ECE 385

Lab 2

A Logic Processor

Fall 2025

Apoorva Sharma
NetID: as204

Samridhi Verma
NetID: sv49

Introduction

The purpose of this lab is to build a 4-bit serial logic operation processor on the breadboard in the first part. The design should be able to perform 8 different functions and route the results back into the registers based on the input from the user. The second part of the lab involves extending the 4-bit serial logic processor to 8 bits on SystemVerilog, where we also learn to use Vivado, vSim, and debug cores.

The lab report is divided into two sections: the first details the 4-bit serial logic processor that we designed on the breadboard, and the second details how the provided code was modified and extended to an 8-bit processor.

4-Bit Serial Logic Processor

Introduction

The first part of the lab involves building a 4-bit serial logic processor on a breadboard using 4-bit shift registers, MUXes, and basic logic gates (NAND, NOR, XOR). Each of the registers holds 4 bits, which are entered through the switches. The design can perform 8 logical operations: AND, OR, XOR, set all bits to '1', NAND, NOR, XNOR, set all bits to '0'. The routing unit routes the results back into the registers. Each register performs a right shift 4 times when execute is pressed, using outputs from the routing unit as the new most significant bit for each shift.

Written Description of the Circuit

Figure 1 shows the block diagram for the serial logic processor. The components circled in blue are the inputs to the circuit.

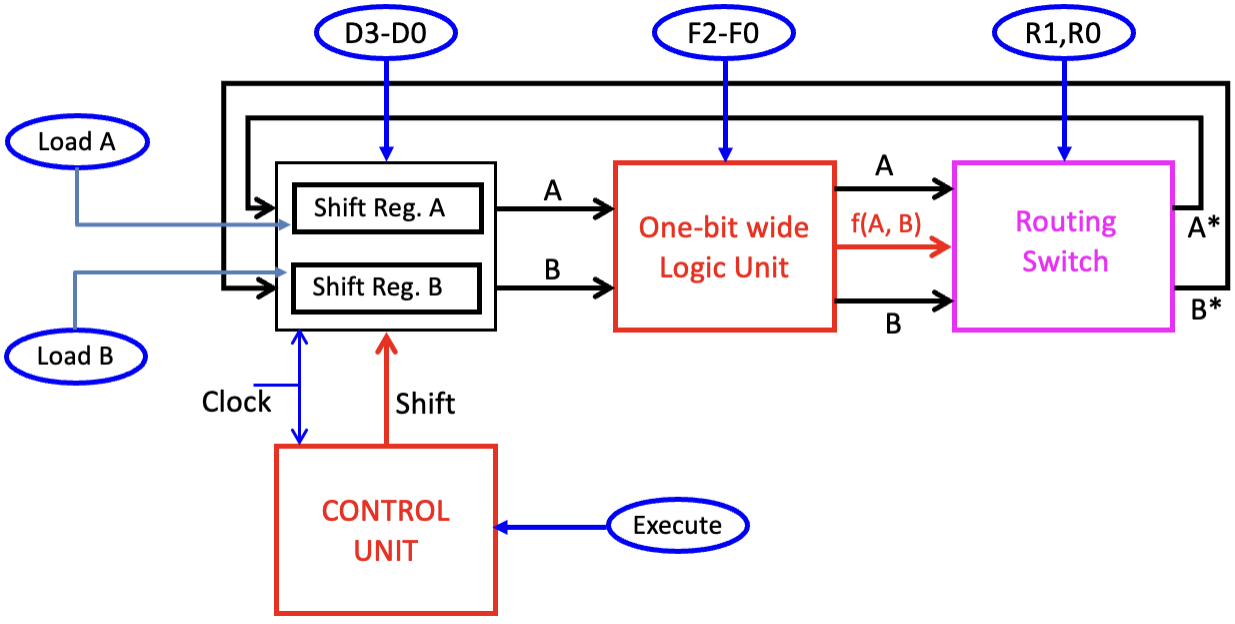

Figure 1. Block Diagram for the Serial Logic Processor

(Courtesy of Professor Zuofu Cheng)

Figure 2 shows the component layout for the entire circuit. A1 and A2 compose the Register Unit; B1, B2, B3, B4 are the Computation Unit; B5 is the Routing Unit; C1, C2, C3, C4 are the Control Unit; and A3 provides the combinational logic for the S1 input into the Register Unit.

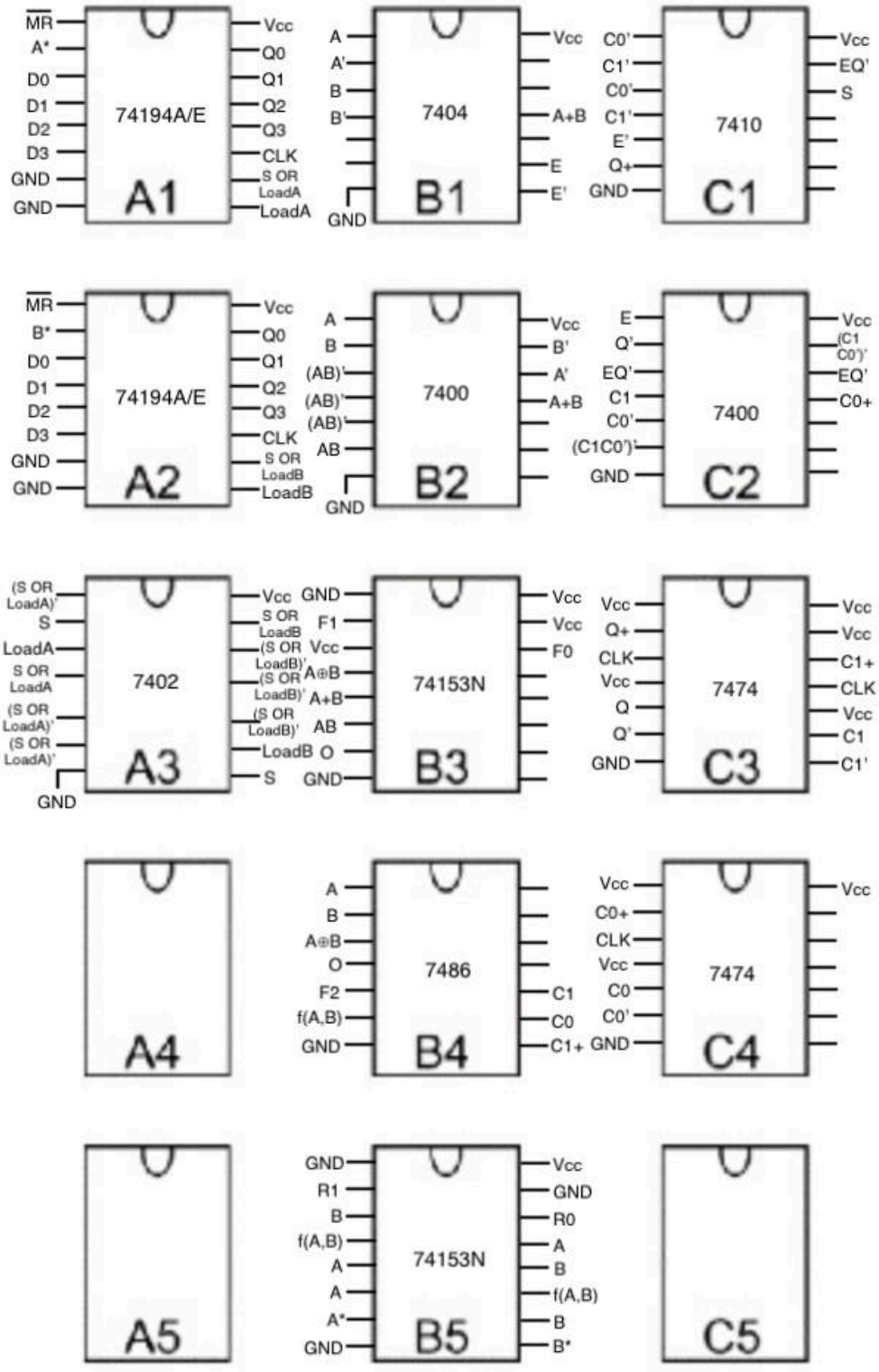

Figure 2. Component Layout for the Serial Logic Processor

Figure 3a shows the implementation of the circuit on the actual breadboard and *Figure 3b* shows the detailed circuit schematic..

