Design and Implementation of a Simple Sequential Processor

Naren Kolli
kolli.na@northeastern.edu

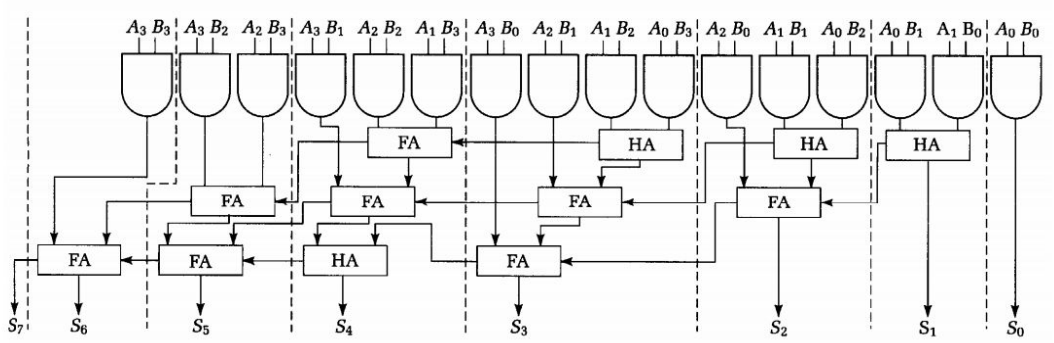
Abstract

The purpose of this project is to design a simple sequential processor utilizing an arithmetic-logic unit (ALU) that can perform various arithmetic operations. The ALU will support four different operations: addition, multiplication, subtraction, and bitwise AND. For simulation and verification, the functionality will be tested sequentially on consecutive clock cycles.

Introduction

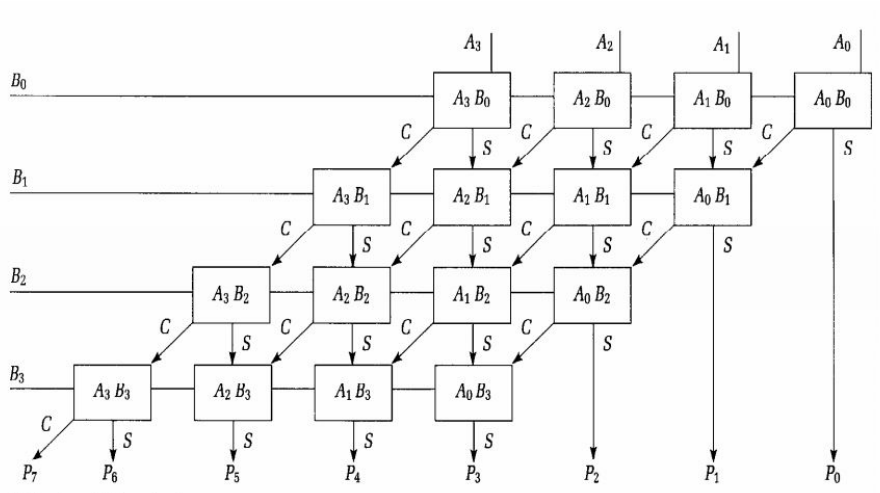
An ALU is the root concept that all successive processing units follow. It is sequential, meaning that the order of events is a necessary variable in computation which further means it requires syncing to a standardized clock. In order to support all four mathematical operations, the development of subsystems are required for implementation: an 8-bit adder, 8-bit subtractor, 8-bit multiplier, 8-bit bitwise "AND" operator, 4-to-1 multiplexer, and an 8-bit register.

In the course of multiplying two binary numbers, each bit in the multiplier is multiplied with the multiplicand. Each of the four products is aligned (shifted left) according to the position of the bit in the multiplier that is being multiplied with the multiplicand. The four resulting products are added to form the final product. This leads to one of two methods of implementation in Simulink:



"A slightly different implementation scheme is shown in the figure below. This multiplier is constructed from an array of building blocks, shown in the first figure below. Each building block consists of an AND gate for computing locally the corresponding partial product (X gY), an input passed into the block from above (Sum In), and a carry (Cin) passed from a block diagonally above. It generates a carry out bit (COUT) and a new sum out bit (Sum Out). The second figure illustrates the interconnection of these building blocks to construct a 4x4 combinational multiplier. The Ai values are distributed along block diagonals (look at the products to see the correspondence), and the Bi values are passed along the rows."

