ECE 385

Spring 2024

Experiment 2

Lab 2
Report

Ziad AlDohaim (ziada2)
Mohammed Alnemer (mta8)

-2.0 for Logic diagram: must use a CAD tool.
-1.0 for SV Module Descriptions: should explicitly list Module/Inputs/Outputs/Description/Purpos for all modules. *For simulation, reading of the results should be shown via annotation on the image*

Introduction

The main objective in this lab was to design a serial 4 bit logic processor. Our system takes in two 4-bit values and executes all the basic logic functions such as AND, OR, XOR, and all their inverses. The processor also has "1111" and "0000" functions that fill the result with 1's or 0's respectively. Our processor operates sequentially operating its computations one bit at a time until we operate on all four bits.

Operating The Processor

In operating the system we used two 8-dip switches, one used to fill in the registers with the desired four-bit values, and the other to choose the function needed to be operated and where we want to store the result of the operation.

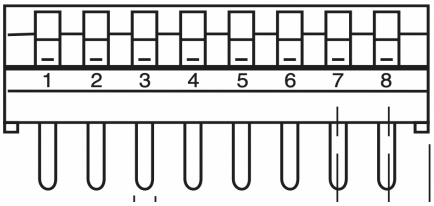


Figure 1: Illustration of the structure of the 8-dip switches

For the switches responsible for the values of the register, the right four bits [5:8] are used to fill the value to be filled into the register and the two leftist switches [1:2] are used directly where we want to load the desired value either in Register A or Register B (switch 1 loads the value into register A and switch B loads the value into register B).

For the switches responsible for executing the system, We have three switches to input which exact operation we want to perform, two switches to choose where we want to store the result of the operation being computed, and one button to enable the execution of the system. The three left switches [1:3] choose the operation to be performed based on a truth table of functions available. The truth table below (figure 3) displayed the exact combination of switched need to perform specific functions with F[2:0] being equivalent to switched [1:3]. The rightest two switches [7:8] are the inputs R1 and R0 which choose where to store the resultant of the operation (see figure 4 for the truth 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 NXOR B
1	1	1	0000

Figure 3: Truth table for computation

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

Figure 4: Truth table for routing operations

Description

We've assembled the 4-bit serial logic processor that's designed to be both efficient and functional within its computing constraints. The processor is composed of several integral units, each with a specific role in the operation of the system.

The Register Unit, consisting of two 4-bit shift registers, is where our binary input data begins its journey. These registers have the ability to hold and transfer data through the system and are controlled by the signals from the Control Unit.

The computational unit performs logical operations on the binary data stored in the registers. It is here that our inputs are transformed, based on the logic function selected by the user through the F2-F0 control inputs.

Once the data is processed, it's passed to the Routing Unit. This unit is critical for directing the flow of processed data, deciding which register to send the data back for further processing or to the output.

Finally, the Control Unit, with its finite state machine, synchronizes the entire process. It controls the state transitions, ensuring that the system operates in the correct mode—loading or executing—at the right times and sequences the interactions between the other units.

Each unit's detailed operations and their synergy will be further elaborated below, demonstrating how they collectively contribute to the functionality of our serial logic processor. This system not only embodies the foundational concepts of digital logic but also showcases the application of these principles to create a working model of a data processor.

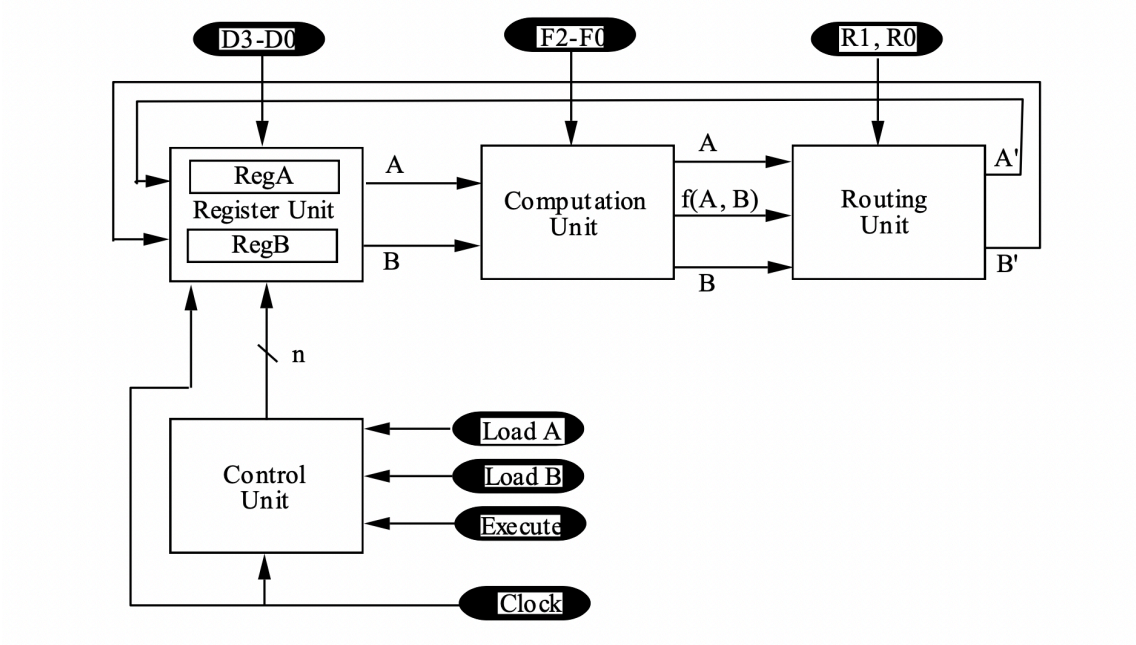


Figure 5: High level block diagram for the 4-bit serial processor

Register Unit

The processor utilizes two four-bit registers, which alternate between two states: loading and executing, directed by the control unit's signal which is further elaborated in the control unit section. Each register, A and B, has a corresponding 'Load' switch, allowing selective data placement. During the loading state, a switch activates, and data from input D[3:0] is loaded into the chosen register.]