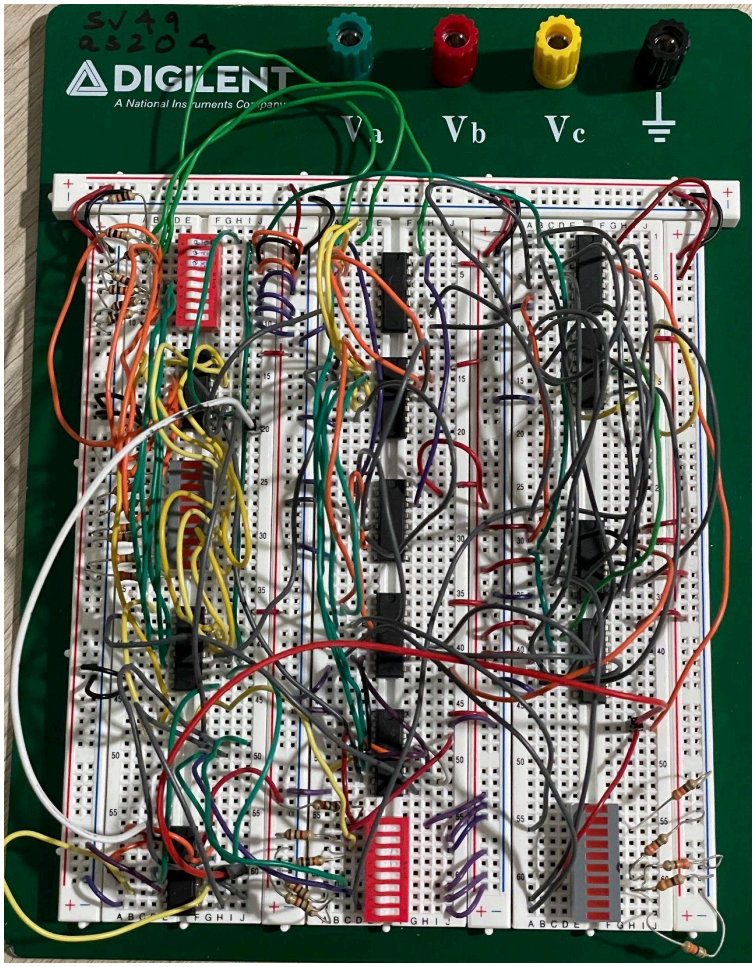

Figure 3a. Breadboard Implementation of the Circuit

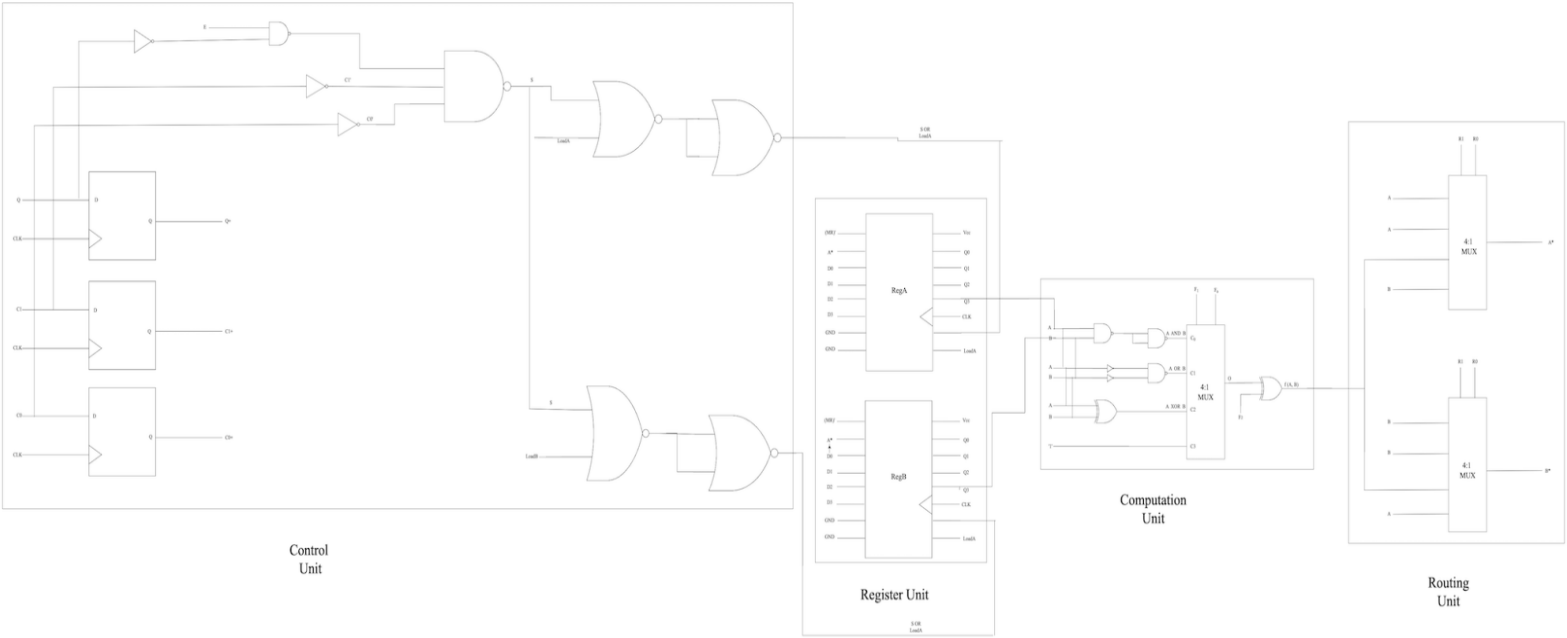

Figure 3b. Detailed Circuit Schematic

Register Unit

The register unit consists of two 4-bit shift registers, each of which holds the values of RegA and RegB, respectively. The values in the register are set using 4 switches and two separate buttons to control which register is supposed to load - LoadA and LoadB. We set the first 4 switches to the initial value we want in RegA and turn on the switch for LoadA. Then we turn off the switch for LoadA, set the value for B using the same 4 switches, and turn on the switch for LoadB. The values in these registers after execution starts are controlled by the control unit, and the serial input is provided by the routing unit. We used 8 LEDs to show the values stored in RegA and RegB at each stage.

For this part, we used two of the 4-bit BIDIR Shift Registers (CD74HC194E). This chip is capable of loading, holding, and right shifting 4 bits, which is what we needed for this lab. The various connections for this chip are shown in *Figure 4*.