The multiplexer, shortened to "MUX" or "MPX", is a combinational logic circuit designed to switch one of several input lines through to a single common output line by the application of a control signal.



(b) 4 × 4 multiplier structure

Simulink Discussion

The hardware designs were constructed in MATLAB Simulink 2019a. With additional preparation done by-hand on paper. Settings used:

- Type: Fixed-step
- Solver: Discrete (no continuous states)
- Fixed-step size: auto

Simulink is developed by the MathWorks and is tightly integrated with MATLAB. Simulink provides a rich GUI for modeling, simulating and analyzing dynamic systems. Simulink enables rapid construction of virtual prototypes to explore design concepts at any level of detail with minimal effort.

The zeroth step for designing with Simulink (as with other tools) is to develop a design specification; fully specifying the system and its requirements. The specification covers what the input ranges and types are, what is expected as output and how the system is supposed to work. After you specify these high-level details of a system, the next step is modeling, where Simulink helps you visualize and debug your design.

Modeling is a process that enables faster and more cost-effective development of dynamic systems. This is especially helpful for systems with nondeterministic behavior. To build and understand a system before mapping the design to a real implementation, we would like to simulate the system in software. When we model our design in Simulink, we can model and test the behavior of our design with test inputs. After completing design verification, the design can be iteratively refined to finally produce the desired behavior.

All of these steps, which include modeling, testing, verification and refinement, can be done in Simulink. In this tutorial we will go through these steps as we build our knowledge of designing systems with digital logic.

Results and Analysis

Initially, an 8-bit adder circuit was created to add two 8-bit positive numbers. When creating this circuit, it was assumed that the sum would be less than 255. The circuit design can be seen in Figure 1.

