ECE 385

Spring 2024

Experiment 2

Logical Processor

Ashton Billings and Carson Conquest

Section AL1/M&W 4:00 – 4:50 p.m.

TA: Tianhao Yu

Introduction-

In Lab 2, we were tasked with designing a logical processor capable of performing eight bitwise operations on two input numbers A and B, outputting the resulting function F(A,B). The possible operations are as follows: AND, NAND, OR, NOR, XOR, and XNOR. The other two are simply outputting all 1s or all 0s irrespective of inputs A and B. The number of bits that make up A and B depend on which processor. The processor in Lab 2.1 had four bits per input, while the processor in Lab 2.2 had eight bits per input.

Operation-

In order to operate both of our Lab 2.1 and Lab 2.2 circuits, we need to do as follows: First load in the two numbers A and B into our processor, then determine which operation we wish to perform, next determine where we wish for our results to be stored after calculation, then finally executing our processor. The following section details just this.

To load data into the A and B registers, we first have to make sure we are in the reset state. This is simply done by making sure the Execute (E) signal is low. Once in this state, we could either switch the four switches corresponding to the D[3:0] bits or switch the eight switches corresponding to D[7:0] bits for either the 2.1 or 2.2 labs, respectively, specifying what value we wish to load into the destination register. Next, we flip either the switch corresponding to the LoadA input, or the switch corresponding to the LoadB input, or even both, to determine which register we wish to load. Upon the next clock cycle, the register should load in our desired value D[3:0]/D[7:0] according to these four/eight switches.

Assuming both values, A and B, are loaded and ready to be operated on, we are ready to decide our desired computation and routing operations. To set the correct computation, we simply flip the three switches controlling input signals F[2:0] according to Table 1 given in the Lab 2 document, or as shown below. The routing unit is similar, in that to decide where and what results are to get routed. To do this, we simply flip the switches controlling input signals R[1:0] according to the same Table 1 given below. After this, we are ready to execute our operation by setting our E signal to high. Note that the function, routing, and executing stages of operation are equivalent in both our Lab 2.1 or Lab 2.2 circuits.

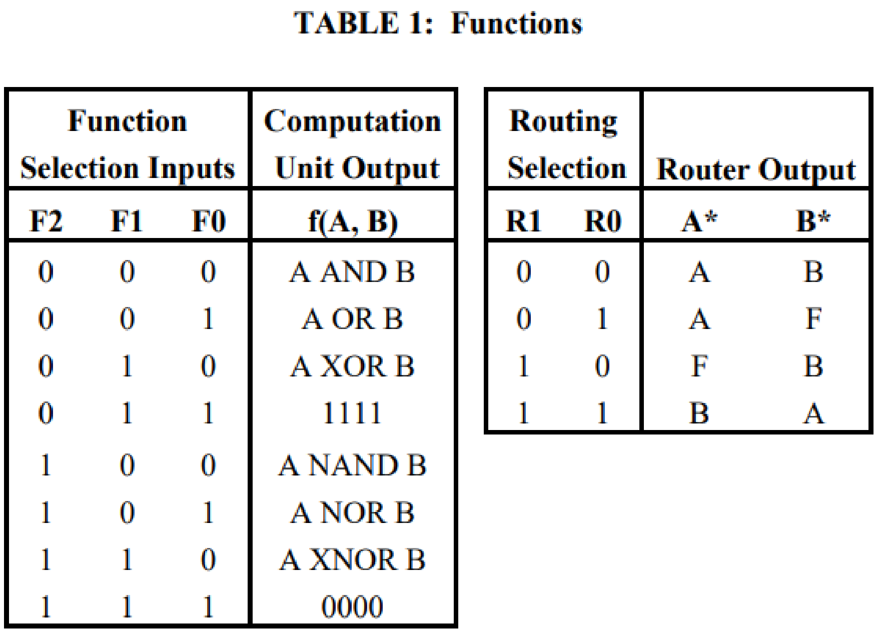


TABLE 1: Functions

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

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

Figure 1: Functions

Circuit Description-

We divided our logical processor into four primary units: control, register, computation, and routing. At a higher level, our circuit calculates our desired function F(A,B) bitwise from least to most significant bit. It does this by right shifting out the least significant bits stored in registers A and B, determining F(A,B) of this bit in the computation unit, then routing back to the most significant bit of the registers our desired values. A more specific explanation of each unit will be necessary to fully grasp how this works, so we will describe how each unit operates in this section.

<u>Computation Unit:</u>

The computation unit is responsible for performing logical operations on the contents of registers A and B, bitwise. It does this by taking in A and B, performing one of eight functions F(A, B), AND, OR, XOR, SET, NAND, NOR, XNOR, and CLEAR, then outputting F(A, B) along with unchanged inputs A and B. Below is shown the higher level block diagram of the Computation Unit.

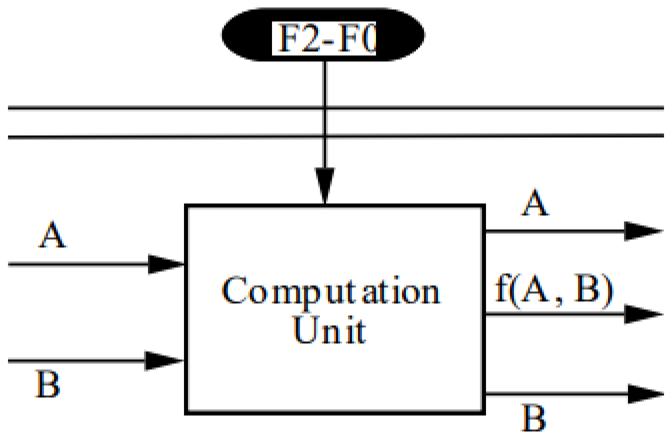


Figure 2: Computation Unit Block Diagram

Notice the input F[2:0]. This is how the user selects which of the four main functions they would like to perform, where the desired function can be chosen according to Table 1.

Routing Unit:

As stated, the results of the Computation Unit F(A, B) as well as the unchanged inputs A and B are the inputs to the Routing Unit. The routing unit also takes in external inputs R[1:0] to decide what to route back to the Register unit based on Table 1. From this, the routing unit simply takes in these inputs, and decides what outputs, A' and B', are. The block diagram for the routing unit is given below.