As illustrated in figure 6 to the right, inputs D[0:3] are fed from a switch to the registers, while outputs Q[0:3] are displayed on LEDs, reflecting real-time register values. The control unit signals S1 and S0 initiate the load operation, enabling data entry into the register.

In the executing state, the register outputs its stored data to the computational unit and receives the processed results from the routing unit bit by bit. This exchange occurs four times, orchestrated by the control unit's finite state machine (FSM), which signals the registers to shift right for four cycles once the 'Execute' switch is activated.

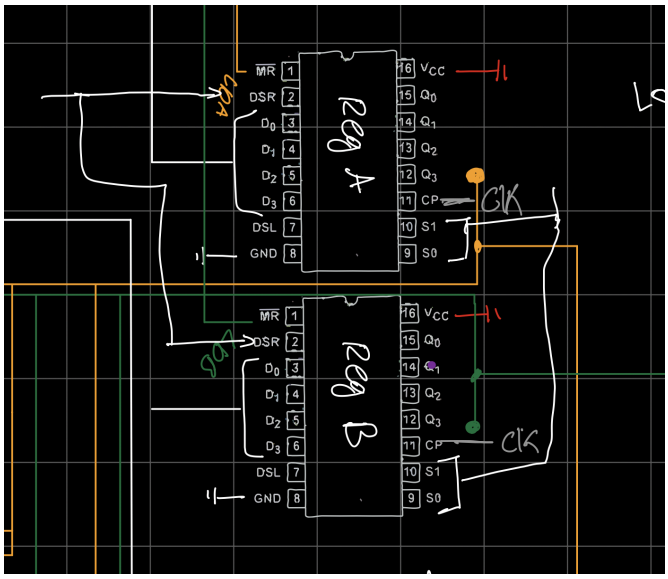


Figure 6 : Registers Illustration

During execution, the bits in the register shift right, with the last bit Q3 linking to the computation unit. The computation unit processes the data from registers A and B and send

the results to routing unit which sends the results back to the register at the DSR Pin. With each clock cycle, a bit is sent to the computational and routing units, and the outcome of this operation is fed back into the registers.

Shifting and Loading the register is directly controlled by the S1 and S0 pins on th registers. when the S1S0 = ‘11’ the registers will be loading the the value of input D[3:0] into the register and when S1S0 = ‘01’ the register will shift right. We control the state of the register using the inputs from the Load A/B switches and the “S” signal which is the control unit command to execute. When the 'S' signal is active, indicating an execution command, the system ensures the registers are in shifting mode, corresponding to the execution state. If both the 'S' signal and the Load A/B signal are active, the system prioritizes execution to prevent loading from interrupting the ongoing process. This prioritization is crucial because it guarantees that the execution state continues seamlessly, even if the Load button is accidentally pressed. Consequently, whenever there is an 'S' signal present, the system ensures the outputs S1 and S0 are set to '01' to maintain the execution without disruption. The truth table below (figure 7) portrayed the logic i/o of the register’s S1S0 pins .

LD A/B	S	S1	S0
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	1

S1 = (LD A/B) *S’

S0 = (LD A/B) + S

Figure 7 : Truth table and logic equations for S1 and S0 inputs of the registers

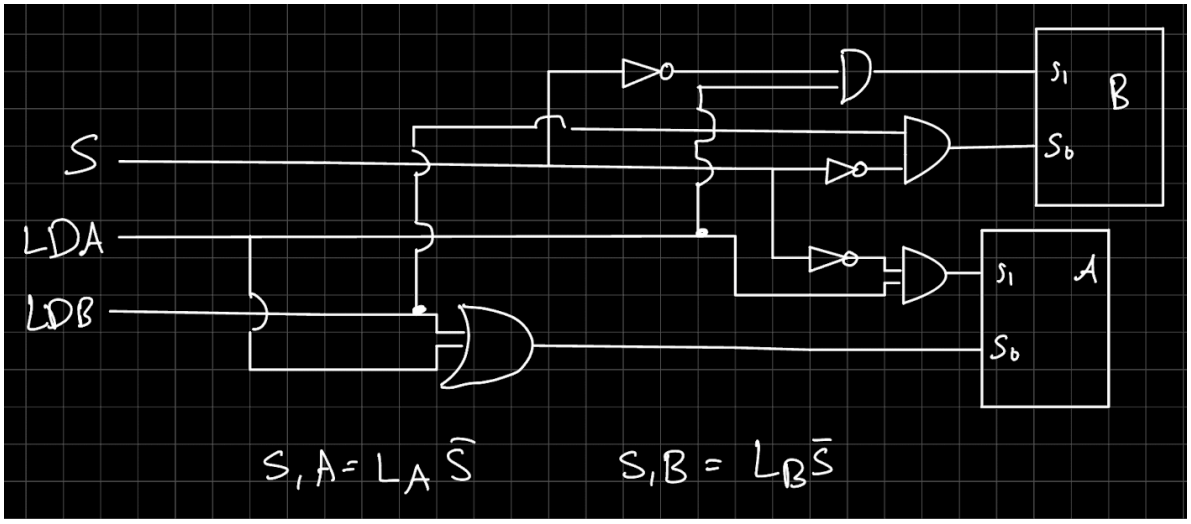
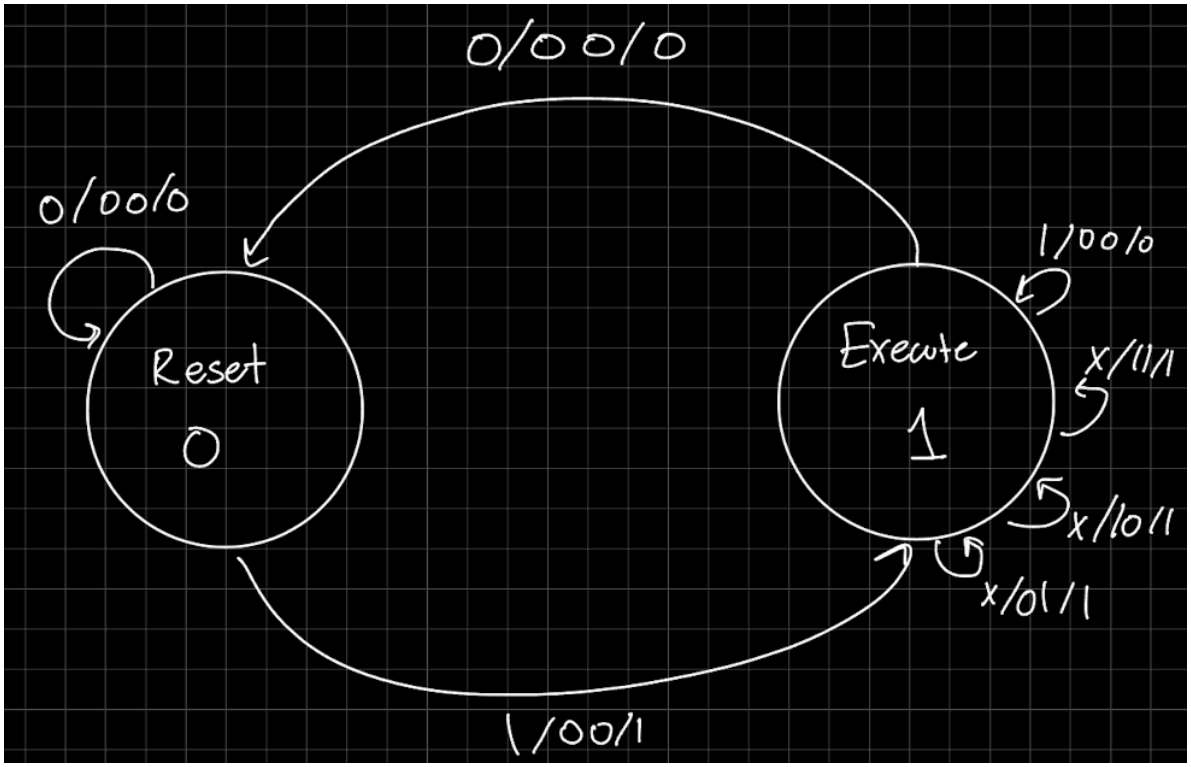


