overline{(A \oplus B)}$$

**Table 3: Computation Unit Function Table**

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

Equations 1-3 show how we implemented the inverted logic operations using NAND, XOR, and inverter. For NAND we just used the chip normally. For NOR, we inverted signal A and signal B, drove those signals through a NAND gate, and then inverted that signal one last time using the inverter chip. XNOR was implemented by driving the signals through a XOR gate, and then inverting the output. 0000 was implemented by connecting directly to ground. All of these signals were then XNOR with the *F2* bit using the inverter and XOR gate like we did for the original XNOR operation.

## Routing Unit

The routing unit delivers the outputs of the computation unit back to the register unit. The inputs of the routing unit are bits *R1* and *R0*. We implemented it by using two 4-1 MUX's. One multiplexer was used to drive the new *A'* signal to Register A and the other multiplexer was used to drive the new *B'* signal to Register B. We needed two separate

multiplexers because there are four possible routing options for each register. The possible routing options are shown in Table 5.

**Table 5: Routing Selection Table**

R1	R0	A*	B*
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

In Table 5,  $F$  represents the result of the operation performed in the computation unit, and  $A$  and  $B$  represent the original signals that were shifted out of the register unit.

## Design Process of Logic Processor

---

### Design Considerations

For most of the circuit we decided on one approach at the beginning of the design process and followed through with it to the end. However for the control unit, we were originally undecided on whether to use a 4-bit counter chip (Manufacturer PN: SN74HC161N) or two D flip-flops. We considered a counter chip because it only required one additional chip, but we had never used the chip before so it would take some time to learn and test the pin assignments so that the chip could function properly. The D flip-flop route would also only require one chip (Manufacturer PN: SN74HC74N) since it contains two separate flip-flops. We initially decided to go with the flip-flops since we also used the same chip for the  $Q^+$  signal of the FSM, but switched to the counter chip half way through building the circuit because we were unable to get the flip-flops to work. Once we learned the pin assignments of the chip the counter was significantly easier to use since the only inputs we had to factor in were the count enable and count inhibit signals. The design and implementation process for the control unit took up the most time because we made separate designs for both possibilities, and then implemented those designs multiple times on the actual breadboard until we finally got the counter chip working properly.

We also mainly used NAND gates and invertors to implement our FSM functions because NAND is a universal operation and can be manipulated to perform functions such as AND or OR. While we could have also used NOR gates, we decided that solely using NAND would make our implementation more uniform, which would simplify the debugging process. However as mentioned in the Logic Processor Components section, we also used XOR chips for the computation unit specifically because it required XOR and XNOR operations, and the simplest and most efficient way to implement those would have been the XOR chip.

## Design Steps

We started our design process by creating the control unit. Table 6 shows the truth table we used to derive two K-MAPs and SOP equations for  $S$ , the register shift bit, and  $Q^+$ , the next state bit. Table 6 is derived from the Mealy machine we used to drive the control unit. Since we ultimately decided to use a counter chip, we did not need to create SOP logic for  $C1^+$  and  $C0^+$  since the chip does the counting for us. However the current count bits  $C1$  and  $C0$  are still included in our expressions because they are important for determining both the select bit and next state as per our FSM.

**Table 6: Control Unit State Transitions**

$E$	$Q$	$C1$	$C0$	$S$	$Q^+$	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	x	x	x	x
0	0	1	0	x	x	x	x
0	0	1	1	x	x	x	x
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	x	x	x	x
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	0	1	0	0

$E$	$Q$	$C1$	$C0$	$S$	$Q^+$	$C1^+$	$C0^+$
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0