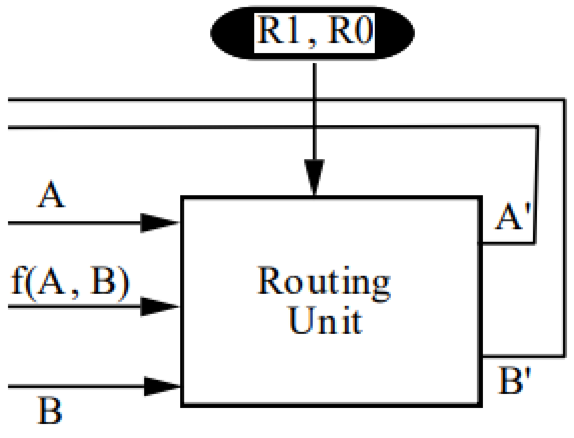


Figure 3: Routing Unit Block Diagram

Register Unit:

The Register Unit takes in four kinds of inputs. It takes in the new most significant bit A’ and B’ from the routing unit, the parallel load bits D[3:0]/D[7:0] for the Lab 2.1 and Lab 2.2 circuits, respectively, the control signal bits from the control unit, and the Clock signal. Its block Diagram is given below.

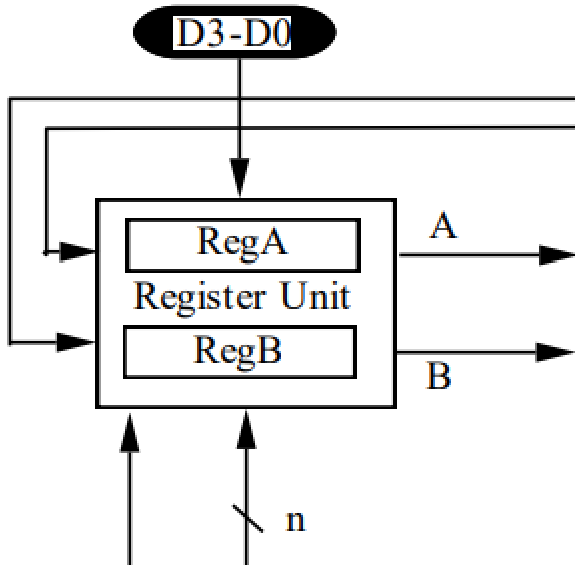


Figure 4: Register Unit Block Diagram

The Register Unit has three different functions: shift, hold, and parallel load. The one it does, and to which register (RegA or RegB), depends on the control signal inputs given by the control unit. If it is to shift, then the register unit will right shift out the least significant bit, A and B, and right shift in a new most significant bit, A’ and B’, upon the rising edge of the clock. If it is in parallel load mode, it will hold its current value, then replace its current value with the new value D[3:0]/D[7:0] upon the rising edge of the clock. If it is in hold mode, it will simply restore its current value for each rising clock edge irrespective of its other inputs.

Control Unit:

The Control Unit takes in external inputs by the user, as well as the clock signal, then converts it to a control signal output S fed to the Register Unit in order to perform the desired process. Its block diagram is seen below.

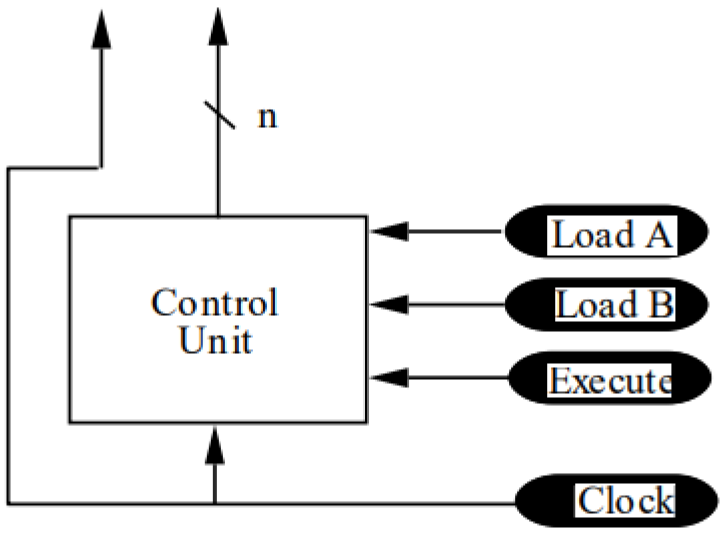


Figure 5: Control Unit Block Diagram

It does this by taking in inputs LoadA, LoadB, Execute (E), and Clock, then outputs signals to be input to the register unit based on both these external inputs, as well as its internal state. Its State Transition Diagram is given below, with labeled states and arc as to be explained.

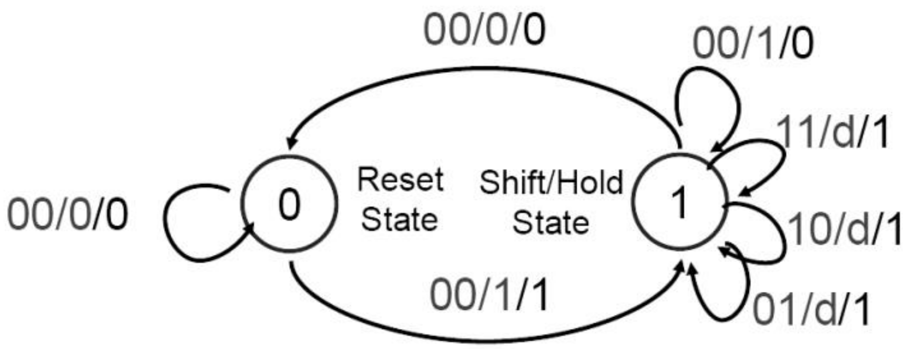


Figure 6: State Transition Diagram

The circuit contains internal logic bits Q, C1, and C0, each stored in its own binary flip flop. Q is our state bit, and determines whether we are in the reset state or shift/halt state, corresponding to the 0 and 1 bubbles in the state diagram. C1 and C0 are our internal counter logic keeping track of how many shifts have occurred; C1C0 counts to four in our Lab 2.1 circuit, while C1C0 counts to eight in our Lab 2.2 circuit. Since there isn’t one unique output per state, in which the output depends on both current state and inputs, this is a Mealy state machine.