Figure 8 : circuit schematic and gate level logic for S1 and S0 input

Control Unit

The control unit is as the name implies the unit that controls the activity of the system.

The control unit's function is to manage whether the system is in the execution or load/reset state and to oversee the execution process. For the execution process, it precisely times the execution state to ensure that exactly four shifts occur in the register. This allows the registers to sequentially give all the stored bits into the computational unit and receive the proper results. To implement such a system we decided to choose a mealy finite state machine. Using a Mealy state machine instead of a Moore state machine makes our system much more efficient as the function of the FSM is a function of the state giving the ability to control the system more dynamically. Figure 9 below shows our FSM diagram with state transitions and how our system should behave.



“E” Execute signal (input) / “C1C0” Counter (input) / “S” shift enable (output)

Figure 9: Finite state machine diagram with E being the execute signal, C0C1 being the signals from the counter, S is the output of the FSM being the shift enable.

In the context of our processor, the FSM coordinates the core actions of loading and executing. From the reset state, the system can transition to either load data into registers A or B, or move into an execution phase where operations specified by the computational unit are performed. During the execution state, it shifts the register four times allowing all values in registers to go into the computational unit get processed, and allow the output to be stored again. Below in figure 5 are the truth table and logic for our FSM.

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	1	1	0	1
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
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 10: Truth table of the FSM design portraying the inputs and expected outputs of the system

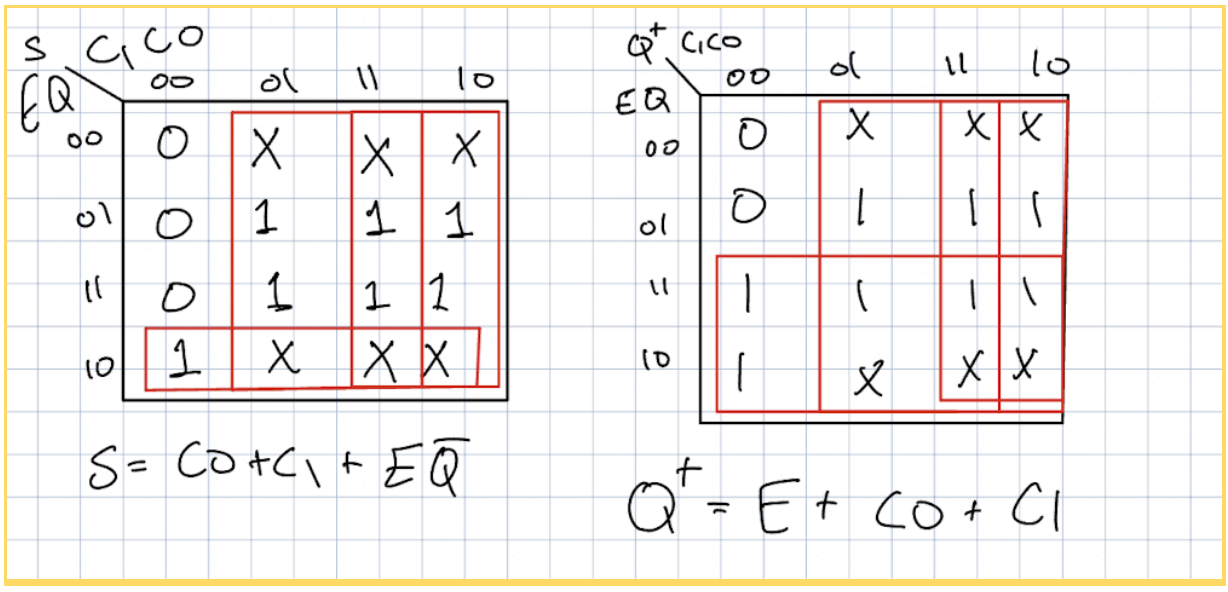


Figure 11: K-maps, and Logic Operations of the control units finite state machine.