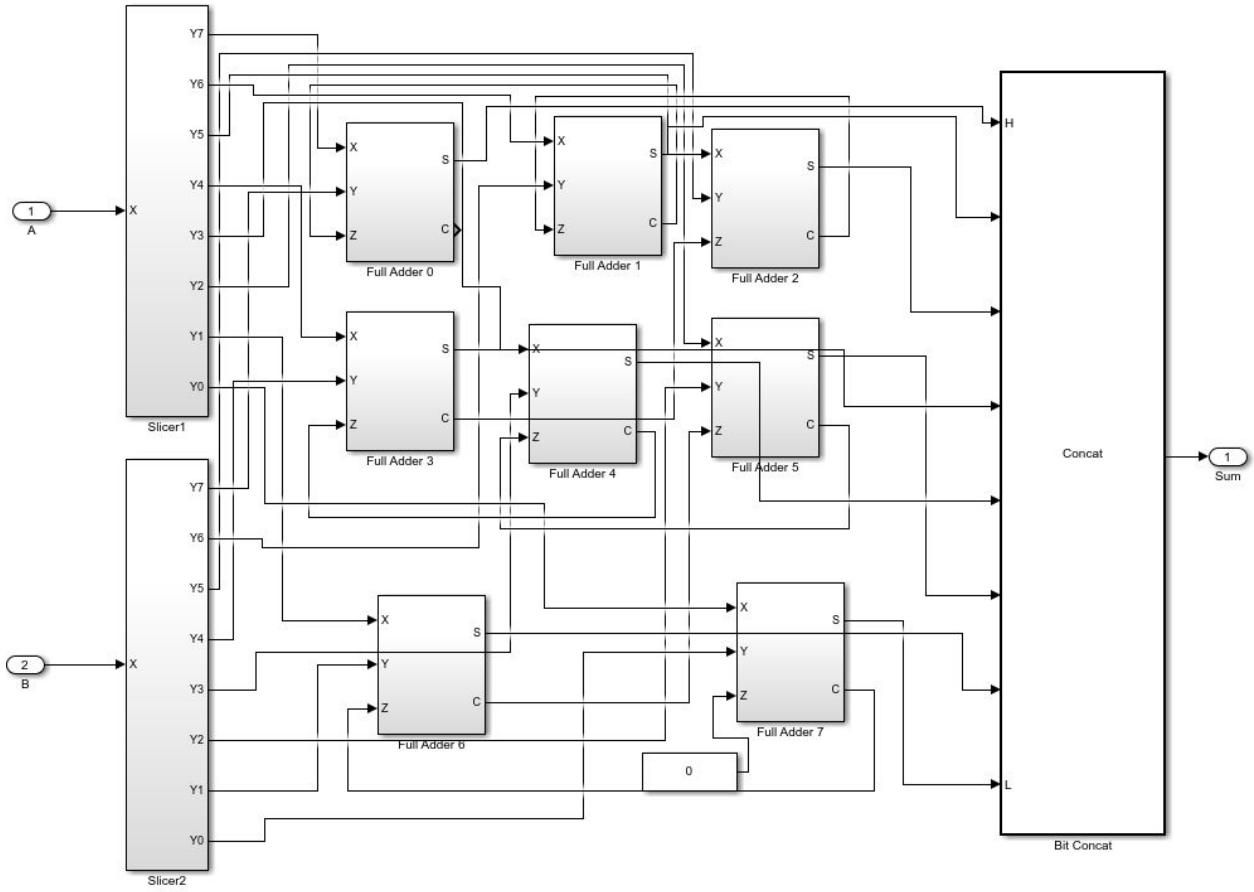


Figure 1: Adder Subsystem

In order to add the two 8-bit numbers, the two numbers were converted to 8-bit binary format using a bit-slicer Simulink block. Then, eight full adder subsystems were utilized to add the two 8-bit numbers together; during this time, the last carry bit was ignored. Then, the sum of the 8-bit numbers was concancated and converted to its decimal form for output. Figures 2 and 3 show sample simulations of the 8-bit adder circuit.