The output S is determined by the arcs seen on the State Transition Diagram. The arcs follow the form C1C0/E/S. The C1C0/E specify what state we should transition too given those inputs. Note how these inputs are external for E and internal for C1C0. This internal nature of inputs C1C0 implies internal logic changes the value of C1C0 on its own, more on this in the design section. The /S section of the arc label specifies the output of the Control Unit, to be fed to the Register Unit.

The Control Unit starts with 00/0/0. Once execute is run, it transitions to 00/1/1, and count iterates four times, outputting S = 1 four times (these are the shift pseudo-states of Q = 1) before settling on 00/1/0 (the hold pseudo-state of Q = 1). Once E = 0, it returns to the Reset State Q = 0, and the process can begin again.

Notice how as of now, the inputs to the Control Unit LoadA and LoadB have yet to be mentioned. This is because there is actually one more layer of logic necessary before signals can be outputted to the Register Unit. This logic takes in S, LoadA, and LoadB and does the following: If S = 0 don't shift, and if LoadA or LoadB are high, not only not shift, but also parallel load RegA and RegB. This logic creates the correct output signals to the Register Unit accordingly. If S = 1, ignore LoadA and LoadB and create the correct output signals such that the Register Unit shifts.

Design/Design Process-

The general approach in our design process is as follows: First have a good understanding of the CMOS chips we would be able to work with. Next, break down each unit from its block diagram into individual circuit components corresponding to circuit elements available to us in the form of CMOS chips. Finally, we plan out the wiring we must do in order to implement the unit with the given CMOS chips. After doing this with each unit, we simply then needed to attach each units inputs and outputs together, giving us our final result.

<u>Computation Unit:</u>

Given the [153], [00] , [02], and [XOR], we knew to breakdown the Computation Unit to 4:1 MUXs for the function select, and into NAND, NOR, and XOR logic for the function creation. For this unit, we decided to stick to NAND logic exclusively, except for use of XOR gates. Why we used XOR gates instead of all NAND gates will be explained later.

For XOR, we will eventually implement it with the [XOR] chip, and SET can be created by simply wiring its select input to voltage high. We can invert these functions by passing it through a NOT gate, however, this was not considered in the end as will be explained in the next section.

At first, the fact that eight operations needed to be chosen from implied that an 8:1 MUX was needed, which could be made out of two 4:1 MUXs. However, upon view of Table 1, we see that a version using just one 4:1 MUX was possible. Notice how the last four functions are simply the inverted version of the first four. Also notice that the last four functions are only different from their first four counterparts in that F[2] = 1, rather than 0. From this, we could come up with the following circuit scheme: Take a 4:1 MUX and have its select signals be F[1:0], with its inputs corresponding to the first four corresponding functions. Then we can simply have the output be inverted if F[2] = 1. This can be done through the bitmask property of the XOR gate. The bitmask property of the XOR gate is such that, taking one input as a sort of “select”, we can invert the other input if our “select” signal is high, or pass the input through unchanged if our “select” is low. In our case, by simply making F[2] our “select” signal, we can successfully design a circuit such that F[1:0] decides what operation we wish to perform, while F[2] decides whether it's the inverted version of this operation or not. This wiring scheme also means that even if the user doesn’t understand the actual purpose of F[2], it won’t effect operation. Our final design using one 4:1 MUX, NAND gates, and XOR gates is shown below.

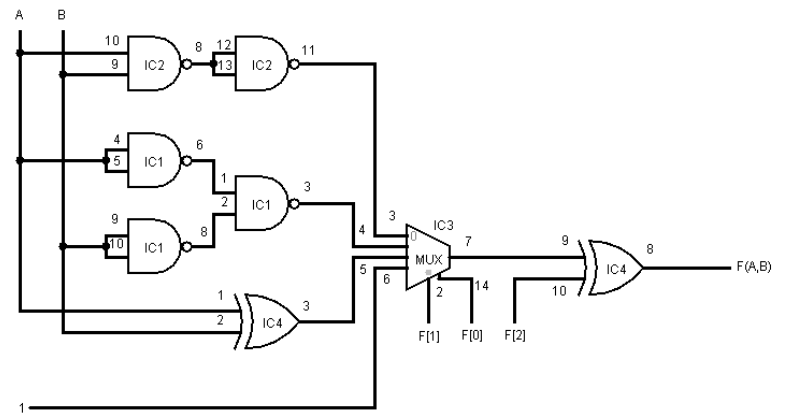


Figure 7: Computation Unit Circuit Diagram

<u>Routing Unit:</u>

Given the [153] Chip, we decided to break down the routing unit into two 4:1 MUXs; one being wired back into the most significant bit of RegA, A', and one being wired back into the most significant bit of RegB, B'. As stated already, the inputs to the routing unit as the least significant bits of RegA, A, and RegB, B, and the result of the logical function F(A,B), F, produced by the Computation Unit.

Initially, we assumed that along with the two 4:1 MUXs, we also needed extra logic to determine the inputs to these MUXs. However, we realized that by simply ordering the inputs to these muxes corresponding to Table 1, we could directly and simultaneously have the MUXs correctly pass through our desired input given input signals R[1:0]. Our circuit diagram of such is shown below.

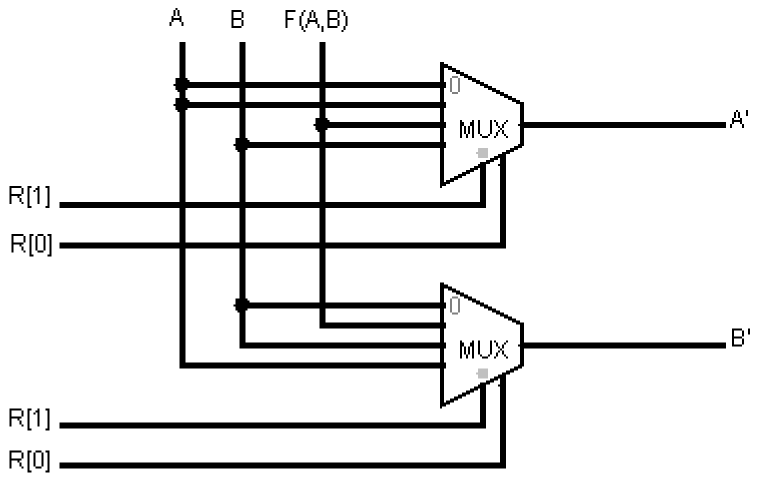


Figure 8: Routing Unit Circuit Diagram

Register Unit:

Given the [193] chip, it was clear we needed to break down the register unit down into two 4-bit registers. However, we broke it down even further into individual D Flip-Flops and 3:1 MUXs for clarity, despite 3:1 MUXs not being necessary in the implementation. This will be commented on more in the Implementation section.