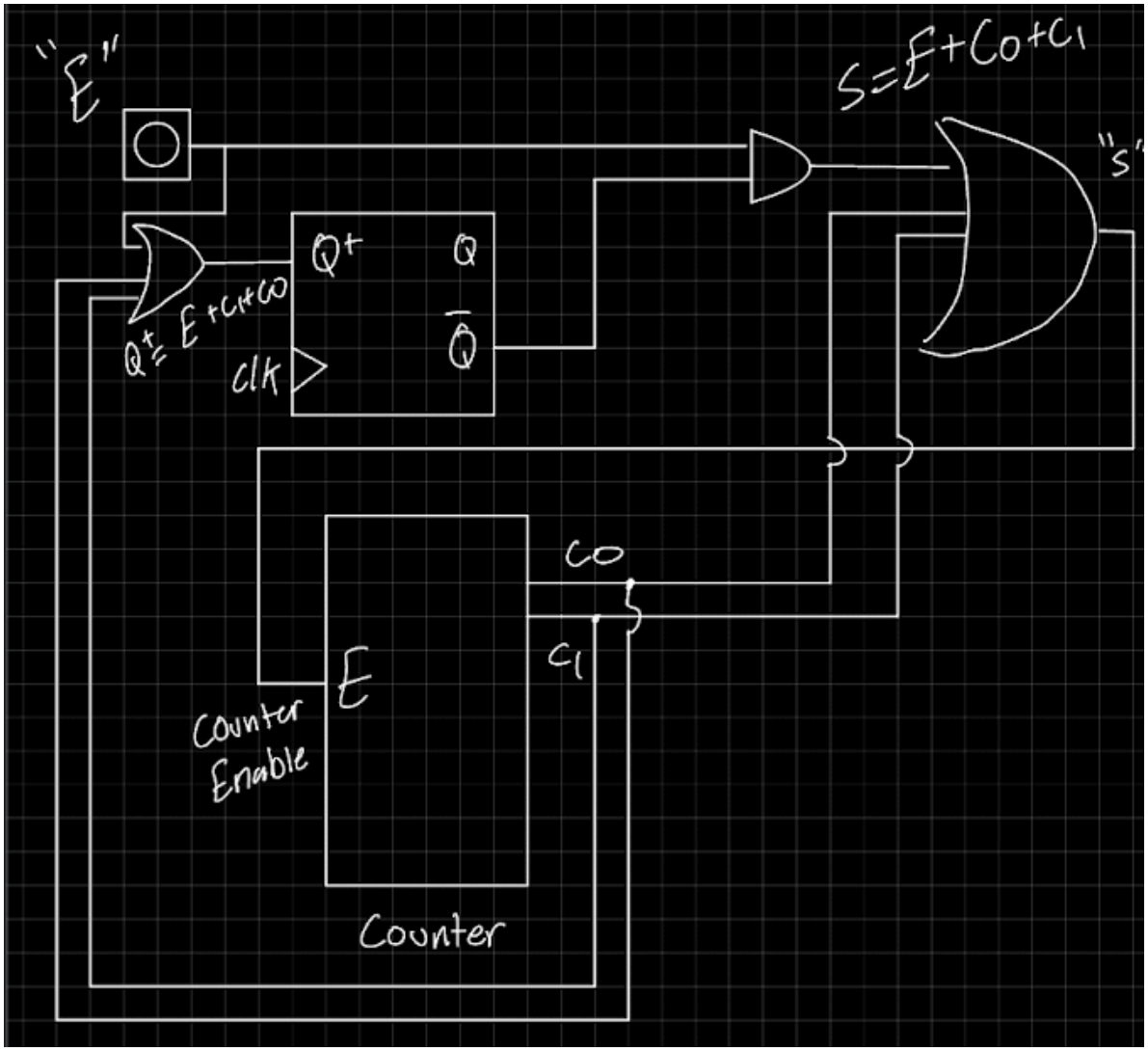


Figure 12: Circuit diagram of the control units finite state machine

Computational Unit

The computational unit is a combinational unit meaning that it does not depend on the clock nor the control unit to operate. It receives two inputs A and B and then through the control inputsF[2:0] it will do a set computation and gives you the result. It instantly gives you the results of the computation it needs to do. During Execute, the register will constantly be giving the ALU new inputs in which the ALU will do a specific operation set by the F[2:0] input set by the user. on the The ALU takes in the values from the register and does the computation set by the F[2:0] input.

Inputs 'A' and 'B' enter the unit, each leading to a set of gates that perform logical operations determined by control signals F2, F1, and F0. These control signals originate from user-defined inputs, selecting the required logic function to be performed on 'A' and 'B'. The unit includes a 4-to-1 multiplexer, which uses the F2, F1, and F0 signals to choose the correct logical operation output. The results from the chosen operation are then forwarded to the routing unit, which directs the output to the appropriate destination within the system.

The table next to the diagram shows the possible logic functions—such as AND, OR, XOR, etc.—that the ALU can perform, contingent on the combination of the F2, F1, and F0 inputs. This setup allows for dynamic computation based on real-time inputs, enabling the execution of various logic operations essential for the processor's functionality.

The computational unit is designed to process four-bit binary values, but it operates serially, handling one bit at a time. This means that each bit from a four-bit input value 'A' and 'B' is fed into the Arithmetic Logic Unit (ALU) sequentially from the registers. The ALU then performs the selected logical operation on these individual bits as they come in. After processing, the results are output one bit at a time to the routing unit, which then directs these bits to their subsequent destinations.

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

Figure 14: Truth table for computational Unit operations.