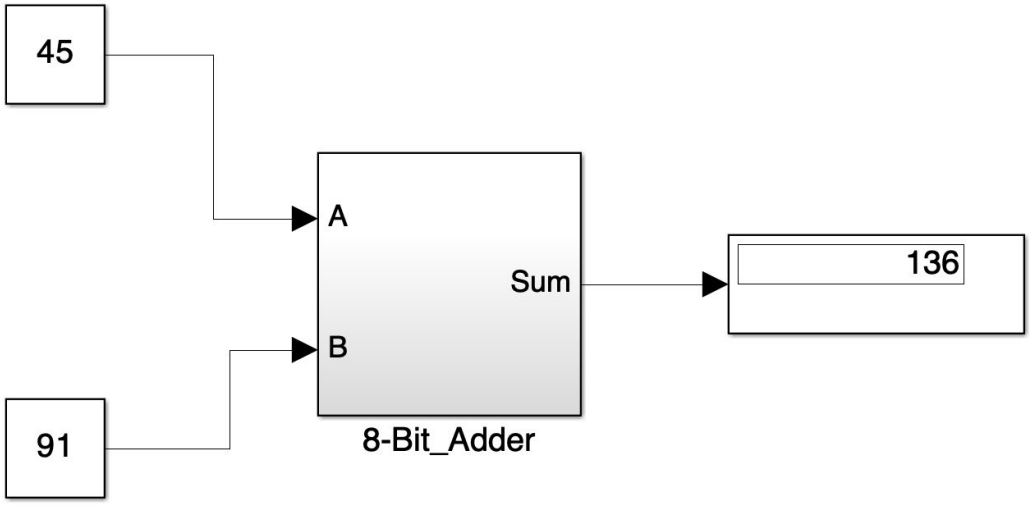


Figure 2: Adder Sample Simulation

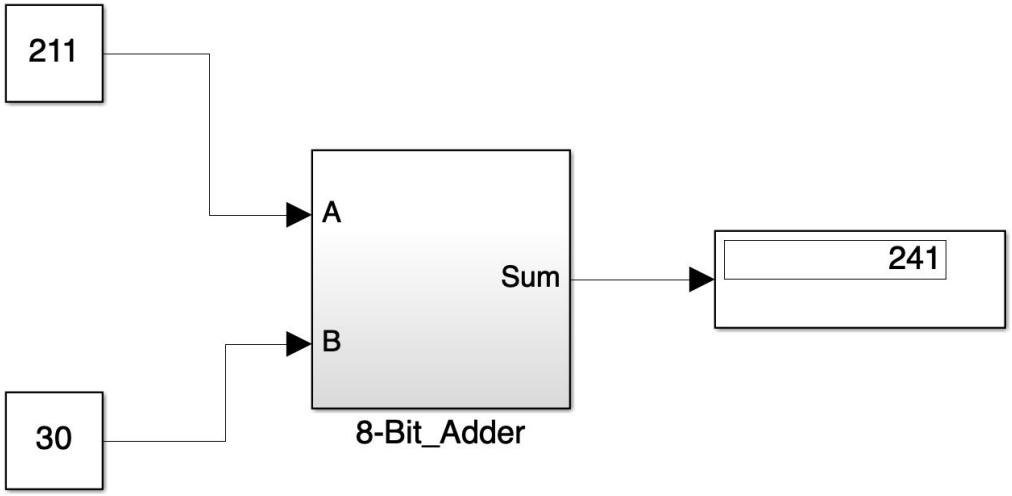


Figure 3: Adder Sample Simulation

Looking at both sample simulations, it can be seen how both simulations produce the correct output when both inputs are added together. 45+91 does equal 136 and 211+30 does equal 241.

Next, an 8-bit subtractor circuit was designed to subtract two unsigned 8-bit integers. The subtractor circuit is very similar to the 8-bit adder circuit and its design can be seen in Figure 4.