The register unit has the following functionality: parallel load, hold, and right shift. These functions are indirectly controlled by the user through inputs from the Control Units signal outputs. We can think of each 4-bit register as four/eight rising edge triggered D flip-flops, with each flip-flops data input P coming from the output of a 3:1 MUX. The MUXs allow the control unit to select what it would like the input P of each register to be. The three inputs to these MUXs are the parallel load bits, the output Q from the previous flip-flop (or the new value from the routing unit if it's the most significant bit), and the original value stored in the individual flip-flop. These three inputs correspond to our desired ability of the 4-bit register unit to either parallel load in a new value of A and B, right shift the bits as the new bits A' and B' are shifted in during computation, or hold our current value for before/after computation. It performs the desired type of load at the rising edge of the clock input signal. The 2-bit signal inputs to each MUX are the output signals from the Control Unit, which decides which of these three functions we wish for the register unit to perform. Below is our final circuit diagram.

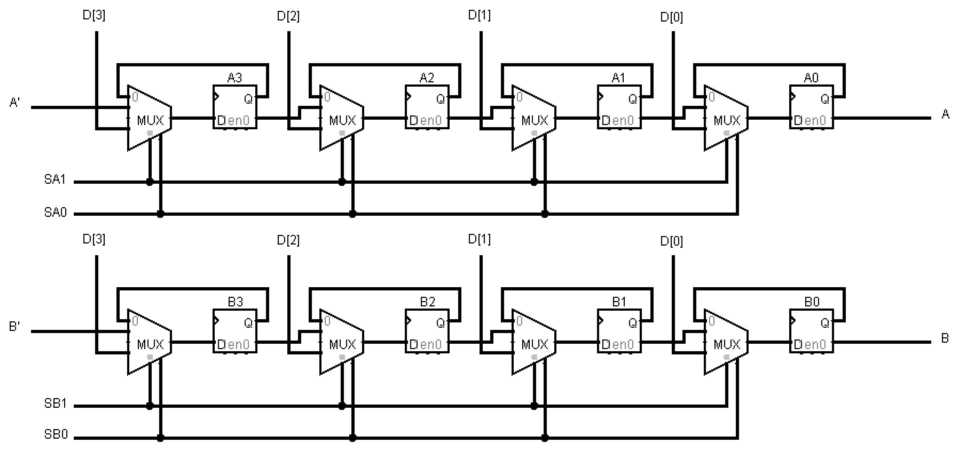


Figure 9: Register Unit Circuit Diagram

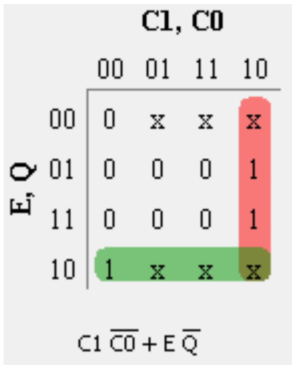
<u>Control Unit:</u>

The control unit was by far the most complex unit out of the four. As stated, we based our design off the Mealy machine given in the Lab 2 document.

From this, we got to work breaking down the control unit into its component circuit elements. Given the [193], [00], and [NOR] chips available to us, we broke the Control Unit down to registers and NAND or NOR logic. Like the Computation Unit, we stuck to NAND logic for our implementation. In specific, we broke the Control Unit down first into three sub-sections: registers holding the state/count bits, NAND next state/count logic, and NAND shift/load logic.

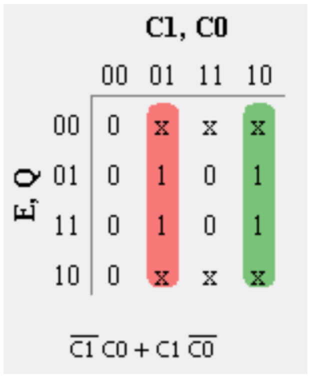
Despite the [194] chip being two 4-bit shift registers, if wired appropriately, we could use each D Flip-Flop standalone. With this, we are able to correctly appropriate three D Flip-Flops to be used for Q^+ , $C1^+$, and $C0^+$ which is why we left the three as such in our final circuit diagram, as will be seen.

Next, we need to implement our next state/count logic. As stated above, we decided to explicitly use NAND chips. Below are the K-Maps and their solutions to the corresponding next state/count logic for Q^+ , $C1^+$, and $C0^+$. Note how this next state logic is determined by both external inputs Execute (E), as well as previous state/count internal logic Q, C1, and C0.



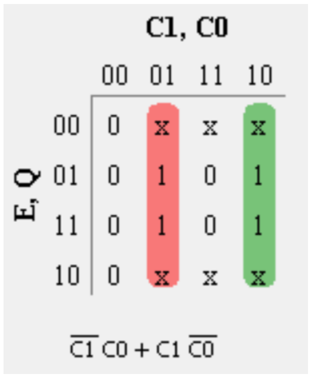
E, Q \ C1, C0	00	01	11	10
00	0	x	x	x
01	0	0	0	1
11	0	0	0	1
10	1	x	x	x

C0+



E, Q \ C1, C0	00	01	11	10
00	0	x	x	x
01	0	1	0	1
11	0	1	0	1
10	0	x	x	x

C1+

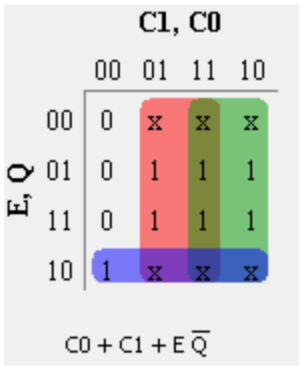


E, Q \ C1, C0	00	01	11	10
00	0	x	x	x
01	0	1	0	1
11	0	1	0	1
10	0	x	x	x

Q+

Figure 10: Next State Logic K-Maps

Finally, we designed the shift/load logic circuit. This circuit can be designed in many ways. We went for a two-layer approach, first producing a general Shift signal S, then taking S along with LoadA and LoadB and producing four output signals. These outputs are the inputs shown above as the inputs to the MUXs in the Register Unit that is used to decide what we wish for the Register Unit to do that clock cycle. The K-Map for S is given below, along with the K-Maps for the new signals S_{A1}, S_{A0}, S_{B1}, and S_{B0}.