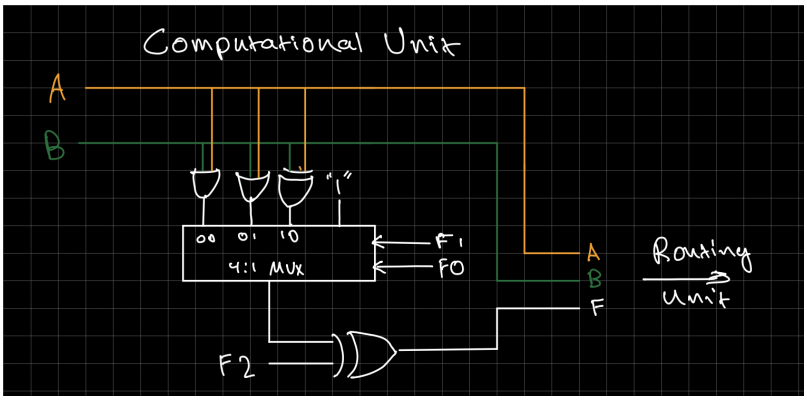


Figure 13: Gate level circuit schematic of the computational unit.

Routing Unit

The routing unit acts as an intermediary within the processor, controlling the direction of data flow post-ALU computation. Specifically, the unit uses control signals R1 and R0 to manage the path of the computational results ('F') from the ALU back into the registers (RegA and RegB). It ensures that the output of each ALU operation is correctly fed back into the registers for further processing or for final storage. This cyclical routing is essential for iterative computational processes, allowing the machine to perform a series of logical operations on the bit-serial data effectively.

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

Figure 15: Routing Unit Truth Table

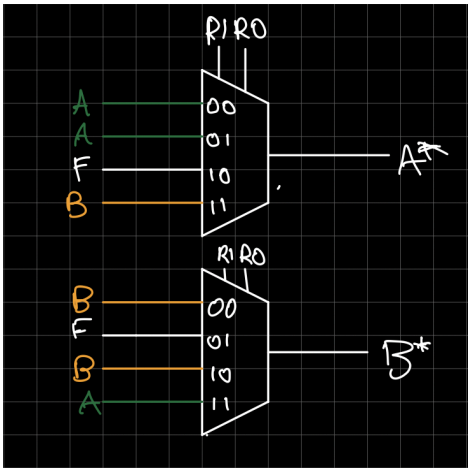


Figure 16: Circuit diagram of the routing unit.

Design

Before implementation, we sketched an outline for each system component.

We initiated with the non-sequential parts—namely, the computation and routing units. Implementing these was relatively straightforward, primarily involving the use of multiplexers (MUXs). The only notable complexity arose when considering how to efficiently implement the inverse operations. Initially, we contemplated using a hex inverter and a secondary MUX. However, we found a more efficient solution by XORing the output of the first MUX—with the four basic functions—with the last input signal. This approach allowed us to perform the inverse operation when the third signal was activated. For the AND, OR, and XOR functions, we chose to implement these exclusively with NAND gates to reduce the number of chips required. Utilizing various chips (NAND, NOR, and inverters) would have necessitated at least five chips, whereas our NAND-only approach required merely three.

Moving to the sequential part of the processor, we began with the registers. Familiarizing ourselves with the chip's structure and functionality, particularly the S1 and S0 pins, was the primary challenge. The input setup was straightforward: we directly connected the four input switches to the register inputs. Subsequently, we explored the interaction between the register and the control unit, focusing on how to shift and load the registers. We have detailed our approach to this design in the aforementioned register unit chapter.

The control unit presented its challenge, centering on the implementation of the finite state machine, the selection of appropriate chips, and particularly the realization of the counter. Having completed the truth table and K-maps for our desired behavior, our task was to figure out how exactly we were going to connect the flip-flop and counter, along with other components of the FSM, while implementing our Boolean logic.