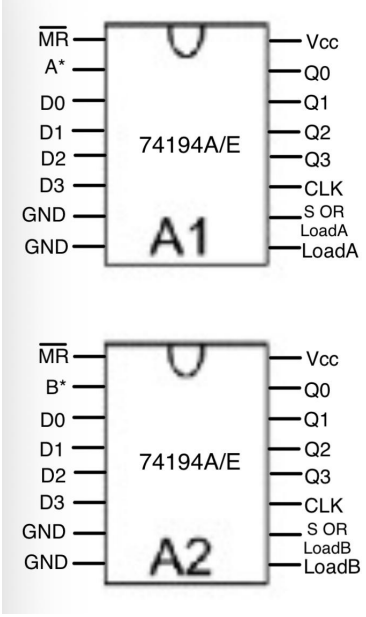

Figure 4. Register Unit

D0, D1, D2, and D3 are the 4 bits from the switches. Since the lab only required right shifting, we disabled the left shift input by connecting it to ground. Q0, Q1, Q2, and Q3 are the outputs from the registers, which are connected to the LEDs for each of the registers. This helps with debugging and verifying the correctness of the circuit at every step. The Reset pin is connected to a switch, which will enable the user to reset the contents of the register (set all values to 0).

Both RegA and RegB use a common clock, which is obtained from the FPGA board. This common clock ensures that the shifting and loading are synchronous. The serial input line for the right shift operation comes from the routing unit.

S1 and S0 together determine when the registers should load, hold, or shift values. S1 and S0 are obtained through combinational logic using combinational logic on the output S from the control unit. The truth table for the combinational logic is given in *Table 1*. LoadA will be applicable for RegA, and LoadB for RegB. For simplicity, we refer to LoadA/LoadB as L in the truth table.

Table 1. Truth Table for Combinational Logic for S1

S	L	S1	S0
0	0	0	0
0	1	1	1
1	0	0	1
1	1	x	x

The register holds the value in the register when S1S0 = 00, performs parallel load when S1S0 = 11, and shifts right when S1S0 = 01 (information obtained from the datasheet for chip CD74HC194E). From the truth table, we infer that S1 is simply the value of LoadA or LoadB for RegA and RegB, respectively.

Table 2. K-Map to Simplify Combinational Logic for S1

	L = 0	L = 1
S = 0	0	1
S = 1	1	x

Using the K-Map in *Table 2,* we simplify the expression for S1 to obtain S1 = S + L. S1 and S0 together ensure the three proper modes of the register: load, hold, shift.

Computation Unit

The computation unit is responsible for executing the desired logical combination based on the function selection inputs and sending that output to the routing unit. *Table 3* shows the computation unit output based on the function selection inputs.

Three switches are used to manually set the values of F2, F1, and F0. The current least significant bits from RegA and RegB are the inputs A and B used in the computation unit. This part of the circuit can perform AND, OR, XOR, set all bits to ‘1’, NAND, NOR, XNOR, and set all bits to ‘0’. To do this, we pass circuits performing AND, OR, and XOR, and set all bits to ‘1’ through a 4:1 MUX. The select lines to this MUX are F1 and F0, which are the two least significant bits of the function selection inputs. The output of this MUX, O, is passed through an XOR gate along with F2, which is the most significant bit of the function selection input. This configuration allows the circuit to generate all eight

operations using only four of the logic function circuits, and this follows the same truth table shown in *Table 3*. *Table 4* shows the truth table for O XOR F2.

Table 3. Truth Table for Computation Unit Output Based on Function Selection Inputs

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

Table 4. Truth Table for O XOR F2

O	F2	f(A, B) = O ⊕ F2
0	0	0
0	1	1
1	0	1
1	1	0

According to *Table 4*, the output is 1 when O and F2 have different values. This part of the circuit basically acts as an inverter that flips the value of O depending on whether F2 is 0 or 1. If F2 = 1, f(A, B) = O XOR 0 = 0. If F2 = 0, (A, B) = O XOR 1 = O'. Clearly, when F2 is 1, the output from the 4:1 MUX is inverted, which is what we require since the inverse of AND is NAND, OR is NOR, XOR is XNOR, and '1' is '0'. From *Table 4*, this is exactly what we want - for the output to be flipped every time F2 is 1.

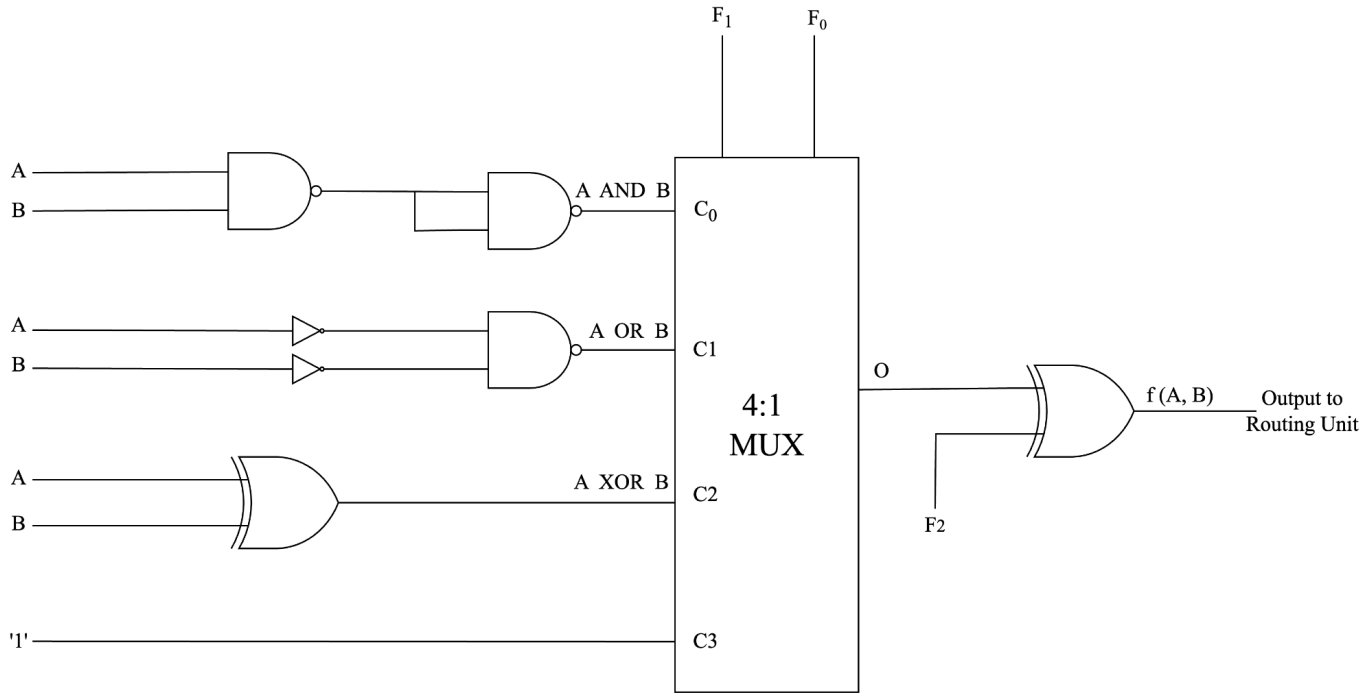

Figure 5. Circuit Schematic for the Computation Unit

The computation unit comprises 3 NAND gates, 2 inverters, 2 XOR gates, and a 4:1 MUX, as shown in *Figure 5*. 2 NAND gates are used to build the AND gate, 1 NAND gate along with 2 inverters, 1 XOR gate for the XOR gate, and 1 XOR gate for the final output from the 4:1 MUX and F2. This meant using 1 hex inverter chip (CD74AC04E), 1 quad NAND chip (CD74AC00E), 1 dual 4:1 MUX chip (SN74HC153N), and a QUAD XOR chip (SN74HC86N).

We used NAND gates to construct most of the AND and OR gates. This is because using NAND gates would require less number of chips compared to using two separate chips for AND and OR. *Figure 6* shows the AND gate implementation of AND, and *Figure 7* shows the OR gate implementation of OR. The circuit connecting to C0 in the MUX in *Figure 5* is the NAND gate implementation of AND, and the circuit connecting to C1 in the same figure is the NAND gate implementation of OR.