E, Q \ C1, C0	00	01	11	10
00	0	x	x	x
01	0	1	1	1
11	0	1	1	1
10	1	x	x	x

Figure 11: Shift S Signal K-Map

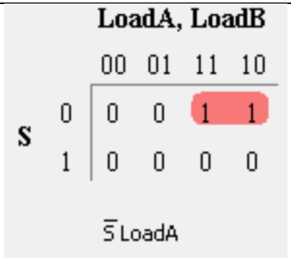
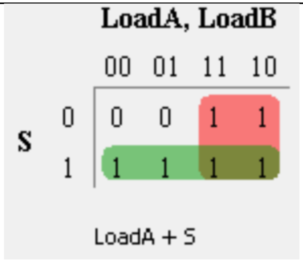
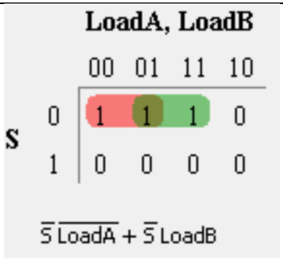
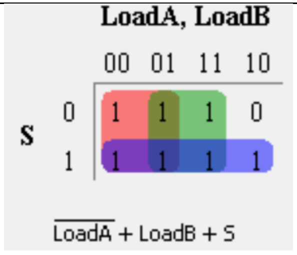
 SA1	 SA0	 LoadA, LoadB S $\bar{S}\,\overline{\text{LoadA}} + \bar{S}$ LoadB SB1	 SB0

Figure 12: Register Unit Signal Pin K-Maps

Finally, we can take our three registers, next state/count logic, and shift/load logic and create the following circuit diagram shown below. Again, note that this is NAND logic as we plan to use the [00] chip in implementation.

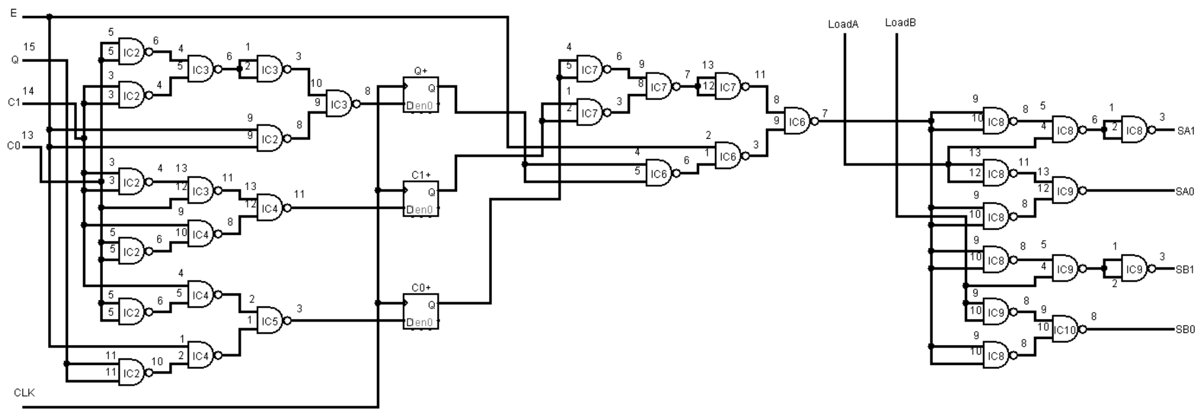


Figure 13: Control Unit Circuit Diagram

Breadboard Implementation-

With our circuit designs out of the way. We can finally go about implementing our design. We will first introduce the chips used for each unit, and provide their pin layouts for clarity. Then our final implementation of that chip will be given.

<u>Computation Unit:</u>

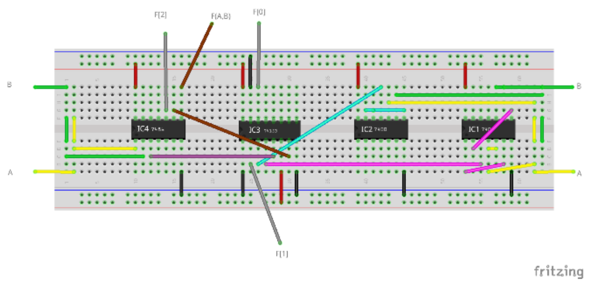


Figure 14: Computation Unit Implementation

<u>Routing Unit:</u>

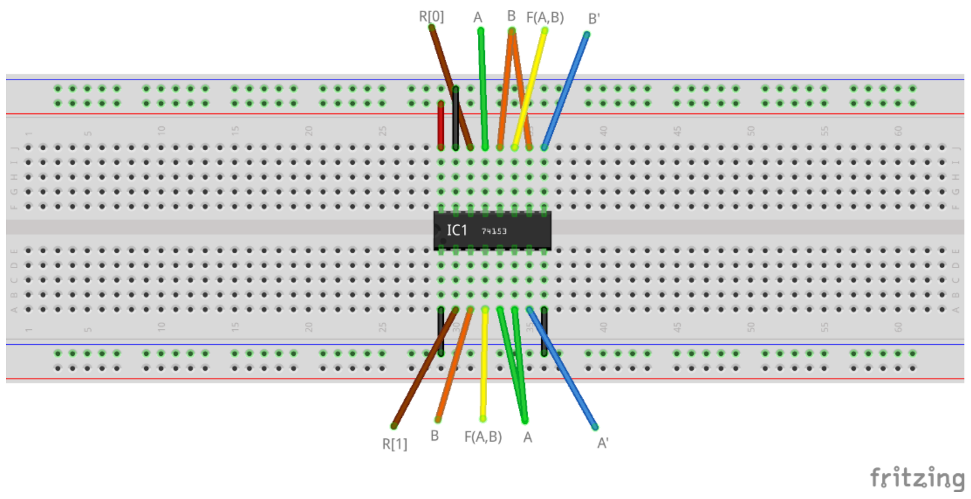


Figure 15: Routing Unit

Register Unit:

Despite the use of 3:1 MUXs in the circuit diagram, the 194 chip already comes with the three functions desired: Hold, Right Shift, and Parallel Load. As such, only two 194 chip were needed, one for RegA and one for RegB.