Breadboard view / Layout sheet

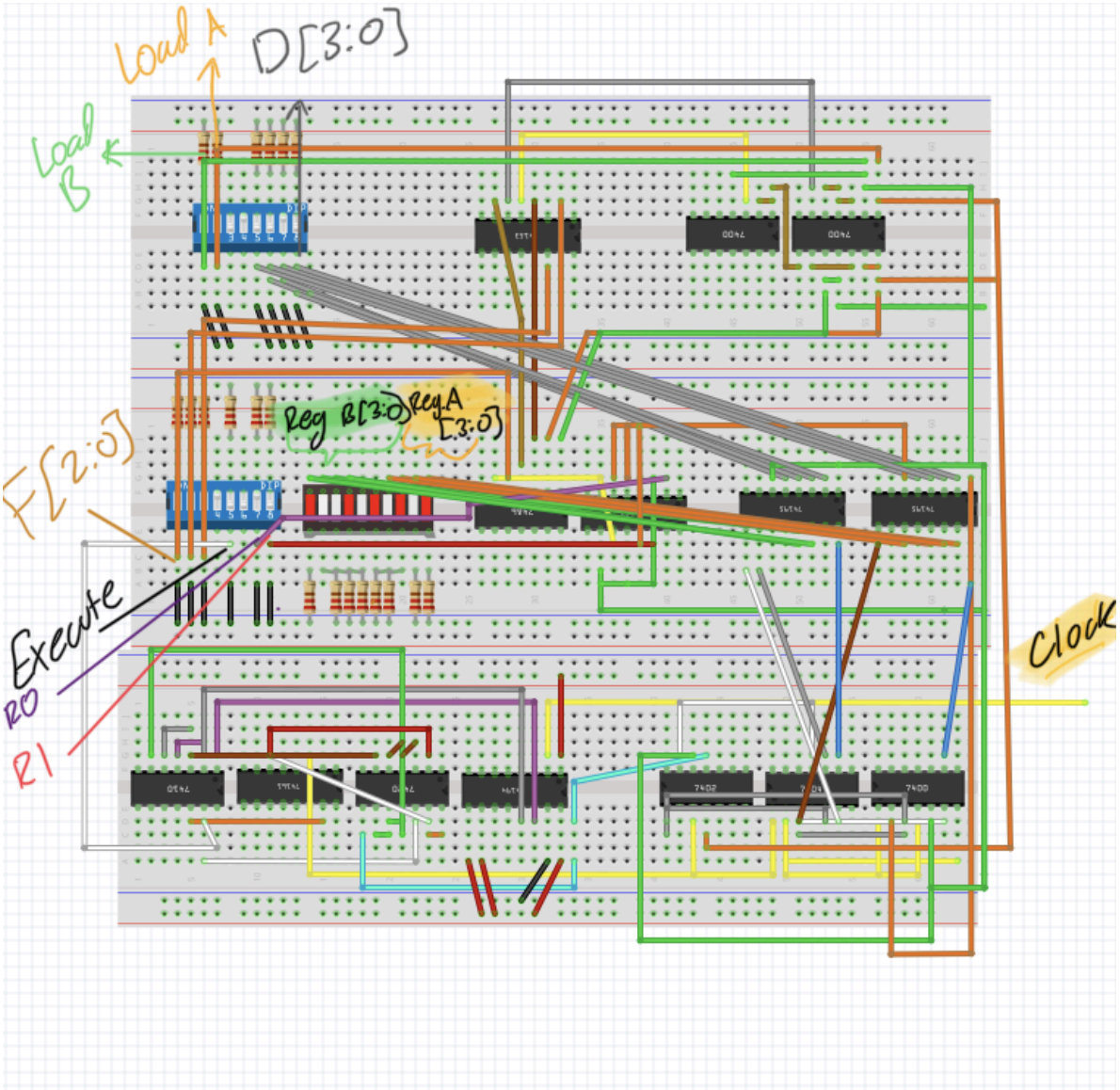


Figure 17: Circuit schematic/breadboard layout using Fritzing with user input labels

LAB 2.2
8-bit logic processor on FPGA
Summary of the modifications applied to our SystemVerilog modules for the purpose of expanding our bit-serial logic processor from 4 to 8 bits.

Processor.sv: This module is the top level of the hierarchy, it takes in all the user inputs which are: Execute, Load A, Load B, Clk, Reset, Din[3:0], F[2:0], and R[1:0]. As well as all the output which are: hex_seg, hex_grid and Debug outputs.[1]

For the 8-bit extension we changed the data inputs and outputs such as Din, Aval, Bval, and the hexadecimal segment display hex_seg, which were all modified to support 8-bit values [7:0]. Additionally, we restructured the logic to handle inputs and outputs for the FPGAs LED display (HexDriver), incorporating an 8-bit synchronous debounce mechanism Din_sync[7:0].

Router.sv:
This module is our routing unit, as its name implies it routes our outputs which are: A_Out and B_Out to the registers based on our inputs which are: R[1:0], A_In, B_In, F_A_B. We did not need to change anything to fit the 8-bit extension.

Reg_4.sv: This module was adapted to handle 8-bit data [7:0] for both input D and output Data_Out, with the initial output set to 8'h00 to signify a clear or reset state.

Control.sv: This is the control unit module, hence it takes the most user inputs. The inputs are: Execute, Load A, Load B, Clk, and Reset. Execute starts the computation when turned on and off (since we use a switch and not a button) Load A and B load in values from the user input D[3:0] to the respective register. Reset stops the shifting and changes the state to the reset state (the first state). The output of this module are: Shift_En, Ld_A, and Ld_B which connect to the register unit (Register_unit.sv).

A critical update was made to the control unit's state machine to accommodate the increased complexity of 8-bit processing. We extended the enumeration logic to [3:0] to account for the necessary nine states, surpassing the limitations of a 2-bit representation and thus moving to a 3-bit logic to add four additional counter states. These new states were integrated within the Mealy machine configuration, setting up a precise sequence that advances to the next state until the completion signal s_done is activated.'

Synchronizers.sv: This acts as a Flip-Flop, which means we did not have to change anything in it to fit the 8-bit design.

HexDriver.sv: This module takes in an 4 or 8 bit sequence and formats it to be displayed on the hexadecimal LEDs to show the values in hexadecimal. The inputs are: in[3:0], ln(), and reset. The outputs are: hex_grid[3:0] and hex_seg[7:0], we did not have to change anything to expand the 4 bits to 8 bits.

[1] LED, Aval, and Bval (these are for debugging)

Register_unit.sv:
This module is our register unit and its inputs are:Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, D [3:0]. The outputs are: A_Out, B_Out, A[3:0] , B[3:0].

We updated the data path within the register unit to support 8-bit operations. The inputs and outputs for registers A and B were both extended to [7:0], ensuring that the processor could handle 8-bit values.

Compute.sv:

These modifications were implemented to ensure that the processor not only maintained its original functionality but also leveraged the capabilities offered by an 8-bit architecture. The transition required a deep dive into the SystemVerilog language and a strategic approach to module expansion.