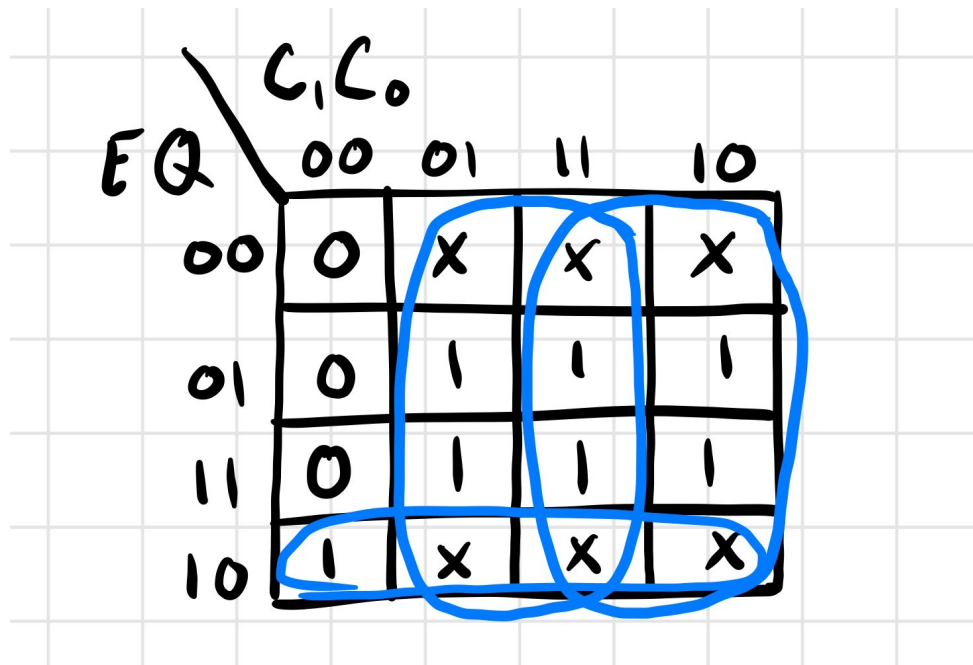


Figure 3: K-MAP for Register Shift Bit  $S$

$$S = (E \wedge \overline{Q}) \vee C0 \vee C1$$

The main output of the entire control unit is the register shift bit  $S$ . This is the signal that goes directly to the register unit, and tells Registers A and B whether or not to shift in the bits. As shown by the K-MAP in Figure 3,  $S$  depends on the Execute signal and the count, so that the register can properly shift four bits during one cycle. To implement the  $S$  signal, we utilized the NAND chips we had along with the D flip-flop chip.

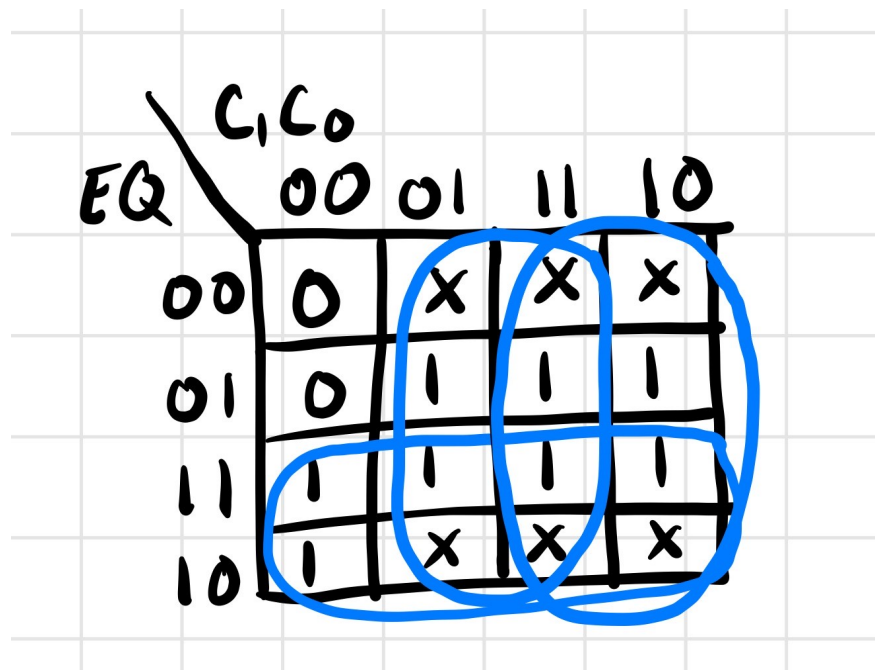


Figure 4: K-MAP for Next State Bit  $Q^+$

$$Q^+ = E \vee C0 \vee C1$$

The  $Q^+$  bit is vital for the internal operations of the control unit. Since our FSM consists of two states,  $Q^+$  tells the FSM and the user which state the FSM is currently in, and ensures that it remains in the shift state for exactly four register shifts before returning to the reset/ready state. Like the  $S$  bit implementation, we also used NAND gates to implement this signal.

The schematics for this signals can be seen in Figure 5, which also shows how the rest of our circuit was designed on a physical level along with the gate level implementations. Our next step in the design process was to design the control and routing unit. Since the rest of the processor was able to be designed with basic logic chips and multiplexers, the design steps for the units was fairly simple and only required us to figure out the pin assignments on the chips.