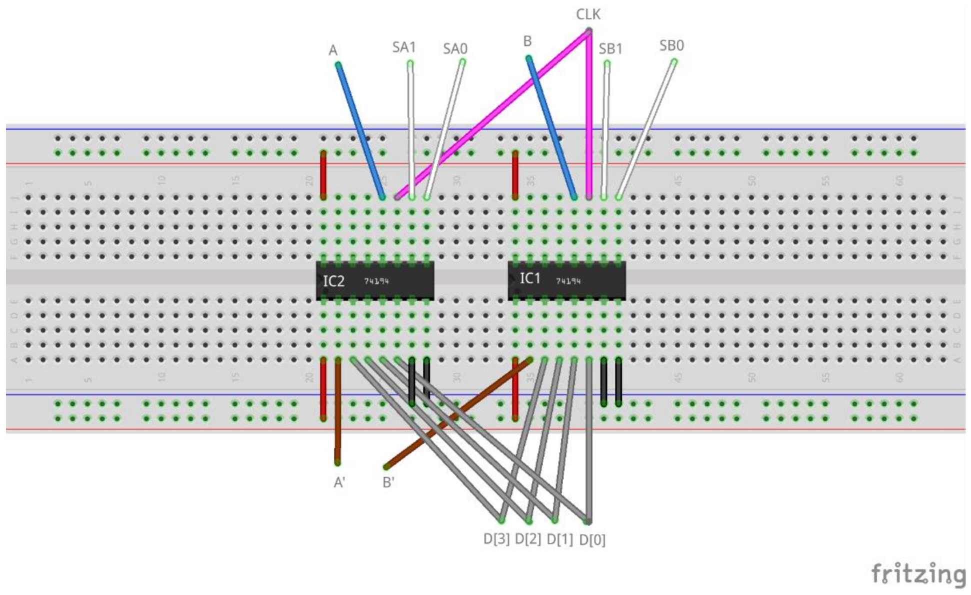


Figure 16: Register Unit Implementation

Control Unit:

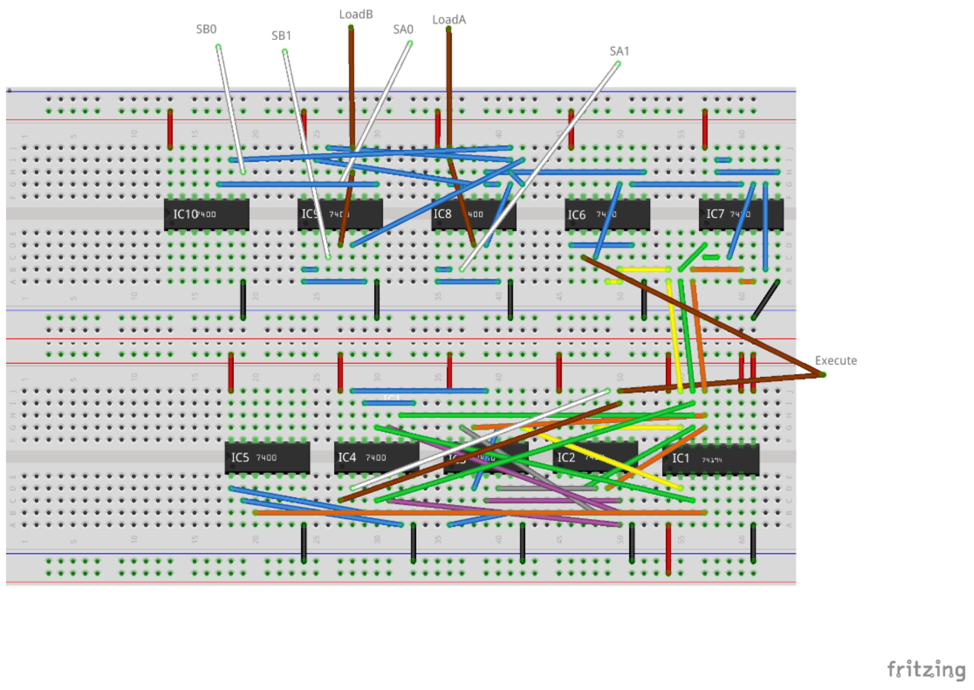


Figure 17: Control Unit

8-bit Logical Processor on FPGA

a. **.sv file summaries:**

   a. 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 is a positivie edge triggered 8-bit parallel load register with asynchronous reset and synchronous load. When Load is high, data is loaded from Din into the register on the positive edge of Clk.

Purpose: We used this module to store the values we would be performing logical operations on and to store the results of said operations.

Module: compute.sv

Inputs: [2:0] F, A_In, B_in

Outputs: A_out, B_out, F_A_B

Description: This is a combinational logic unit that has 4 logical functions that we select between on a 4:1 mux using F[1:0] and then we select whether to invert using F[2].

Purpose: We use this module to perform logical functions on A and B.

Module: Router.sv
Inputs: [1:0] R, A_In, B_in, F_A_B
Outputs: A_out, B_out
Description: This is a dual 4:1 mux module that selects what value is routed back to registers A and B using R[1:0]. The possible values are A, B or F_A_B.
Purpose: We use this module to choose which value we want each register to display after a computation has been performed.

Module: Control.sv
Inputs: Clk, Reset, LoadA, LoadB, Execute
Outputs: Shift_En, Ld_A, Ld_B
Description: The control unit is a Mealy FSM moving between rest, hold and counting states. It begins at the rest state then upon execute, moves through eight counting states in which is outputs Shift_En telling the register unit to shift right. After it shifts eight times, cycling through its eight counting states, it stays in the hold state until execute is low.
Purpose: We use this module to control the selector bits that determine the function we perform and the location of the outputs.