RTL block diagram

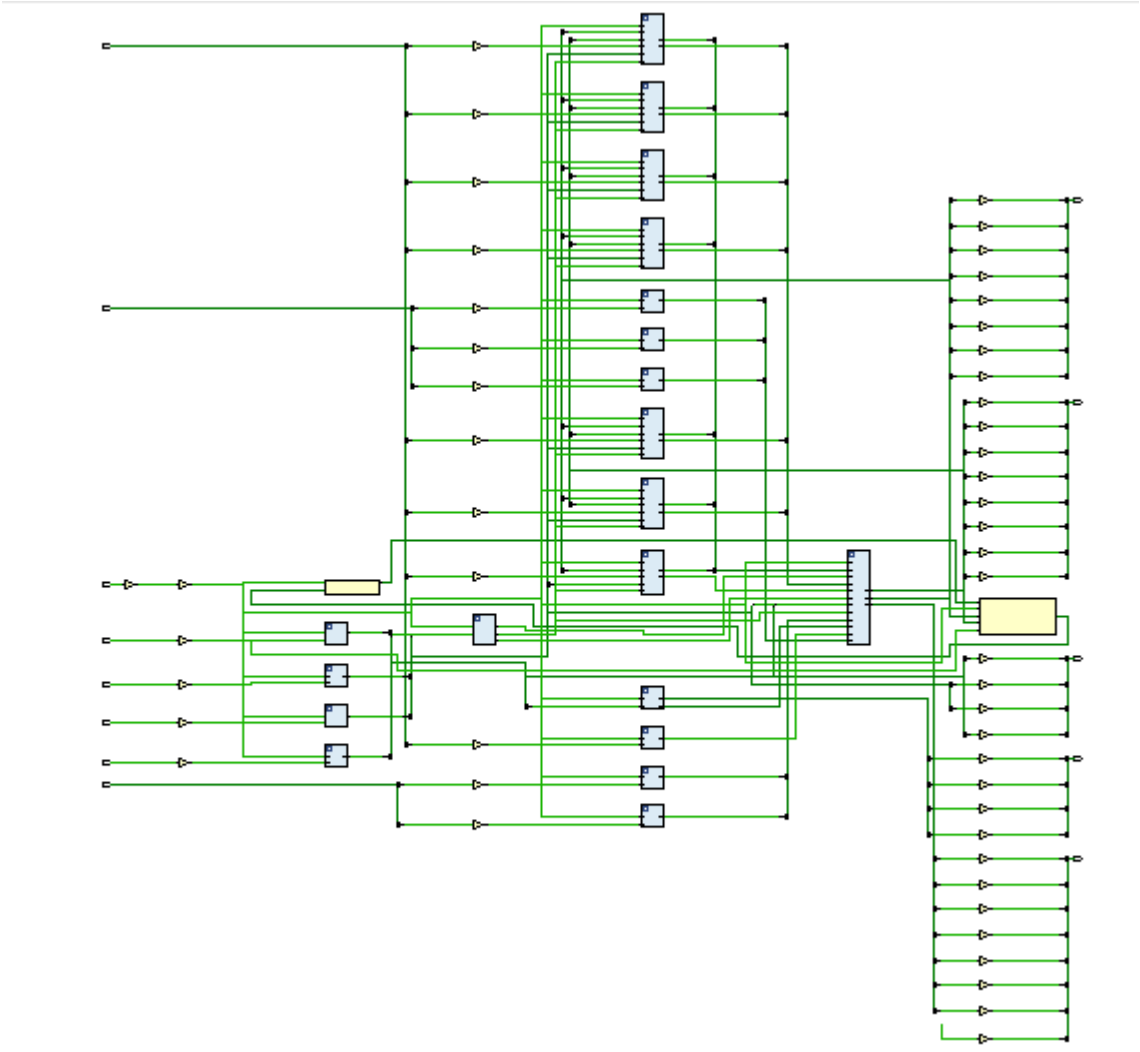


Figure 18: RTL block diagram from Vivado with Debug Cores

Vivado Debug Core

Vivado Debug Cores, part of Xilinx's Vivado Design Suite, are specialized tools designed for real-time monitoring and debugging of designs implemented on FPGA hardware, allowing for in-depth analysis and optimization directly on the device. Here are the steps to get it working:

Step 1) Under Synthesis, Set up Debug

Step 2) Choose Signals you want to see how it's changing and when. For example, A and B (inputs) and hex_seg (output)

Step 3) Set Sample Size

Step 4) Finish

Step 5) Ensure I/O Ports are assigned

Step 6) Run, and use inputs on the FPGA Board.

Step 7) Ensure you give enough time for the entire process to run, in our case 1,200,000 ns.

e. Include the output of the debug core trace performing an operation on the 8-bit logic processor (this doesn't need to be the same operation as the one your TA asked to demonstrate, but it needs to be non-trivial).

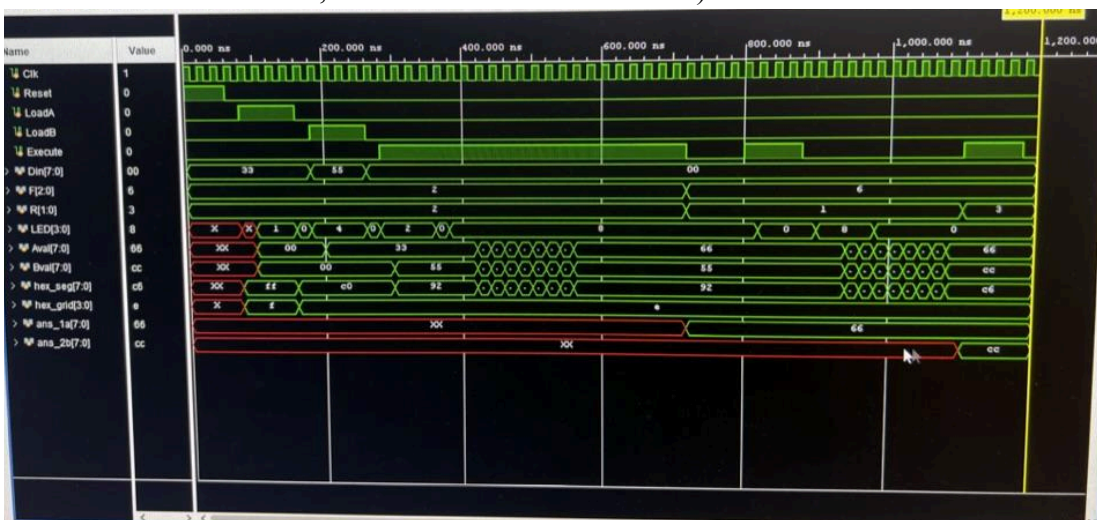


Figure 19: Simulation of Processor using Test Bench from Vivado and Debug Cores.

Annotations of Simulation of Processor:

The waveform analysis for the A_val, B_val, and hex_seg signals, reveals distinct operational phases of the bit-serial logic processor. From approximately 420ns to 580ns, there is a visualization of bits being right-shifted, indicating the processor's execution of serial operations. Subsequently, the period from 580ns to 840ns showcases the result of the first processing cycle, where the results become evident, providing insight into the processor's capability to execute logical operations and route the outcome appropriately.