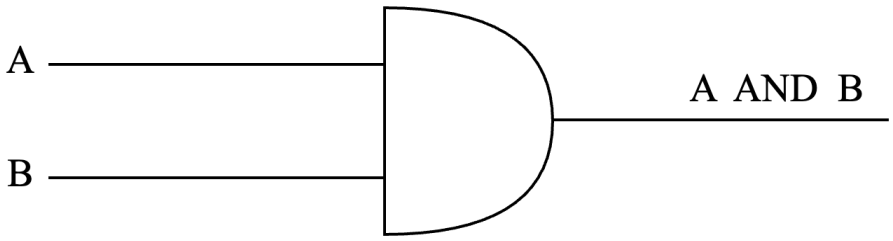

Figure 6. AND gate implementation of AND

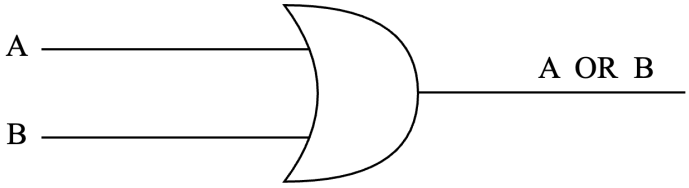

Figure 7. OR gate implementation of OR

The output from the computation unit is sent to the routing unit.

Routing Unit

The routing unit's main task is to route the desired signals A' (new A bit) and B' (new B bit) back to the register unit after the logical computation has been completed by the computational unit. The outputs A, B, and f(A, B) of the computational unit are used as inputs for the routing unit. The selection signals R1 and R0 are used to select the desired signals A' and B' as the output. We used a dual 4:1 MUX chip (SN74HC153N) to implement the routing unit. *Table 5* shows the truth table for the desired output based on the inputs R1R0.

Table 5. Truth Table for O Routing Unit Output Based on Function Selection Inputs

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

The four different outputs are sent into the MUX with R1 and R0 as the select lines. Based on the select lines, one of the 4 values is chosen for A' and B' respectively as shown in *Figure 8*.

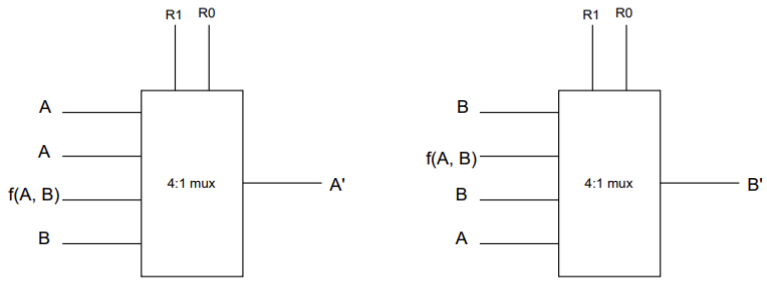

Figure 8. Circuit Schematic for the Routing Unit

Control Unit

The control unit generates control inputs for the register unit. The inputs for this unit are Load A, Load B, Execute, and the Clock signal. The Load A and Load B signals parallelly load data from the input switches (D3 - D0) into the A and B registers. The Execute signal tells the control unit to begin the computation cycle. The control unit sends the signal to right shift the register unit four times and then halts until the Execution is set to low and then high again. The Clock input is provided by the FPGA. The control unit can either be clocked fast at 1kHz or clocked slow at 1Hz.

For the internal logic of this unit, we used a Mealy FSM. This implementation is shown in *Figure 9*. A Mealy Machine's output depends on both the current state and the current input. It helped group similar operations into one state, using a combination of the current state and the current input to operate, reducing the total number of states used.

The inputs of the FSM are Execute, single-bit state representation Q, and the 2-bit count C1C0. The Execute signal initiates the circuit computation. The single bit 'Q' is split into 2 states: Reset and Shift/Hold. The count bits C1C0 are used to track the number of shifts in the shift/hold state. The outputs of the mealy machine are the 'Reg. Shift' ('S') signal, which goes to the register unit to tell it when to shift, the next state Q^+, and the next count $C1^+C0^+$. When the machine is in the Reset state (Q = 0, C1C0 = 00) and Execute is low (E = 0), nothing happens; that is, the registers hold their values. When Execute switches to high (E = 1), the FSM immediately outputs a shift pulse (S = 1), moves to the Shift/Hold state (Q = 1), and starts incrementing the counter. In the Shift/Hold state, the counter shifts the registers for four clock cycles (S = 1 during each shift). After the fourth shift, the counter resets to 00, and the FSM stays in the Shift/Hold state with S = 0, holding the values of the registers. Execute signal being pressed during shifting does not affect the shifting.

The hardware works by connecting the outputs of the FSM (S, Q^+, $C1^+$, $C0^+$) to the registers and counters, while the internal FSM logic decides what the outputs should be for each clock cycle based on the current state and inputs.

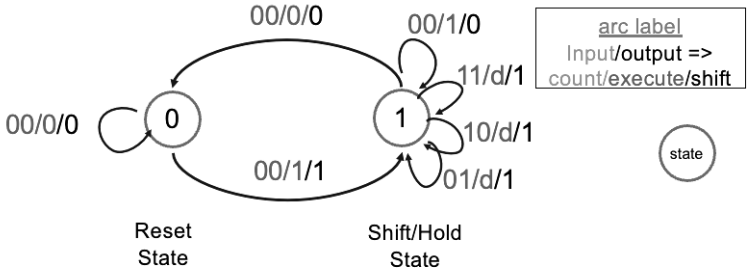

Figure 9. State Diagram for the Control Unit Mealy Machine

(Courtesy of Professor Zuofu Cheng)

The next state logic S, Q^+, $C1^+$, and $C2^+$ are calculated using the K-Maps shown in *Table 6, 7, 8, 9*.

Table 6. K-Map for Reg. Shift ('S')

	C1C0 = 00	C1C0 = 01	C1C0 = 11	C1C0 = 10
EQ = 00	0	d	d	d
EQ = 01	0	1	1	1
EQ = 11	0	1	1	1
EQ = 10	1	d	d	d

Sum of Products (SOP) for S: S = C0 + C1 + EQ'

Table 7. K-Map for Q^+

	C1C0 = 00	C1C0 = 01	C1C0 = 11	C1C0 = 10
EQ = 00	0	d	d	d
EQ = 01	0	1	1	1
EQ = 11	1	1	1	1
EQ = 10	1	d	d	d

Sum of Products (SOP) for Q^+: Q^+ = C1 + C0 + E

Table 8. K-Map for $C1^+$

	C1C0 = 00	C1C0 = 01	C1C0 = 11	C1C0 = 10
EQ = 00	0	d	d	d
EQ = 01	0	1	0	1
EQ = 11	0	1	0	1
EQ = 10	0	d	d	d

Sum of Products (SOP) for $C1^+$: $C1^+$ = C1'C0 + C1C0' = C1 XOR C0

Table 9. K-Map for $C0^+$

	C1C0 = 00	C1C0 = 01	C1C0 = 11	C1C0 = 10
EQ = 00	0	d	d	d
EQ = 01	0	0	0	1
EQ = 11	0	0	0	1
EQ = 10	1	d	d	d

Sum of Products (SOP) for $C0^+$: $C0^+$ = C1C0' + EQ'

For the hardware design implementation, we used NAND, NOR, and XOR gates to build the circuits for S, Q^+, $C1^+$, and $C2^+$. Shown below is the conversion of the AND/OR expression to the NAND/NOR expression for the next state logic bits:

S = C0 + C1 + EQ' => S= (C0' C1' (E Q')')'

Q^+ = C1 + C0 + E => Q^+= (C1' C0' E')'

$C1^+$ = C1'C0 + C1C0' => C1 XOR C0

$C0^+$: $C0^+$ = C1C0' + EQ' => ((C1 C0')' (E Q')')'

Figure 10 shows the combination logic circuit for S.