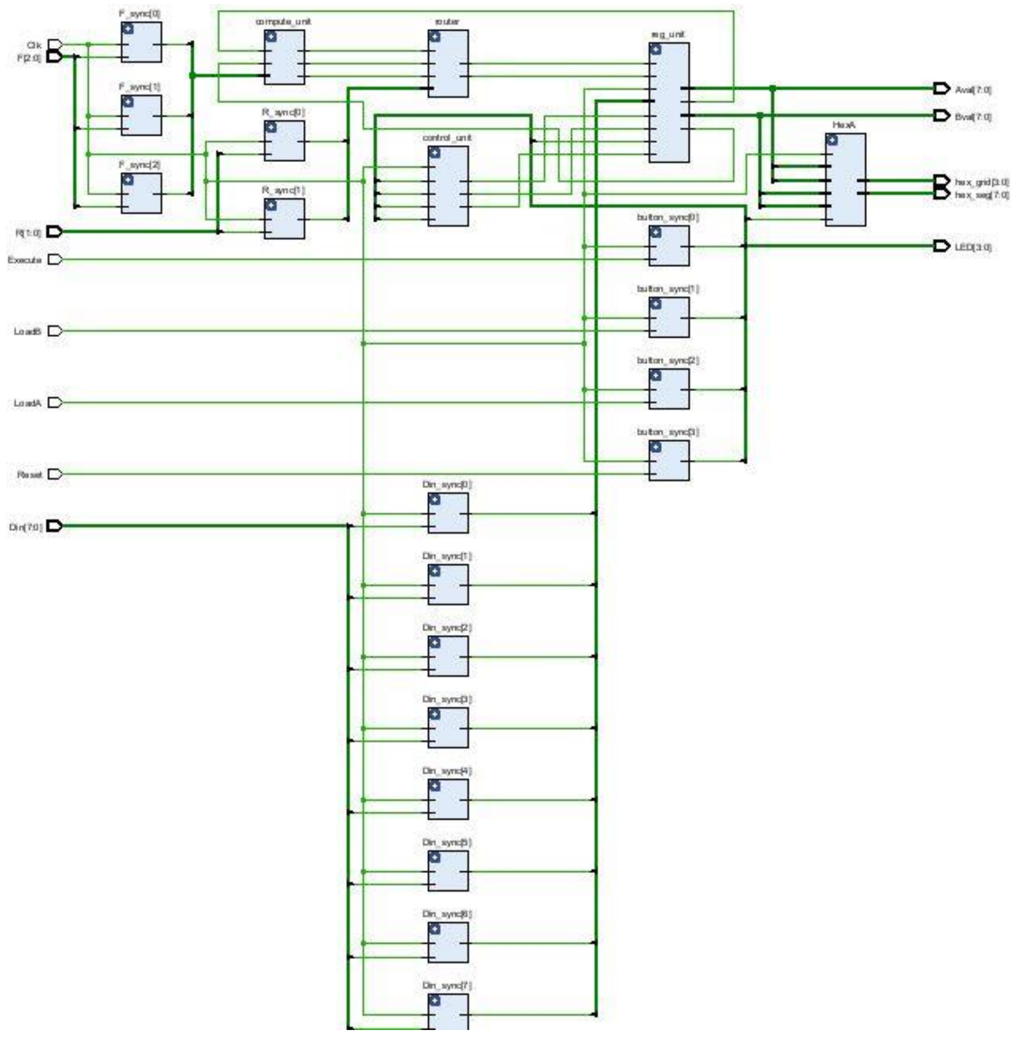
Module: HexDriver.sv
Inputs: clk, reset, [3:0] in
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: A binary to hex converter that takes 4 bits of binary as input and converts those bits to a hex bit.
Purpose: We use this module to convert our binary inputs to hex digits on the FPGA.

Module: Synchronizers.sv
Inputs: Clk, d

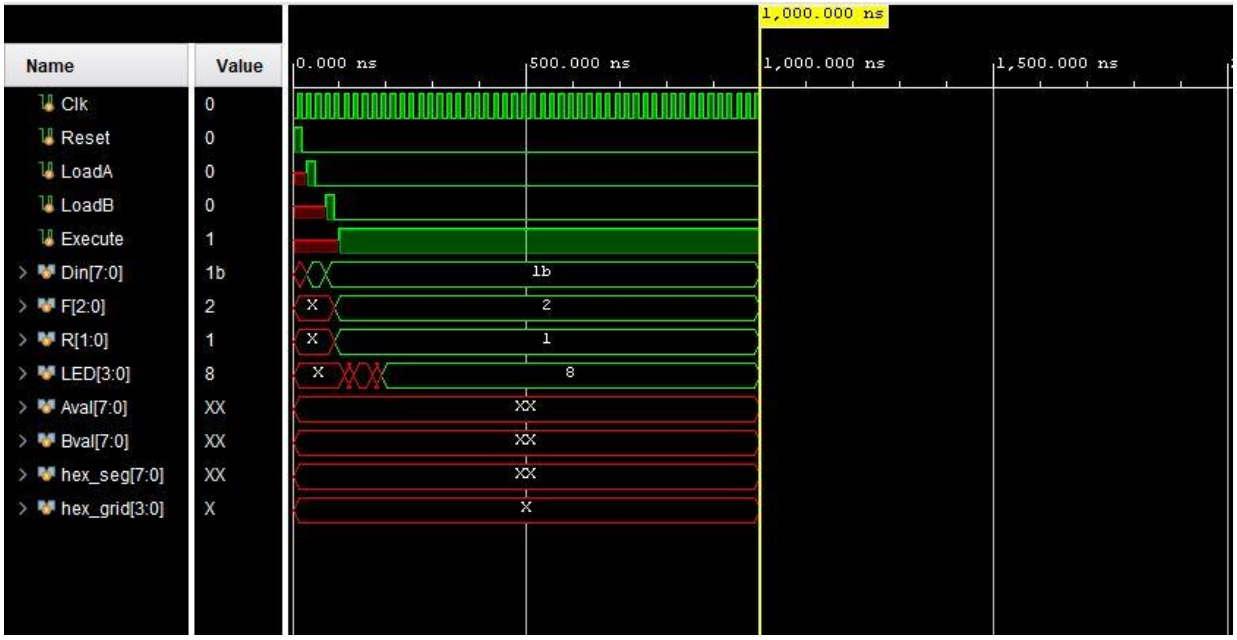
Outputs: q
Description: A sequential logic module that using non-blocking assignments to synchronize the flip-flops.
Purpose: We use this module to implement a debouncer circuit inside the processor switches.

   b. Here we will detail all the changes we made to the provided .sv files to extend the functionality of the logical processor from 4 bits to 8 bits. The first change we made was replacing the values of 3 in the register unit to 7. In other words, [3:0] became [7:0] for both D and Q. No changes were made to either the computation unit or the routing unit, as their functionality was never dependent on the bit width of A and B; there were also no changes added to the control unit.

b. RTL diagram pictured below: (Note that the values of A and B are x6a and x1b respectively; the computation performed, and the destination are XOR and register B respectively.)