## Breadboards and Circuit Schematics for Logic Processor

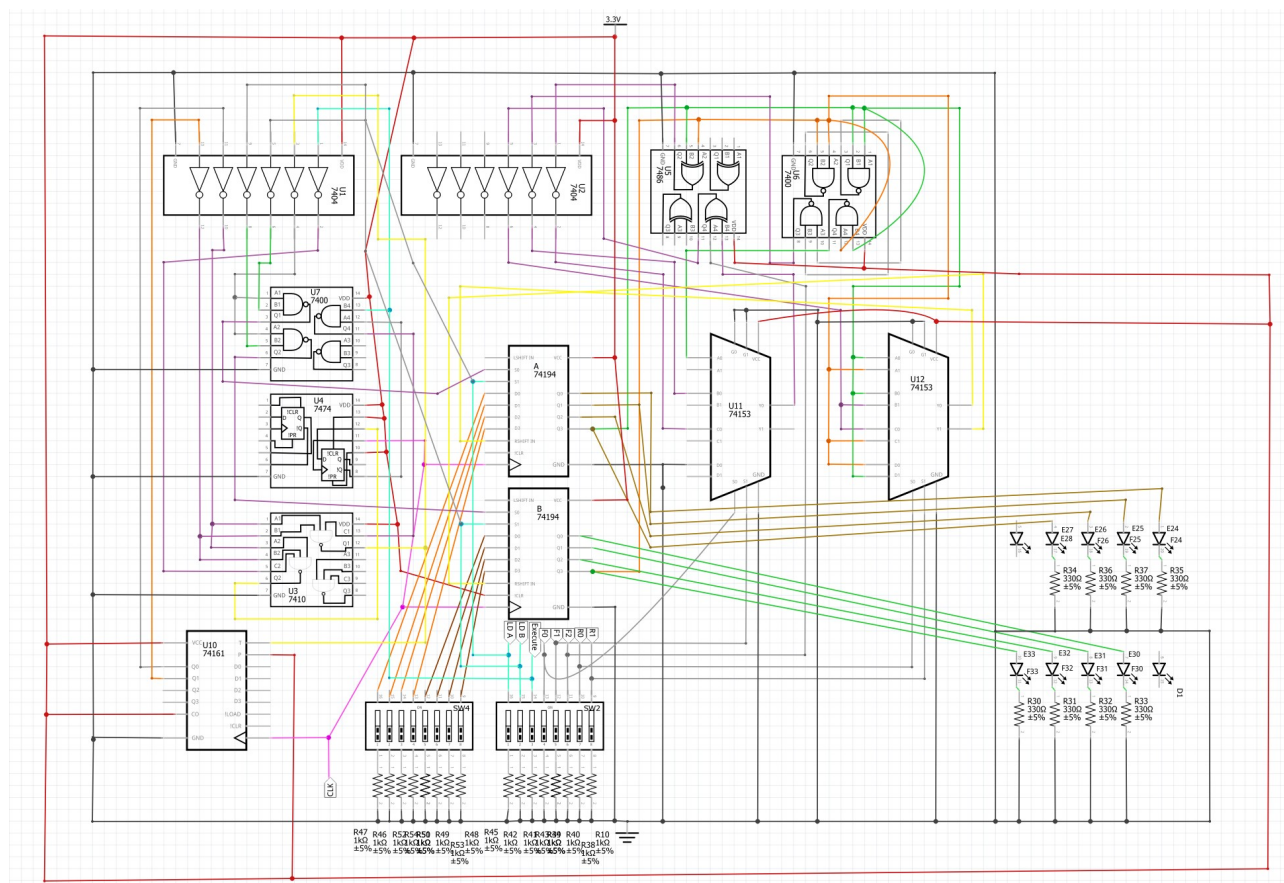


Figure 5: Logic Processor Schematic

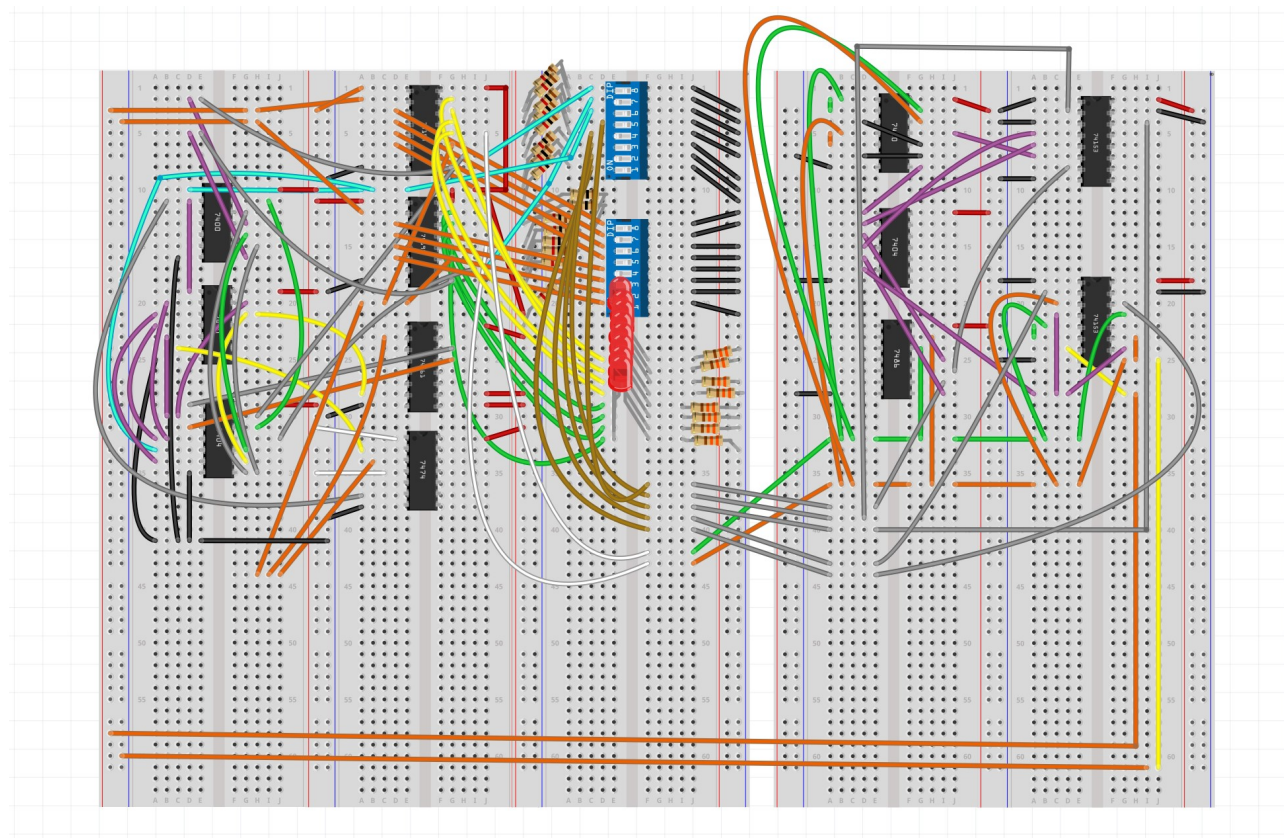




Figure 6: Logic Processor Breadboard Design

Our breadboard and gate level circuit schematic took a clock input at PIN G44 on the left most breadboard so that both registers, counter, and the flip-flop could operate on the same clock signal. For debugging we used the debounced button on the FPGA but the clock signal can also be generated from ADALM using square waves.

## Logic Processor on FPGA

### Summary of .SV Files

In order to extend the logic processor from 4 bits to 8 bits, we utilized ten different modules to drive and output the logic processor on the FPGA. Table 7 shows the individual modules, and its overall functionality in the logic processor. All of these modules were given to us to implement the 4 bit processor, but we only had to change some of them to extending it to 8-bits.

**Table 7: .SV Modules for Logic Processor**

MODULE	UNIT/PURPOSE
Processor.sv	Top Level Module
HexDriver.sv	LED output on FPGA
Control.sv	Control unit
Compute.sv	Computation unit
Reg_8.sv	Register to store the bits
Register_unit.sv	Register unit
Router.sv	Routing unit
Synchronizers.sv	Bringing async signals to FPGA
testbench_8.sv	Test the logic processor

## Processor.sv

### Code Block 1: module Processor

```
1 module Processor (input logic   clk,      // Internal
2                   Reset,      // Push button 0
3                   LoadA,      // Push button 1
4                   LoadB,      // Push button 2
5                   Execute,    // Push
6   input  logic [7:0] Din, //input data (CHANGED 3:0 TO 7:0 FOR 8
   BITS)
7   input  logic [2:0] F, //hardcoded for testbench, otherwise
   input from switches
8   input  logic [1:0] R, //hardcoded for testbench, otherwise
   input from switches
9   output logic [3:0] LED, // DEBUG
10  output logic [7:0] Aval,Bval // DEBUG (CHANGED TO 7:0)
11  output logic [6:0] AhexL,AhexU, BhexL,BhexU);
```

Code block 1 shows the Processor module, and its inputs and outputs. Processor serves as the top level module, and drives the whole logic processor by using its inputs to call all the other modules, and then outputting the result to the FPGA using HexDriver. Processor was already written for us to implement 4-bits, so to extend it to 8-bits, all we had to do was change `input logic Din` from 4-bits to 8-bits since it represents the actual data being calculated. Since