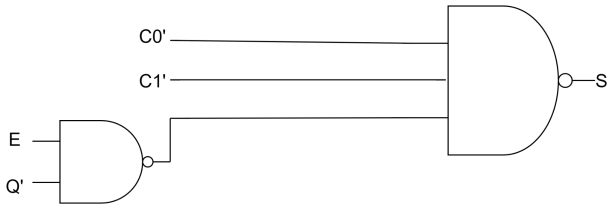

Figure 10. Circuit schematic for S

Figure 11 shows the combination logic circuit for Q^+.

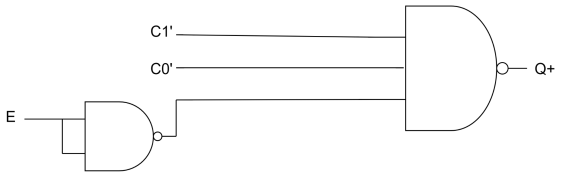

Figure 11. Circuit schematic for Q^+

Figure 12 shows the combination logic circuit for $C1^+$.

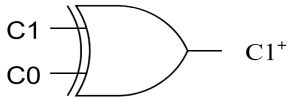

Figure 12. Circuit schematic for $C1^+$

Figure 13 shows the combination logic circuit for $C0^+$.

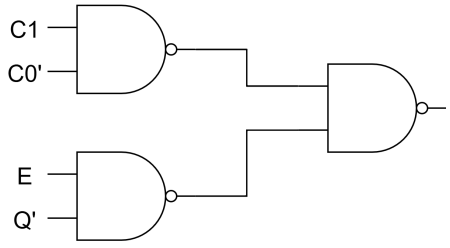

Figure 13. Circuit schematic for $C0^+$

We used three D flip-flops: one to store the state Q and the other two to store the incremented values of the counter bits $C1^+$ and $C0^+$.

Key Considerations

Switches and LEDs

For switches, we made use of two of the 8-DIP Switches (chip 76SB08T). One was used to load the register values in A and B and input values of select bits of the computational unit F2, F1, and F0. The other was used to input the select bits of the routing unit R1 and R0, and also initialize Execute, Load A, Load B, and Reset. We pulled down all the switches so that there are no floating values that will result in incorrect outputs. This ensured that each input stayed at a stable low value (0) whenever no voltage was applied.

For LEDs, we used the Multisegment LED (chip 76SC04T) to ensure that the correct values were input into the registers. We also pulled down the LEDs for the same reason as above.

Computational Logic

It is possible to build the control unit with an 8:1 MUX where each of the 8 logical functions is connected to a separate input line of the MUX. This would require additional gates, consequently additional chips for the 4 additional logical operations. Instead, by using only a 4:1 MUX and generating the other four functions by XORing the MUX output with the most significant function select bit, we minimized the number of chips used. This design conserves hardware resources and also simplifies the circuit without sacrificing functionality.

Counter

We did not use a counter to increment the values of $C1^+$ and $C0^+$. Instead, we used two D flip-flops to store the incremented values of $C1^+$ and $C0^+$, respectively. The control unit keeps track of how many times the registers have shifted by updating these flip-flops. With each shift, the next $C1^+$ and $C0^+$ values are fed into the flip-flops, effectively incrementing the count. Once four bits in the registers have shifted, the control unit then automatically stops the shifting process.

Circuit Operation

The left-most DIP Switch was used to load the register values in A and B and input values of select bits of the computational unit F2, F1, and F0. The top three switches represent F2, F1, and F0. The bottom 4 switches represent the values D3, D2, D1, and D0 that would be loaded into registers A and B. We left the 4th switch from the top ideal to help us differentiate between the inputs (F2- F0) and (D3-D0).

The bottom-most DIP Switch on the middle row of the breadboard was used to input the select bits of the routing unit R1 and R0, and also initialize Execute, Load A, Load B, and Reset. The top 2 switches

represent R1, R0. The third switch represents Reset. We left the 4th and 5th switches ideal. The 6th, 7th, and 8th switches represent Load A, Load B, and Execute.

Issues Faced and Corrective Measures Taken

One of the problems we faced was incorrectly connecting the input switches. We had connected the switches to pull-up resistors, which caused unintended behavior when a switch was pressed. We corrected this by changing all pull-up resistors to pull-down resistors, that is, connecting each switch to ground using a resistor. This ensured that the switches drove the inputs high or low consistently and also ensured stable logic signals without glitches.

Conclusion

In conclusion, we successfully designed and implemented a 4-bit serial logic processor on a breadboard using shift registers, MUXes, and basic logic gates. The processor was able to perform 8 logical operations and route the computed result back into the registers based on user-selected inputs. Each component was carefully designed to operate synchronously and efficiently. The key challenge we faced was ensuring stable input signals for the switches, which we resolved by switching from pull-up to pull-down resistors. This project not only enhanced our understanding of digital logic design and the interaction between different components but also provided valuable hands-on experience with debugging and testing real-world circuits.

8-Bit Serial Logic Processor

Introduction

The second part of the lab involves extending the provided 4-bit serial logic processor SystemVerilog code to work for 8 bits. This required us to edit the control unit and the register unit. The computation and routing units worked the same because they only deal with 1 bit at a time. The processor is implemented in Vivado. Debug Cores and vsim are used to verify and find any errors in the design. This part of the lab also helped us explore SystemVerilog, Vivado, vsim, and debug cores.

Module Descriptions

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 module instantiates and connects the registers, computation unit, routing unit, control unit, synchronizers, and hex display driver to form the complete system.

Purpose: This is the top-level module of the 8-bit logic processor. It ties together all components, handling input, data flow, and display output.

Module: **Register_unit.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: This module instantiates the two 8-bit registers, `reg_A` and `reg_B`. Each register supports parallel load, right shift, hold, and reset. The inputs `A_In` and `B_In` provide the data for shifting. `D` provides the data for loading.

Purpose: This module provides the 8-bit registers that store operands A and B for use in the processor.

Module: **Reg_v.sv**

Inputs: `Clk, Reset, Shift_In, Load, Shift_En, [7:0] D`

Outputs: `Shift_Out, [7:0] Data_Out`

Description: This is the 8-bit register with synchronous load, right shift, hold, and synchronous reset functions. Depending on the control signals, the register either loads parallel data from `D`, shifts in serial data from `Shift_In`, or holds its current value on the positive edge of `Clk`.

Purpose: This module is used to create the registers that store operands A and B in the serial logic processor circuit.

Module: **compute.sv**

Inputs: `[2:0] F, A_In, B_In`

Outputs: `A_Out, B_Out, F_A_B`

Description: This module is the 1-bit computation unit. Based on the 3-bit function select input F, it performs the logical operations AND, OR, XOR, set all bits to '1', NAND, NOR, XNOR, or set all bits to '0'. on `A_In` and `B_In`. It produces the output `F_A_B`. `A_Out` and `B_Out` are the original inputs that the computation unit also passes as output.

Purpose: This module provides the logical operations for the serial logic processor.

Module: **Router.sv**

Inputs: `[1:0] R, A_In, B_In, F_A_B`

Outputs: `A_Out, B_Out`

Description: It determines which value should be sent to the registers after computation based on the 2-bit input `R`. It determines whether `A_Out` and `B_Out` should be the original values (`A_In` and `B_Out`), values should be switched or if one of those should be `F_A_B`.