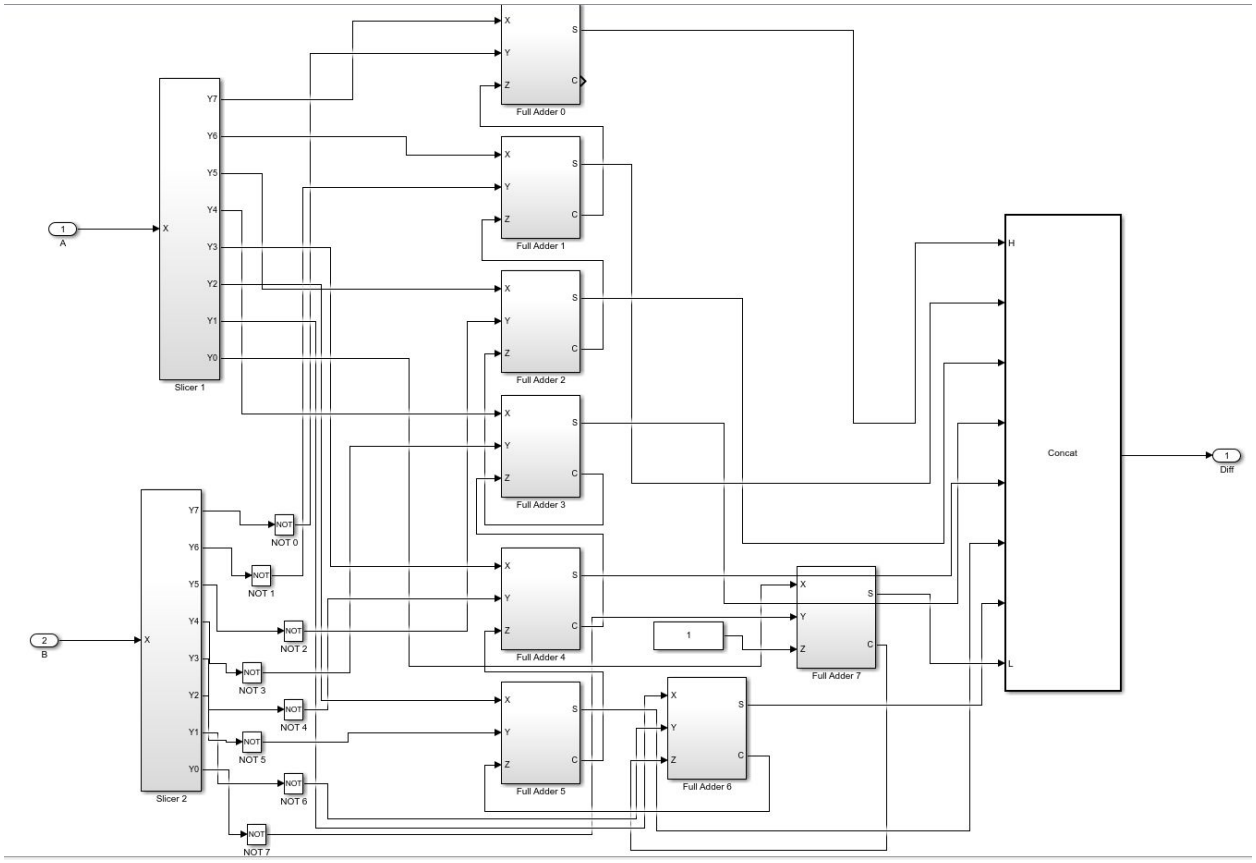


Figure 4: Subtractor Subsystem

For this subtractor circuit, the logic behind it was that the 8-bit adder circuit could be used with some changes. The changes would be that we are adding the first number with the negative of the second number. In order to accomplish this in 8-bits, the second number had to be converted to its complement using the NOT gates and a one had to be added to the sum. This was done by making the third input of the first full-adder a one. Finally, the result was concatenated and converted into its decimal form. Figures 5 and 6 show sample simulations.

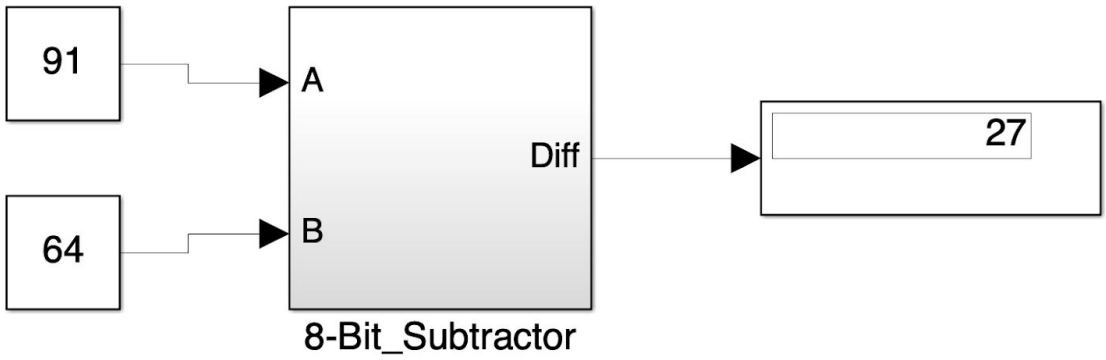


Figure 5: Subtractor Sample Simulation

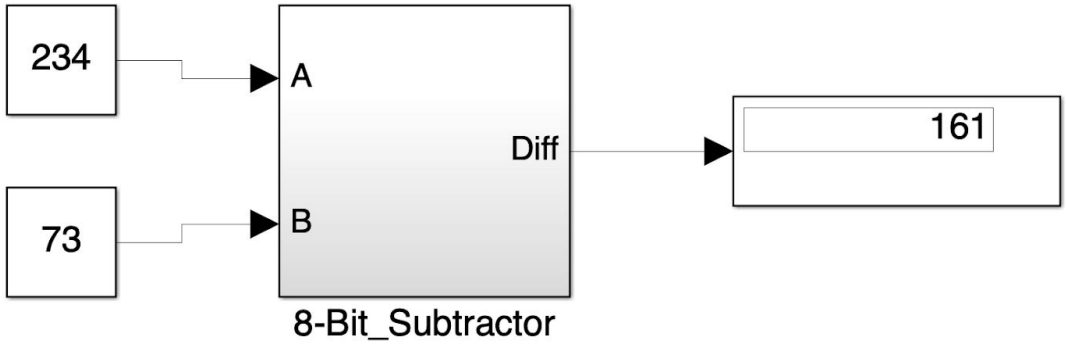


Figure 6: Subtractor Sample Simulation

Looking at both sample simulations, it can be seen how both simulations produce the correct output when both inputs are subtracted from one another. 91-64 does equal 27 and 234-73 does qual 161.

Then, an 8-bit binary multiplier circuit was created to multiply two 8-bit positive numbers together. Since the possible output of multiplying two 8-bit numbers can be a 16-bit number, the assignment required us to remove the 8 most significant bits of our 8-bit product and output the least significant, or lower, 8 bits; this was done in decimal form. The design of the 8-bit multiplier can be seen below, in Figure 7.