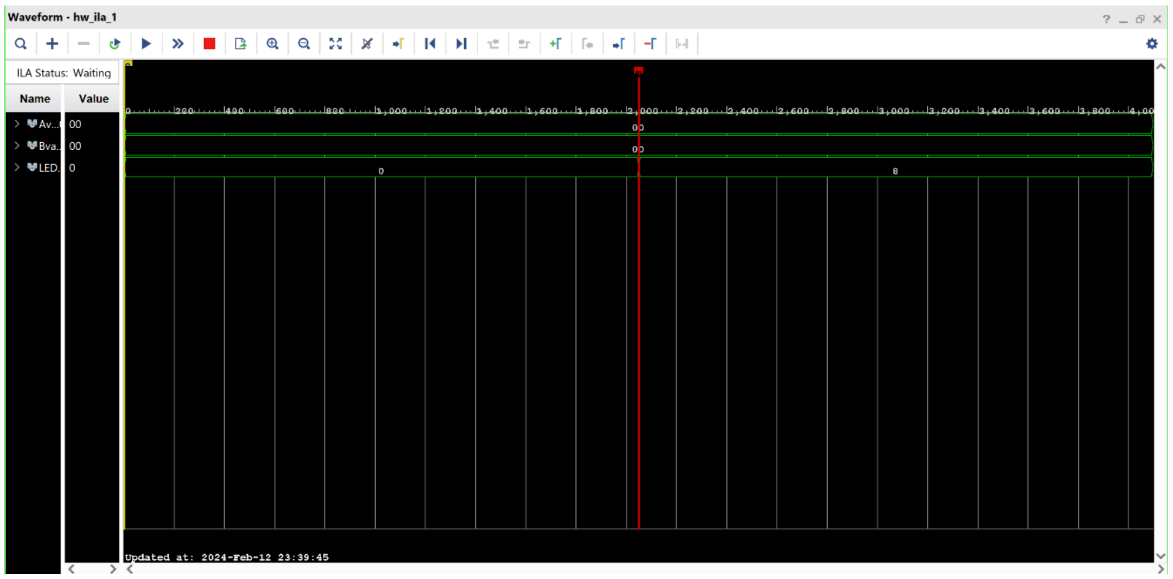
c. Simulation of the processor pictured below:



d. **Procedure used to generate Vivado Debug Core trace:**

1. Set the trigger on signals Aval_OBUF[4:0], Bval_OBUF[4:0], and LED_OBUF[3:0].
2. Change the number of samples captured to 4096 to reduce time needed to run bitstream.
3. Set the conditions for the triggers to be on the rising edge of the most significant bits of each.

e. Pictured below is the Vivado Debug Core setup using the procedure above:



```
Updated at: 2024-Feb-12 23:39:45
```

Description of All Bugs Encountered / Corrective Measures

By far our biggest bugs in lab 2 stemmed from our uncertainty of the nature of the 4-bit parallel register chip. According to the truth table provided in the datasheet of the Texas Instruments SN74194, a rightward shift would pop the value of Q[3] and shift the least significant bit (LSB) of the new 4-bit value into Q[0]. This appeared to be the opposite of the procedure we desired in our design. We therefore wired the relevant routing output to the pin labeled DSL (shift left.) This resulted in a rather frustrating bug wherein regardless of the values A and B or the operation performed, the registers were shifting in all 1's. Upon closer inspection, we deduced that the 'all 1's' bug originated from the DSR (shift right) pin not having any wired connection. We hard wired DSR to ground and re-ran the test, only to observe that the registers were being filled with all 0's now. We surmised that DSR was in fact the correct pin to have the routing unit output to, and as such we swapped the wiring from DSL into DSR. We then hard wired DSL to ground for good measure and observed that the registers now displayed the correct values based on the function performed on the previous values.

Conclusion

a. Lab 2 was a lab that required a great deal of effort for multiple reasons. While the high-level block diagram was given to us in the lab document, we had to derive the entirety of the logic and physical structure of the circuit ourselves. This lab also clearly illustrated for us the advantages of implementing a circuit on an FPGA as opposed to CMOS chip implementation. Furthermore, the structure of the lab allowed us to learn SystemVerilog code more effectively by implementing a circuit whose block structure we were already familiar with.

b. **Answers to Post Lab Questions:**

1. Describe the simplest (two-input one-output) circuit that can optionally invert a signal (i.e., one input determines if the output is equal to the other input or equal to the other input inverted). Explain why this is useful for the construction of this lab.

The simplest way to implement an optional inversion of a signal is to wire the output of that signal to an XOR gate along with an inverting

signal. For example, if the signal we wanted to invert was the result of an AND function, we would XOR that signal with logical 1 to invert it. This is known as bit masking, and it proved to be a useful concept in lab 2, as it allows us to implement just four of the functions out of NAND gates instead of all eight functions. This reduced the space on the breadboard taken up by the computation unit in lab 2.1 and allowed us to achieve full circuit functionality with fewer lines of code in lab 2.2.

2. Explain how a modular design such as that presented above improves testability and cuts down development time.
A modular design is useful for development and debugging because it allows us to recycle previously written modules. Furthermore, the recycling of prior modules eliminates the need for constant debugging, as we would only have to find as many bugs as there are unique modules. Development time is also cut down significantly, as the hierarchal structure allows us to use more module instantiations as opposed to independent module declarations.

3. Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?
The primary argument governing whether to use a Mealy or Moore state machine boils down to operating efficiency versus design complexity. When designing the state machine for our logical processor, we opted to use a Mealy machine. The driving force behind this decision was the requirement for the execute signal. We needed the processor to change states based on the logical value of execute, and therefore we needed to base our design on a Mealy machine. This allowed us use only two states (greater operating efficiency) but required more complex output logic definitions (greater design complexity.)

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

The most prominent difference between vSim and Vivado Debug Cores is the level of abstraction in which each operates. VSim would be a preferable tool to use when the thing we wish to test is the overall functionality of the ciruit, whereas Vivado Debug Cores would be more useful for testing lower-level hardware implementations. For example, if we wanted to test the behavior of the circuit in its entirety, we would run a behavioral simulation using a module of written code called a testbench to simulate inputs and set clock cycles. This would allow us to see exactly what each piece of the circuit is outputting after an execution sequence finishes. In contrast, if we wanted to keep track of a specific signal or group of signals, and we wanted to manually control the inputs being fed to the circuit, we would implement a Vivado Debug Core. This tool would be useful in debugging localized functionality issues as it allows us to analyze circuitry on a lower level than simulation.