Purpose: This module is responsible for routing the signals back to the register unit after computation.

Module: **Control.sv**

Inputs: Clk, Reset, LoadA, LoadB, Execute

Outputs: Shift_En, Ld_A, Ld_B

Description: This is the control unit for the entire serial logic processor. It has an internal counter (s_count0 to s_count7). It uses the counter states to enable shifting for eight cycles when Execute is high. The machine begins in s_start, steps through the count states, and finishes in s_done. This module controls the loading of the registers, shifting, and decides when execution starts and ends.

Purpose: This module provides the counter to perform shifting 8 times and generates the control signals needed to coordinate register loading and shifting during execution.

Module: **HexDriver.sv**

Inputs: Clk, reset, [3:0] in[4]

Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: This module drives the 4-digit 7-segment display. It converts each 4-bit input nibble into the corresponding hexadecimal display for the 7-segment display.

Purpose: This module provides the visual output of the register values on a 7-segment display for debugging and seeing the functioning of the circuit.

Module: **Synchronizers.sv**

Inputs: Clk, d

Outputs: q

Description: This module synchronizes and debounces the input from the switch. It uses flip-flops and a counter to ensure that the output only changes after the input has been stable for a certain number of clock cycles.

Purpose: This module debounces the inputs from the switches to provide a clean and stable signal.

Description of Changes Made to the Provided 4-Bit Processor Code

Changes were made only in the Processor.sv, Control.sv, Register_unit.sv, and Reg_4.sv.

In Processor.sv, inputs Din, Aval, and Bval were changed from 4 bits to 8 bits to support the 8-bit operation. Internal registers A, B, and synchronized output Din_S were also updated to 8 bits. The output hex_seg was also extended from 4 bits to 8 bits to accommodate the larger registers for display.

In `Reg_4.sv`, The input `D` and the output `Data_Out` were changed from 4 bits to 8 bits. The internal signal `Data_Out_d` was also updated to hold 8 bits. The functionality of this module remained the same.

In `Register_unit.sv`, only the input `D` and the outputs `A` and `B` were extended from 4 to 8 bits.

In `Control.sv`, we extended the number of count states. The additional count states `s_count4` through `s_count7` were added to extend the shifting sequence to 8 cycles. Also, the `enum` was extended from 3 bits to 4. A 3-bit `enum` would only be able to represent 8 states at most, while we needed 10. A 4-bit enum can represent up to 16 states, so we had to extend it to be able to represent all the 10 required states in the 8-bit implementation.

Thus, by essentially only modifying the Control Unit and Register Unit, we were able to extend the 4-bit register to deal with 8 bits.

RTL Block Diagram

Figure 14 shows the RTL diagram for the 8-bit serial logic processor implementation on SystemVerilog.

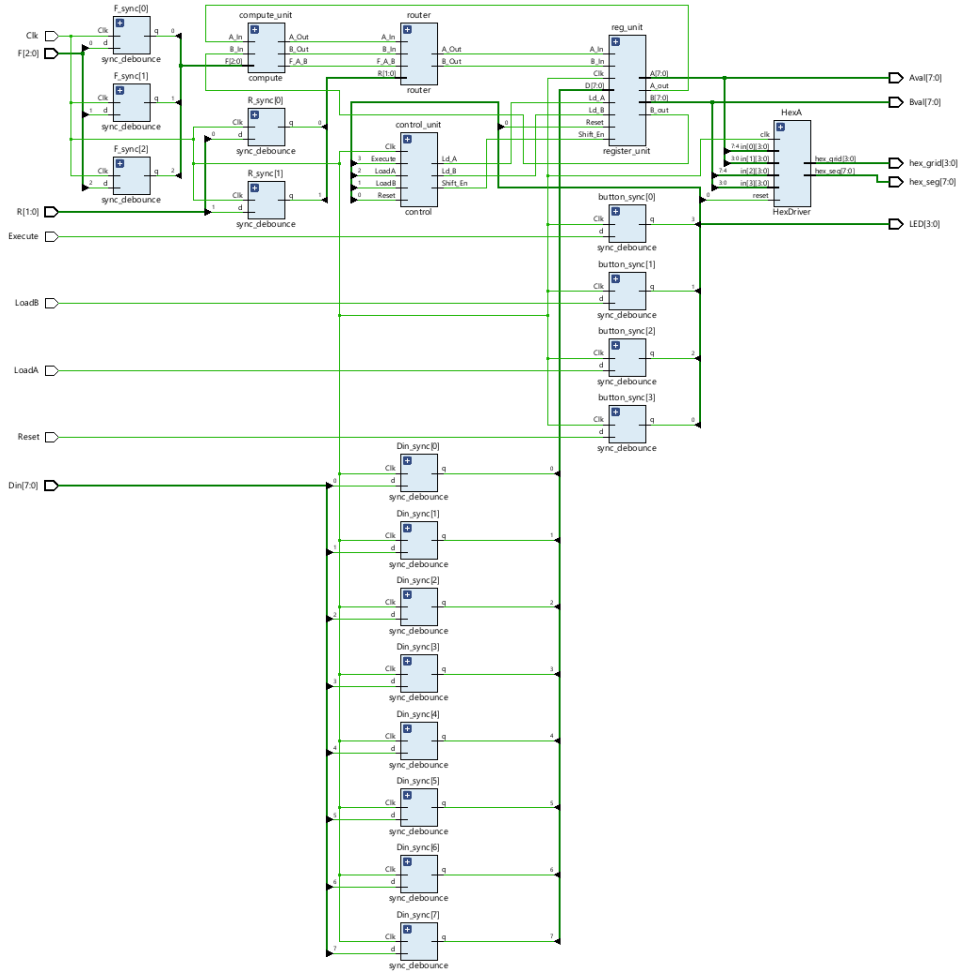

Figure 14. RTL Block Diagram

Simulation

The simulation tool that enables users to test and verify the design before programming it onto hardware.

Figure 15 shows the annotated Simulation waveform for the provided testbench.