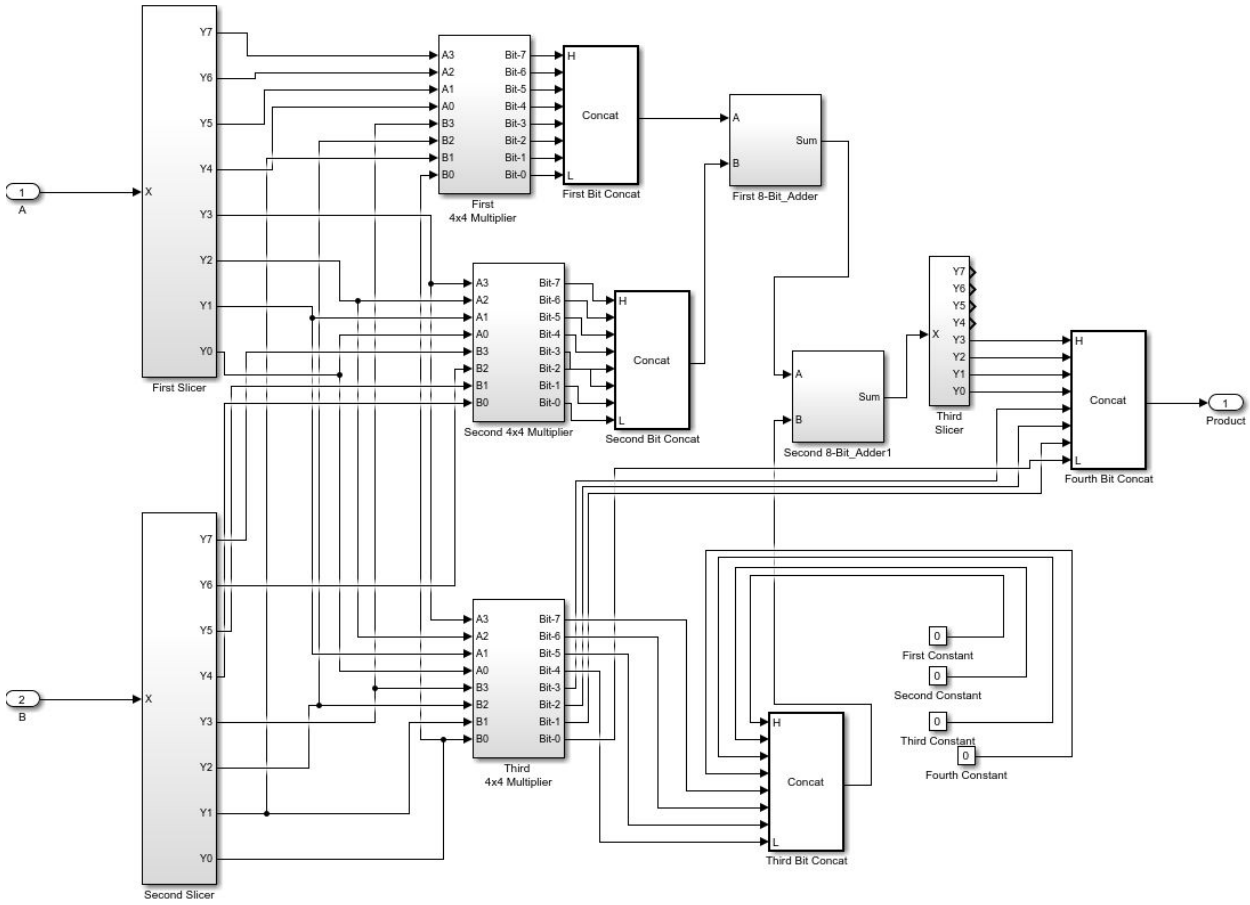


Figure 7: Multiplier Subsystem

Figures 8 and 9 show sample simulations of the multiplier circuit with numbers 12 and 13 and 15 and 7.