the rest of the module inputs don't depend on the size of the input signals, we left them unchanged so that the logic processor could function properly. The Processor module also included inputs such as `[2:0] F`, `[1:0] R` and outputs such as `[3:0] LED`, `[7:0] Aval`, `Bval` for debugging purposes and hardcoding the testbench. We left those in the report and final module to increase understanding of the overall functionality.

## Control.sv



## Code Block 2: module control

```
1 module control (input logic clk, //internal
2                 input logic Reset, LoadA, LoadB, Execute, //from
   push buttons/switches
3                 output logic Shift_En, Ld_A, Ld_B ); //outputs to
   Register unit
```

Code block 2 shows the Control module, and its inputs and outputs. This serves as the control unit. While the inputs and outputs do not depend on the number of bits, the actual functionality does as the unit houses the FSM. Therefore we did not change the inputs and outputs, but we did change the number of states to accurately extend the processor from 4 bits to 8 bits. For the 4-bit implementation, we had `enum logic [2:0]` which allowed for up to eight states. However only six were needed, since four states were for the register, one bit was for reset, and one more bit was for halt. We changed it to `enum logic [3:0]` so that we could add four more states, `W`, `X`, `Y`, `Z`, for the four additional bits we were extending to. The reset and halt bit remained unchanged. We then modified the functionality of our `always_comb` block so that we could assign outputs based on the correct state. The overall outputs of the module were then driven to the register unit where they either load the registers or shift them depending on whether or not an operation is being performed.

## Compute.sv

### Code Block 3: module compute

```
1 module compute (input logic [2:0] F, //determines which
   operation to be performed
2                 input logic A_In, B_In, //bits from registers A
   and B
3                 output logic A_Out, B_Out, F_A_B); //original
   bits and computed bit
```

Code block 3 shows the Compute module declaration and its inputs and outputs. This serves as the computation block, and is essentially a 1-bit ALU. The operation is selected by the 2-bit F signal, allowing the module to perform eight different operations. We left it unchanged since it functions one bit at a time, so functionality does not depend on how

many bits the logic processor is built to support.