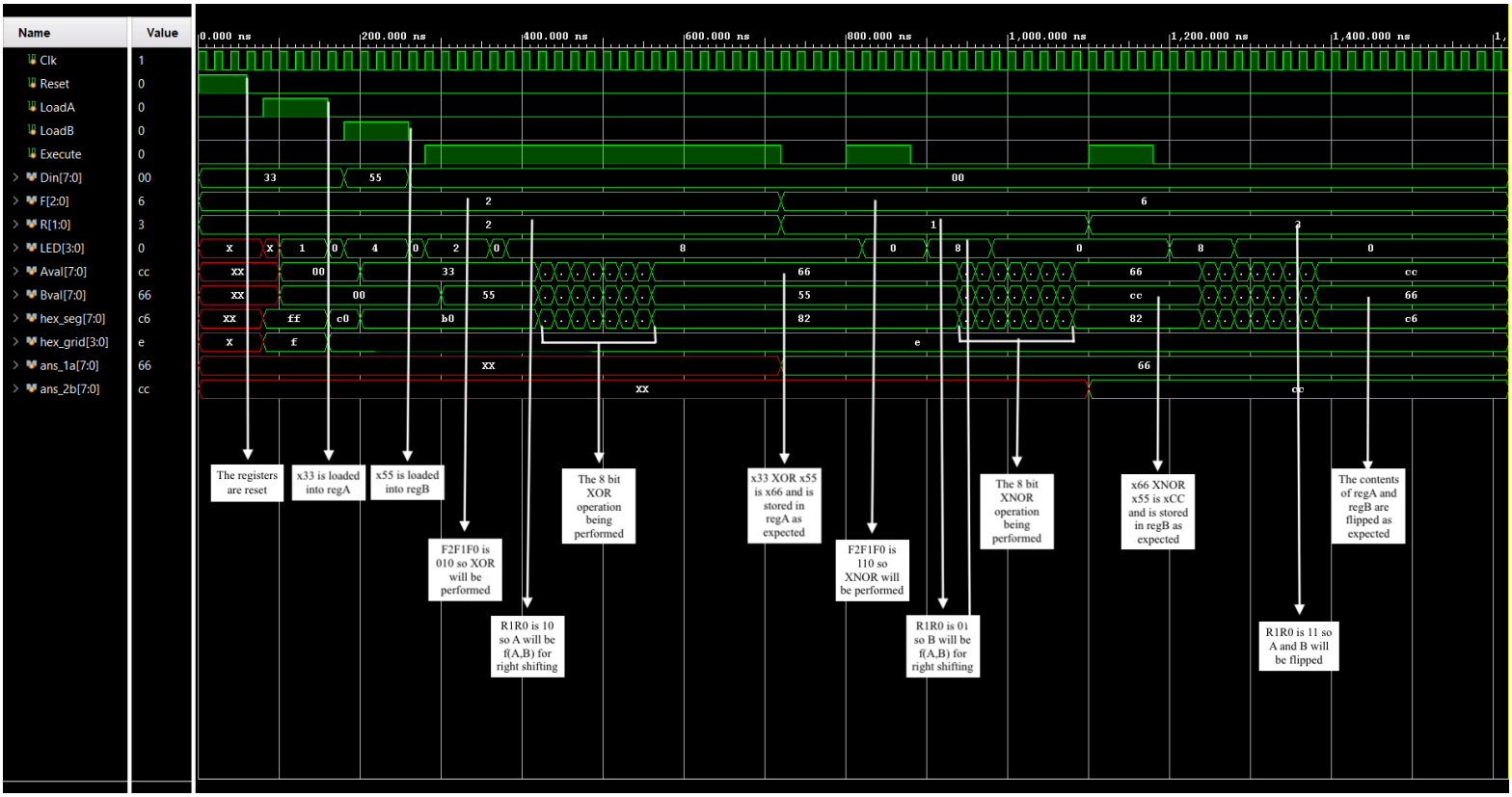

Figure 15. Simulation Waveform

Vivado Debug Core

The Debug Core is used to test the actual hardware functionality on the FPGA.

To generate a debug core, we follow the following procedure:

- Open Synthesized Design.
- Select A_val, B_val, Execute on the Synthesized Design, right click, and select ‘mark as debug’.
    - A_val, B_val give us an insight into the values of the two registers before, during, and after Execution.
    - Execute is chosen because it starts the computation process and acts as the appropriate trigger value that would enable us to correctly understand the output.
    - We mark the Execute signal after it exits the debouncer. This is to ensure we get a stable Execute signal that we can set as Trigger. Otherwise, there may be glitches and the Debug Core might not function properly.
- Choose the ‘Run Debug Core’ option from Synthesis → Synthesized Design → Set Up Debug

- For ‘Nets to Debug’, setup as shown in *Figure 16.*
    - A_val_OBUF is the value in regA
    - B_val_OBUF is the value in regB
    - LED_OBUF is the Execute signal after passing through the debouncer.

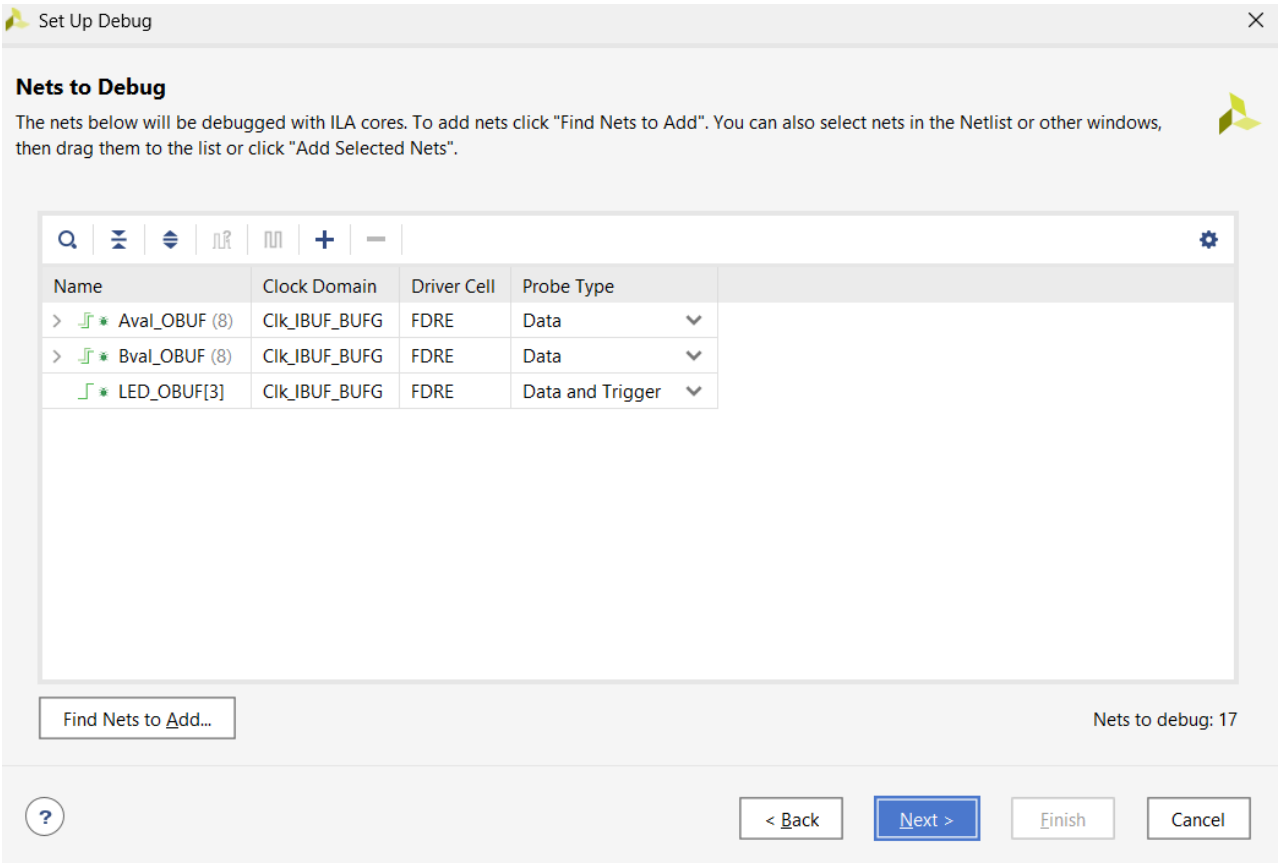

Figure 16. Debug Core Setup

- We then generate the bitstream and program the device.
- We select Execute under the Trigger Setup and set the Trigger value as ‘R’.
- We then ran the trigger for the ILA core.
- We loaded the values for regA, regB, F2F1F0, and R1R0.
- We then pressed Execute on the FPGA board.
- Following this, we observed the serial shifting process and confirmed that after 8 cycles, the registers stabilized with the expected values.

To demonstrate the functioning of the Debug Core, we used the FPGA to set regA to 10101010, regB to 01010101, F2F1F0 to 010, and R1R0 to 01. On pressing execute, we expected regA to retain its value, regB to be set to f(A, B), which in this case would be 11111111 or xff. This is what we observed on the Debug Core Waveform, as shown in *Figure 17*.