This repeats from 840.00ns to 1100ns, during which right-shifting states are again seen, marking the second operational cycle. This repetition shows the processor's sequential processing. Finally, from 1100ns to 1200ns, the results of the second cycle are seen.

Description of all bugs encountered, and corrective measures taken.

During Lab 2.1, we encountered several challenges. Initially, we faced difficulties with a button that consistently dislodged from the breadboard. To resolve this, we opted for DIP switches as the execute button. Additionally, our unfamiliarity with the pull-down switch mechanism led to issues in powering the switches. This hurdle was overcome through further research and office hours.

In the development phase for the counter component, the concept of a "strobe" was initially unclear to us. Through dedicated research and more office hours, we were able to grasp its function, leading to the realization that the strobe should be grounded for correct operation.

The biggest mistake of them all was the misalignment of chips with their intended connections, resulting in incorrect wiring, such as connecting a chip intended for VCC to the Load A input, and similarly misaligning subsequent connections. We did not know what went wrong since we spent so much time planning before taking action, so we did not know what to do. After checking each wire connection and making sure that everything was placed correctly in regards to our schematic we realized our tiny mistake. This taught us the importance of attention to detail.

The question of when to incorporate resistors also presented a learning curve. Since we forgot basic ECE 110 knowledge, however, through experimentation we realized that you only need resistors when using buttons and switches.

To improve the organization and manageability of our setup, we introduced an additional breadboard dedicated to the switches. This not only streamlined our workspace but also enhanced the ease of use of our circuit.

Transitioning to Lab 2.2, our primary challenge shifted towards understanding the syntax and intricacies of SystemVerilog. Understanding the code, rather than merely adjusting numerical values, was important in extending our processor from 4 bits to 8 bits effectively.

Each of these challenges, while initially daunting, contributed significantly to our learning experience. Through debugging, research, and valuable office hours, we were able to navigate these obstacles and apply measures effectively, enriching our understanding and skill set in the process.

Conclusion

In this lab, our primary goal was to construct a bit-serial logic processor composed of four distinct modules: a register unit, a computation unit, a routing unit, and a control unit (Figure 5). This intricate system was designed to accept two 4-bit inputs and perform a comprehensive suite of basic logic operations, including AND, OR, XOR, as well as their inverse functions. Unique to our processor are the "1111" and "0000" functions, which output the result with either all 1's or all 0's, respectively. The operational behavior of the processor is determined by two principal inputs, F[3:0] and R[1:0], with the detailed logic and operational outcomes delineated in a truth table (Figure 3 and 4)**.** The next phase of the lab, Lab 2.2, involved scaling the processor design from 4 bits to 8 bits, utilizing SystemVerilog within Vivado. This expansion required an understanding of SystemVerilog syntax and a strategic approach to extend the processor's capabilities while maintaining its foundational logic functions.

Post Lab Questions

Q1) 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 capable of optionally inverting a signal is an XOR gate combined with a control input. In this configuration, one input acts as the data signal, while the other serves as the control signal to determine if the output should be the same as the input or its inverse. The mathematical representation of it would be: $Z = A \oplus B$

This is useful in the construction of this lab for multiple reasons, it allows for the processor to perform standard and inverse operations with a control signal making the system more versatile with less circuitry.

Additionally, the XOR gate allows for efficient routing and control which minimizes the need for more components and routing for inversion, for instance using another MUX would require much more complicated routing.

Lastly, this simplifies the state machine logic since you can directly manipulate the control signal.

Q2) Explain how a modular design such as that presented above improves testability and cuts down development time.

A modular design enhances the testability and development of complex systems, such as the bit-serial logic operation processor described in the lab. By dividing the processor into distinct, functional modules (register unit, computation unit, routing unit, and control unit) each component can be isolated for more straightforward testing and debugging. This isolation allows for targeted unit testing, where each module is individually verified against its own criteria, ensuring each component functions correctly before integration.

Moreover, the modularity allows for the creation of reusable test benches, which simulate operational scenarios for each module, simplifying the testing process and improving the system's reliability and performance.

Furthermore, modular design speeds up the development process by enabling parallel development workflows, where you can split up tasks with your lab partner, allowing you to complete the processor in a faster manner.

Lastly, the modular design creates a more robust, flexible, and maintainable system within a reasonable development cycle.

Q3) Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

A Mealy machine's output depends on both its current state and the input, making it more responsive to input changes. This property was crucial for our applications since we require immediate reaction to varying inputs, as it can potentially lead to a reduction in the number of states needed compared to a Moore machine. However, this responsiveness comes at the cost of potentially more complex logic circuits, as the output logic is intertwined with the input.

On the other hand, a Moore machine's output depends solely on its current state, not the input. This separation simplifies the design of the output logic but may require more states to achieve the same functionality as a Mealy machine, especially in systems where the output needs to change immediately in response to different input.

Q4) What are the differences between vSim and Vivado Debug Cores? Although both systems generate waveforms, what situations might vSim be preferred and where might debug cores be more appropriate?

vSim is a simulation tool that supports VHDL, Verilog, and SystemVerilog simulations, enabling designers to verify the correctness of their digital logic and circuit functionality in a virtual environment. This capability is crucial for identifying and rectifying logical errors before physical implementation, to ensure proper functionality. vSim is mostly used when a pre-synthesis simulation is necessary to ensure that the complex algorithms and interactions within the digital system operate as intended, free from the constraints of hardware.

Conversely, Vivado Debug Cores, integrated into the Xilinx Vivado Design Suite, are tailored for debugging and optimizing designs on FPGA. These tools, including the Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO), allow for real-time observation and debugging directly within the FPGA device. Vivado Debug Cores are used during the later stages of development, especially when real-world performance and hardware-specific, for example, timing constraints. Debug Cores provide insights by enabling designers to monitor and adjust the behavior of their circuits under actual operating conditions.

The selection between vSim and Vivado Debug Cores comes down to the development stage and specific needs: vSim for early-stage logic verification and simulation, and Vivado Debug Cores for in-depth, real-time debugging on hardware.