## Reg\_8.sv

### Code Block 4: module reg\_8

```
1 module reg_8 (input logic Clk, //internal
2               input logic Reset, Shift_In, Load, Shift_En,
3               //inputs from control unit
4               input logic [7:0] D, //data loaded in when load
5               high
6               output logic Shift_Out, //high when Shift_In and
7               Shift_En
8               output logic [7:0] Data_Out); //value in register
```

Code block 4 shows the register module declaration and its inputs and outputs. This module is part of the register unit, and stores the bits being calculated by the logic processor. For the 4-bit implementation, `input logic D` and `output logic Data_Out` where 4-bits. Since we are extending the processor, we changed it from `[3:0]` to `[7:0]` in order to handle 8-bits. This module essentially takes its inputs and clears the register if the reset signal is high, or loads in user data if Load signal is high. However if shift is enabled, the module concatenates the shifted in data to the left most bits. Then the module outputs the least significant bit to the computation unit where the logic processor operation can be performed.

## Register\_unit.sv

### Code Block 5: module register\_unit

```
1 module register_unit (input logic Clk, Reset, A_In, B_In, Ld_A,
2                       Ld_B, Shift_En,
3                       input logic [7:0] D,
4                       output logic A_out, B_out,
5                       output logic [7:0] A,
6                       output logic [7:0] B);
```

The register unit module simply initializes two instances of the `reg_8` module, one for Register A and one for Register B. The Load signals determine which register is to be loaded, and `Shift_En` determines whether or not to shift bits into their respective registers. To extending the module to 8-bits, we just changed the size of the `D`, `A`, and `B` signals to from `[3:0]` to `[7:0]`. The functionality remains unchanged since we are just calling the other module, and we already made the respective changes to handle 8-bits.

## Router.sv

### Code block 6: module router

```
1 module router (input logic [1:0] R,  
2               input logic A_In, B_In, F_A_B,  
3               output logic A_Out, B_Out);
```

The router unit uses the select bits `[1:0] R` to route the outputs of the computation unit to the chosen registers in the register unit. Since this is essentially implementing two 4-1 MUXs, we did not need to change any of the code to extend it to 8-bits because the module routes one bit at a time, so it programmed the same way for both 4-bit and 8-bit logic processors.

## Synchronizers.sv

### Code block 7: module sync, sync\_r0, sync\_r1

```
1 module sync (  
2   input logic Clk, d,  
3   output logic q  
4 );  
5  
6 module sync_r0 (  
7   input logic Clk, Reset, d,  
8   output logic q  
9 );  
10  
11 module sync_r1 (  
12   input logic Clk, Reset, d,  
13   output logic q  
14 );
```

```

12     input logic Clk, Reset, d,
13     output logic q
14 );

```

The synchronizer modules help to bring asynchronous signals into the FPGA. `module sync` is the default and works with switches and buttons when there is no reset signal. `module sync_r0` is the synchronizer with reset set to 0, and assigns the next state to zero and `module sync_r1` is the synchronizer with reset set to 1. The purpose of this module is to handle asynchronous signals, and since it was already given to us we did not edit any of it to extend the overall functionality of the logic processor to 8-bits.

## Testbench\_8.sv

### Code block 8: module testbench

```

1  module testbench();
2  // These signals are internal because the processor will be
3  // instantiated as a submodule in testbench.
4  logic Clk = 0;
5  logic Reset, LoadA, LoadB, Execute;
6  logic [7:0] Din;
7  logic [2:0] F;
8  logic [1:0] R;
9  logic [3:0] LED;
10 logic [7:0] Aval,
11     Bval;
12 logic [6:0] AhexL,
13     AhexU,
14     BhexL,
15     BhexU;
16
17 // To store expected results
18 logic [7:0] ans_1a, ans_2b;

```

The testbench module is simply to test our system verilog program. This was also given to us for the 4-bit implementation, and all we had to do was extend the `Din` and the `Ava1`, `Bva1`, `ans_1a`, `ans_2b` signals to 8-bits for accurate testing. As stated in Code block 8, these signals are not assigned as inputs and outputs in the actual module declaration because the processor will be instantiated as a submodule in testbench.

## HexDriver.sv

### Code block 9: module HexDriver

```
1 module HexDriver (input logic [3:0] In0,  
2                   output logic [6:0] Out0);
```

The HexDriver module controls the display on the FPGA board. It simply takes the 4-bit input and extends it to 8-bits so that the DE10 board can display the results of the logic processor as hex values to the user. The module stays the same when the logic processor is extended to 8-bits so we did not change anything.