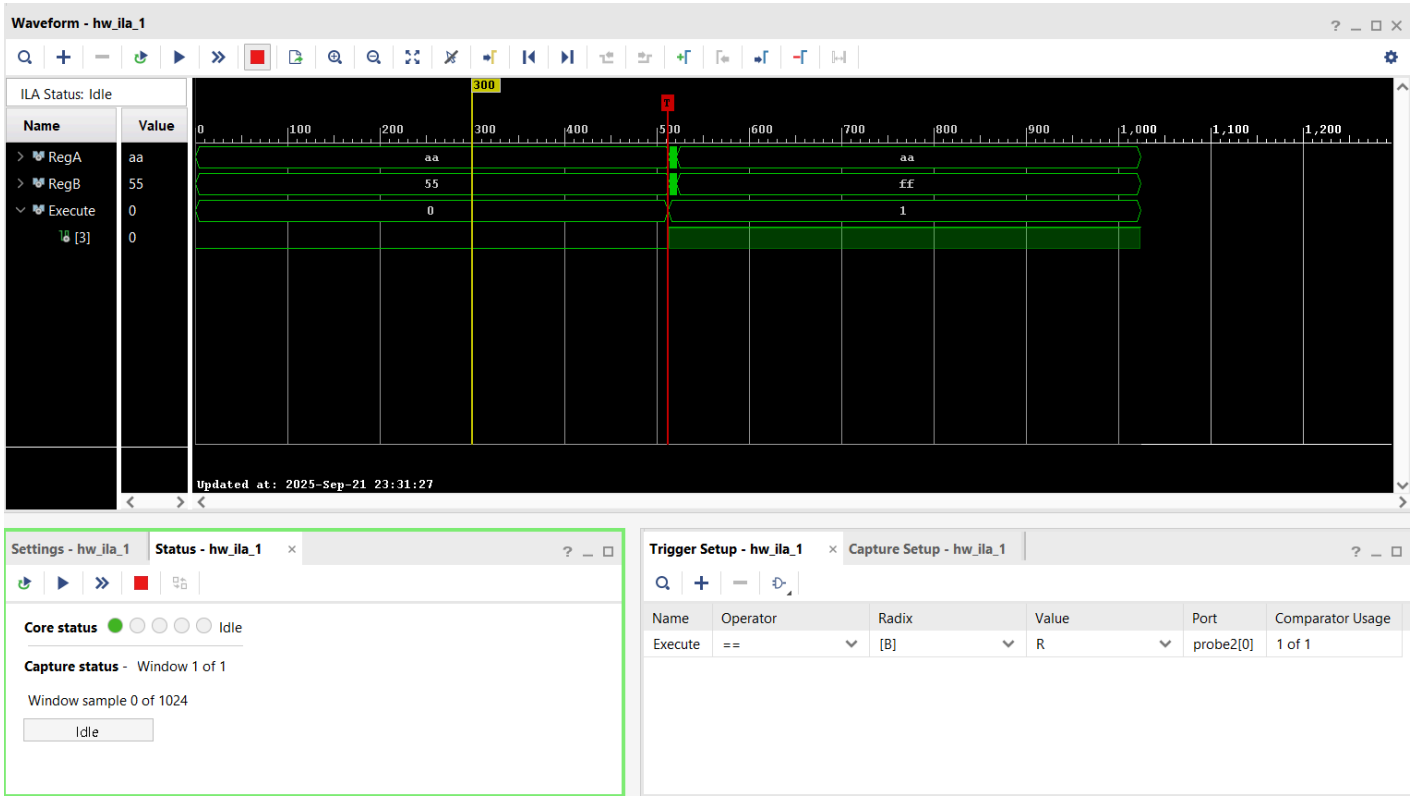

Figure 17. Debug Core Waveform

Bugs and Corrective Measures

We had initially misunderstood that the output `LED` was the output for the register values, so we extended that to 8 bits to match the 8-bit registers. However, the LED signals were actually intended to show the states of `Execute`, `LoadA`, `LoadB`, and `Reset`. Extending the LED width added extra I/O pins that were not defined in the .xdc file, causing the bitstream to fail during synthesis. To correct this, we changed the LED output back to its original 4-bit width.

We also encountered several issues with the .xdc constraints file. Some of the pin assignments were incorrect or flipped. For example, the `LoadA` and `LoadB` buttons were swapped, and the LEDs to display `RegA` and `RegB` were also flipped. We went through the file manually and corrected all of the misaligned pins to take inputs and display outputs correctly.

Conclusion

In conclusion, we extended the provided 4-bit serial logic processor to an 8-bit processor by modifying the Control and Register units. We expanded the Register unit to include 8-bit inputs and also increased the number of states of the control unit from 4 to 8 to include extra counts for the bit shifting. The computation and routing modules remained unchanged because they were serialized and processed one bit at a time. We also learnt how to use Vivado and how to code in SystemVerilog. We became familiar with synthesis, implementation, and debugging in Vivado. Debug cores and vSim were used to verify

the design and helped us identify and fix errors in control signals and pin assignments. We also corrected mistakes in the .xdc file to ensure proper hardware operation. Through this process, we gained a deeper understanding of sequential circuits, register operations, and control logic.

Answers to Post-Lab Questions

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

Answer: The simplest circuit that can be used to invert a signal is for Z = A XOR B. The truth table for this logical operation is given in *Table 10*. *Figure 18* shows the circuit schematic for this operation.

Table 10. Truth Table for Z = A XOR B

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

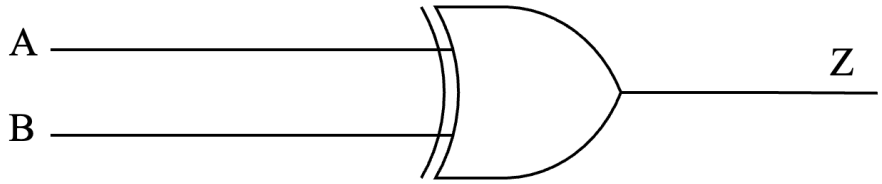

Figure 18. Circuit Schematic for Z = A XOR B

From *Table 10*, we observe the following:

When B = 0,

Z = A XOR 0 = A

When B = 1,

Z = A XOR 1 = A’

Thus, the output is A when B is 0, and the output is the complement of A when the output is 1.

This result is useful in the computation unit of the lab. Using this, we can minimize the number of circuits for the logical operations required in the computation unit. The first four functions required are AND, OR, XOR, and set all bits to '1', and these are the functions for which we end up building individual circuits. For the other four functions (NAND, NOR, XNOR, set all bits to '0'), which are the inverse of the first four functions, we use an XOR gate to flip the output based on the value of F2. This enables us to use fewer chips and almost halves the space the computation unit uses.

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

Answer: We used a Mealy FSM to design our control unit because its outputs depend on both the current state and the current inputs. Due to this, we were able to group similar operations (Shift and Hold) within a single state, which reduced the total number of states needed compared to a Moore machine and simplified the circuit implementation.

The tradeoff is that Mealy machines have a more complex design because they depend on both input and current state for output. Due to the same reason, they are more sensitive to input changes, which can sometimes make the outputs less stable. On the other hand, Moore machines' output only depends on the current state and thus, have a simpler design compared to Mealy machines and also have more stable outputs. However, for this project, a mealy machine was more efficient and practical because it allowed us to minimize hardware.

Question: *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?*

Answer:

vSim	Debug Cores
It is a software simulation tool that enables users to test and verify the design before programming it onto hardware.	It is used to test the actual hardware functionality on the FPGA.
Users write testbenches to simulate inputs.	Uses real inputs from the FPGA.
Users can observe all internal signals in the design.	Users can only pick certain signals to see before running on hardware.

vSim is preferred for early functionality testing of the design, whereas debug cores are preferred for hardware-level debugging once the design is on the FPGA.