## RTL Top Level Block Diagram

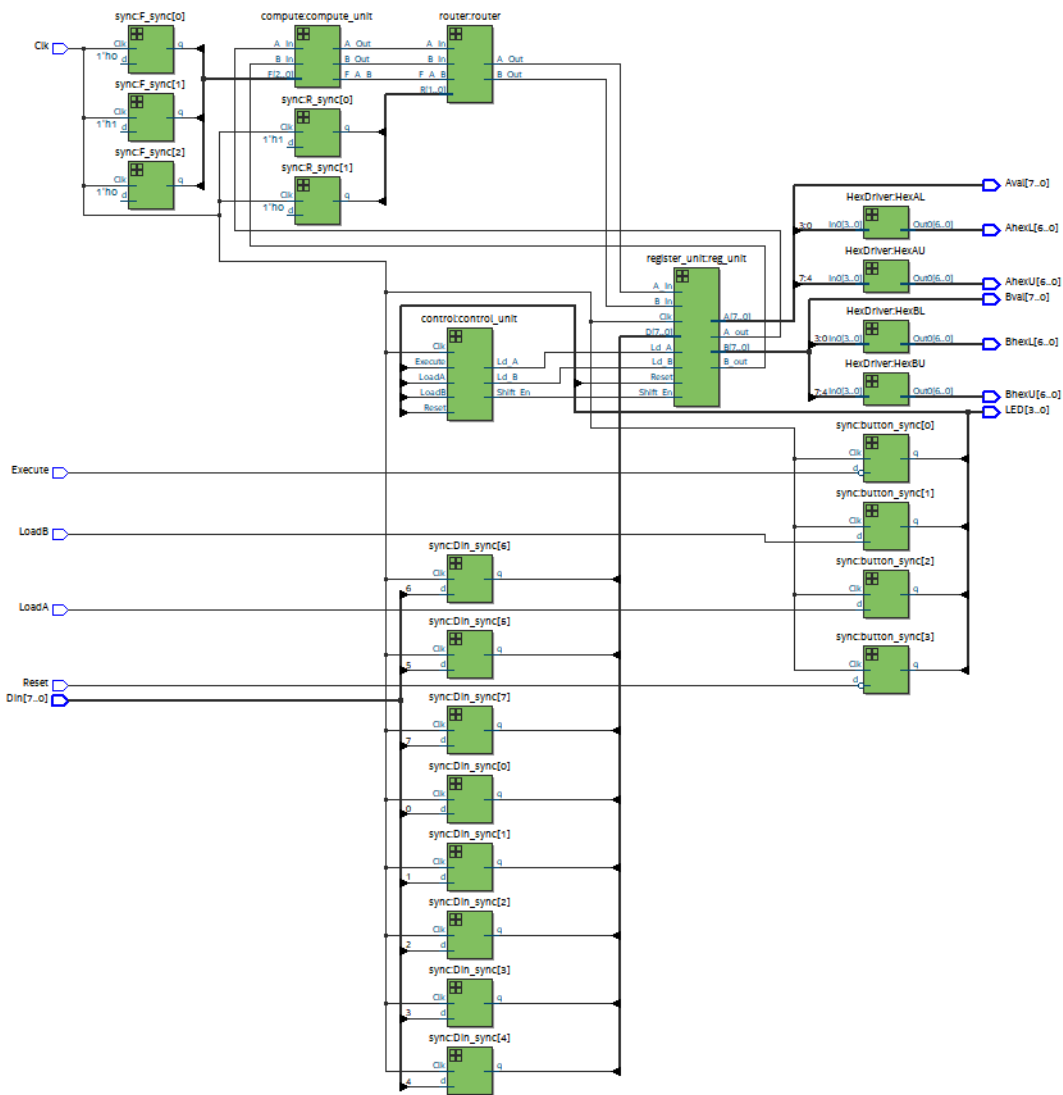


Figure 7: RTL Block Diagram of Top Level Processor Module

## Processor Simulation

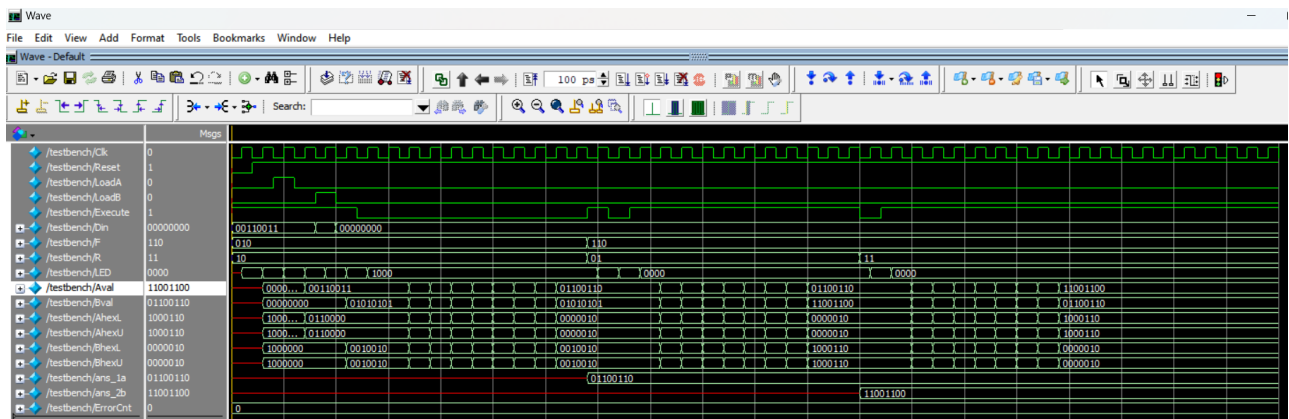
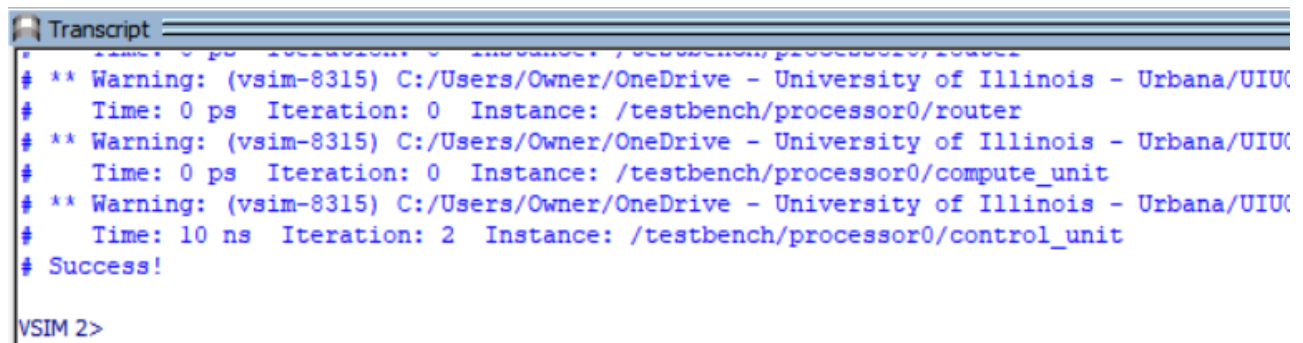


Figure 8: Wave Simulation of Top Level Processor Module



```
Transcript
# ** Warning: (vsim-8315) C:/Users/Owner/OneDrive - University of Illinois - Urbana/UIUC
# Time: 0 ps Iteration: 0 Instance: /testbench/processor0/router
# ** Warning: (vsim-8315) C:/Users/Owner/OneDrive - University of Illinois - Urbana/UIUC
# Time: 0 ps Iteration: 0 Instance: /testbench/processor0/compute_unit
# ** Warning: (vsim-8315) C:/Users/Owner/OneDrive - University of Illinois - Urbana/UIUC
# Time: 10 ns Iteration: 2 Instance: /testbench/processor0/control_unit
# Success!

VSIM 2>
```

Figure 9: Simulation Transcript Confirming Successful Functionality

## SignalTap ILA Trace

## Troubleshooting Bugs and Glitches

As we were initially working through the experiment, we ran into some obstacles that we needed to figure out how to overcome. One problem we had initially was manually assigning our F and R values. Although we manually assigned the input values when we attempted to debug through the testbench in ModelSim we had not known that we needed to input F and R as logic input instead. Once we were able to complete this, we could then move on to checking if extending our bits from four to eight would function properly once we were able to assign our inputs properly instead of on a switch. Another issue we ran into was solving our next-state logic in the Register for S1 and S0 since we could not get the counter chip to stop after four cycles. We solved this problem by setting up the FPGA so we can utilize a debounced button instead of creating our own to implement with the counter chip.

# Lab Questions

## Inverting a Signal

**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 two-input one-output circuit that can optionally invert a signal is a two-input XNOR gate. In our design for the computation unit, we had to decide with F2, F1, and F0 which operation we will be computing and sending to the routing unit for all four of the bits we were computing. When F2 is low, the possible signals that we compute are the inverted signals of if F2 is high. Because of this pattern, we only implemented A NAND B, A NOR B, A XNOR B, and 0000 using our chips and put each of their output signals into a 4-to-1 mux. With selection inputs F1 and F0, we chose which value of  $f(A, B)$  we would choose of the four and then we input that value into an XNOR gate. Since the other input of the gate was the F2 input when we have F2 selected to high, the output remains the same; however, if the F2 selection input is low, then we will invert our signal accordingly so we now use the inverted computation. For example, if we complete an A XNOR B function and have a result of 1100, when we set F2 to high, our  $f(A, B)$  output will stay the same. On the other hand, if we want to compute A XOR B, then setting F2 to low as an input in our XNOR gate will invert the resulting  $f(A, B)$  so we get the correct A XOR B output which would be 0011.

## Modular Design

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

In our experiment, we have four distinct sections that we must implement together to create the overall functioning unit that loads A and B into the register and shifts the unit bit-by-bit to then compute the chosen function to then finally return our preferred output bits into the designated registers. Our modular design is specifically split up into the Control unit, the Register unit, the Computation unit, and the Routing unit. Each section of the design functions individually to complete its task but the output of each section becomes the input of the next section. In our situation, we load two four-bit values to A and B as “build components” and similar to an assembly line we send all of the build components through the line and complete tasks to return a final product. In this context,



our task is completing the chosen computation which is then sent down the “assembly line” by the routing unit as a final product to where it should be loaded. Since we knew how each individual function would work on its own this significantly cut down on development time as we could build each part individually and test them by providing example inputs to see if the output is correct. Building each part individually cut down on debugging time as well as the testability improved such that any error we found we could see which part it was in and move on to resolving it.

## Moore vs. Mealy Finite State Machines

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

The FSM is responsible ensuring that logic processor performs exactly one operation on the 4-bit data, and that the clock cycle does not repeat itself until Execute is turned back off for at least one clock pulse, before being turned on again to perform another operation. To build a functioning FSM, we first had to decide between a Moore machine or a Mealy machine. While both are viable options, we implemented the Mealy machine because it depends on both the current state and current inputs of the FSM, allowing it to function with fewer states than a Moore machine.

We used the FSM design given in the lab document since it was the simplest implementation of the Mealy machine and we knew we would be able to successfully implement it in our control unit. The Mealy machine inputs are 1-bit Execute signal  $E$ , 1-bit state representation  $Q$ , and a 2-bit count  $C1C0$ . Execute starts the computational cycle,  $Q$  represents the two states of the Mealy machine, which are the reset state and the shift/halt state. The count bits represent the current number of shifts, and counts up to 3 before returning to 0. The counter is held in the shift/halt state, and helps to reduce the total number of states required to implement the FSM. The outputs of the Mealy machine are the register shift bit  $S$ , the next state bit  $Q^+$ , and the 2-bit next count  $C1^+C0^+$ .

The transition arcs are labeled  $C1C0/E/S$ , where  $C1C0$  is the current count input,  $E$  is the execute input, and  $S$  is the register shift output. The FSM starts in the reset state, where  $EQC1C0 = 0000$ . The FSM will remain in this state until Execute is turned **ON**, starting the cycle. While Execute is low, the register unit and the next state also remains the same, where  $SQ^+C1^+C0^+ = 0000$ . The next state does not change until the next clock edge after Execute is flipped high, where  $EQC1C0 = 1000$ . The resulting next state is then  $SQ^+C1^+C0^+ = 1101$ , where  $S = 1$  because shift is enabled,  $Q^+ = 1$  because the next state is the shift/halt state, and  $C1^+C0^+ = 01$  since the count has been enabled so the FSM begins counting to decimal 3. Once four shifts are completed, the FSM remains in the

shift/halt state, where  $SQ^+C1^+C0^+ = 0100$  until Execute is turned back off for one clock edge, at which point it returns to the reset state and  $SQ^+C1^+C0^+ = 0000$ . This represents one full complete cycle of the logic processor, and waits in the reset state to perform another cycle when Execute is switched back ON.

## ModelSim vs. SignalTap

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

ModelSim and SignalTap are both simulation tools used in digital circuit design, but they serve different purposes and have different features. ModelSim is a simulation tool that we use for verifying the functionality of our circuits. It allows us to simulate the circuit with timing and gate-level simulations so we can verify if the design's behavior is correct and debug any digital design problems that may appear. SignalTap, on the other hand, is a debugging tool on the chip level integrated into our Altera FPGA development. It allows us to look at and analyze the internal signals of the FPGA during operation in real-time, allowing us to debug problems that may appear when we initialize our system or configure our FPGA. Furthermore, we use SignalTap to capture data from the FPGA and generate waveforms. The two are different in this sense as ModelSim can be used to see if our digital circuit is functioning properly, whereas SignalTap allows us to debug problems related to the FPGA operation. In this experiment, we used a testbench with ModelSim to verify if our digital circuit files are functioning correctly. We used SignalTap when we were instead developing the physical breadboard circuit to see if every value was correct in each register during each bit-shift after we hit the execute trigger. SignalTap allowed us to debug any problems that appeared when we developed the physical circuit as we could see where the error in our logic was and resolve it immediately.

## Conclusion

All in all, we were able to successfully finish both parts of the experiments. We built a bit-serial logic operation processor with a modular design on a breadboard that can load four-bit inputs in two registers and compute a logic operation between the two chosen by the user and store it in registers also chosen by the user. We implemented this part of the lab with a Mealy machine and an FPGA as a debounced button to represent our clock signal. Once execute is switched high, the loaded shift registers shift A and B for four clock signals, one bit for each clock signal, so the computation unit can complete the logic operation and the routing unit may store the output back into whichever register has

been chosen. For the next part of the lab, we were provided SystemVerilog code which was a logic operation processor for four bits like our circuit in part 1, and were tasked with extending it to compute eight bits. We changed the inputs to eight bits, adjusted the output hex LEDs accordingly, and added four more states to the Mealy machine so the code can compile and compute logic operations for eight-bit inputs A and B. Although we ran into various obstacles along the way, we were able to overcome them to create a very interesting circuit in part one that allowed us to demonstrate more of what we have been learning as ECE students for the past few semesters. Part two of the experiment introduced us to SystemVerilog's ability to digitally complete what we have been learning to do physically and theoretically and allowed us to dive into how to manipulate the code to make the hardware behave how we would like it to.