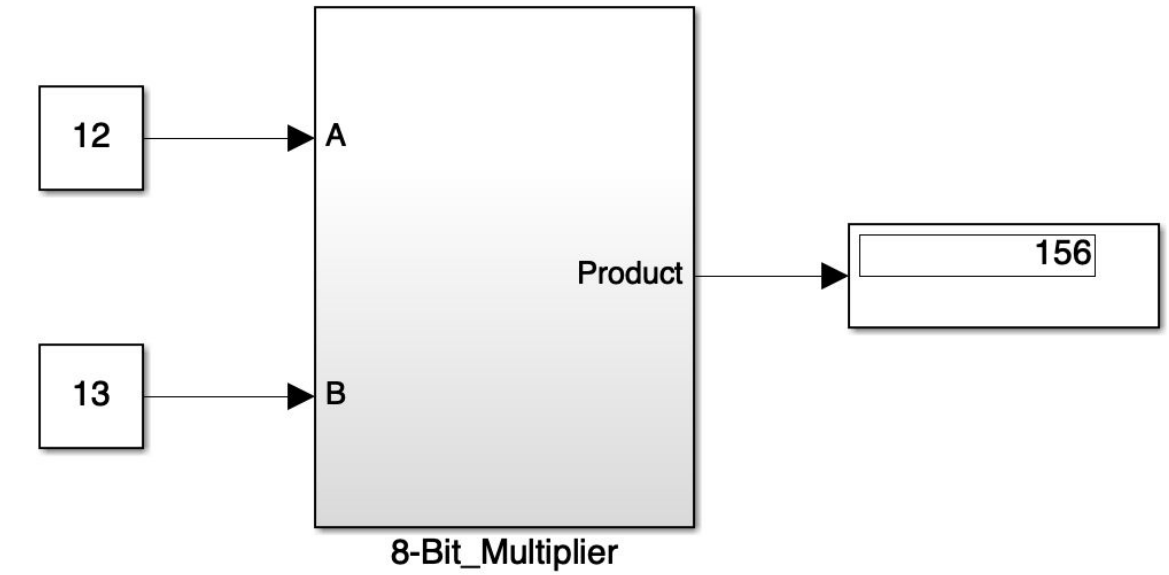


Figure 8: Multiplier Sample Simulation

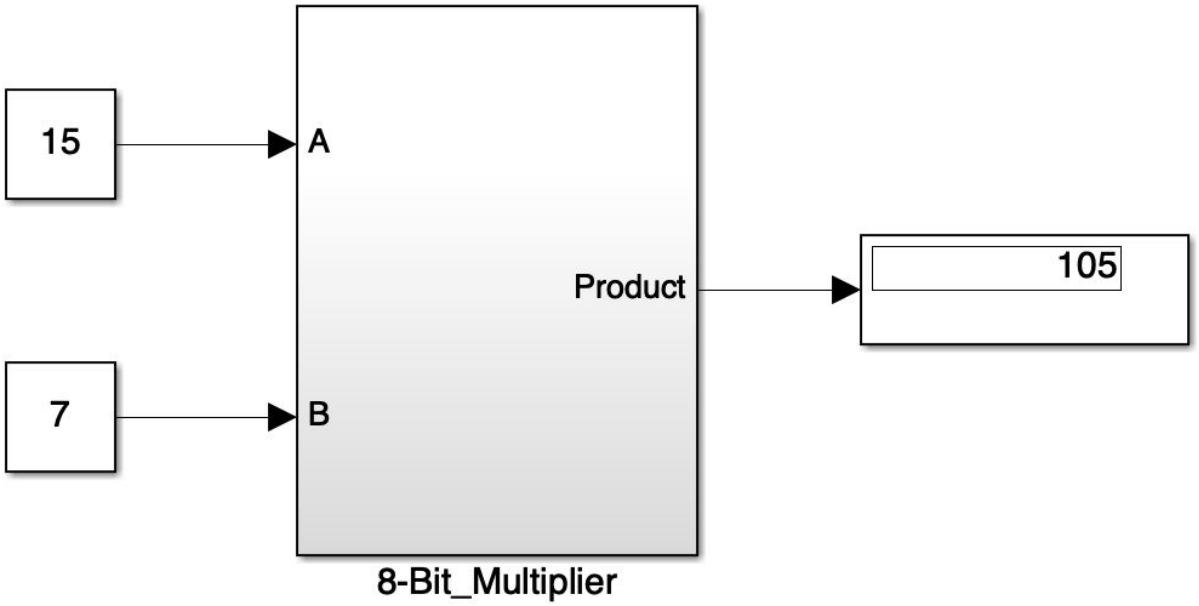


Figure 9: Multiplier Sample Simulation

Looking at the sample simulations, it can be seen that the multiplier circuit produced the correct output when the inputs were multiplied together.

Next, an 8-bit bitwise AND operator circuit was designed. The circuit takes in two 8-bit operands A and B, and returns a result Y in which each individual bit position is calculated as the logic product of the bits in A and B at the same positions (Y0 = A0*B0, Y1 = A1*B1, etc). Inputs A and B are 8-bit numbers but inputted in their decimal form. Figure 10 shows the design of the circuit.

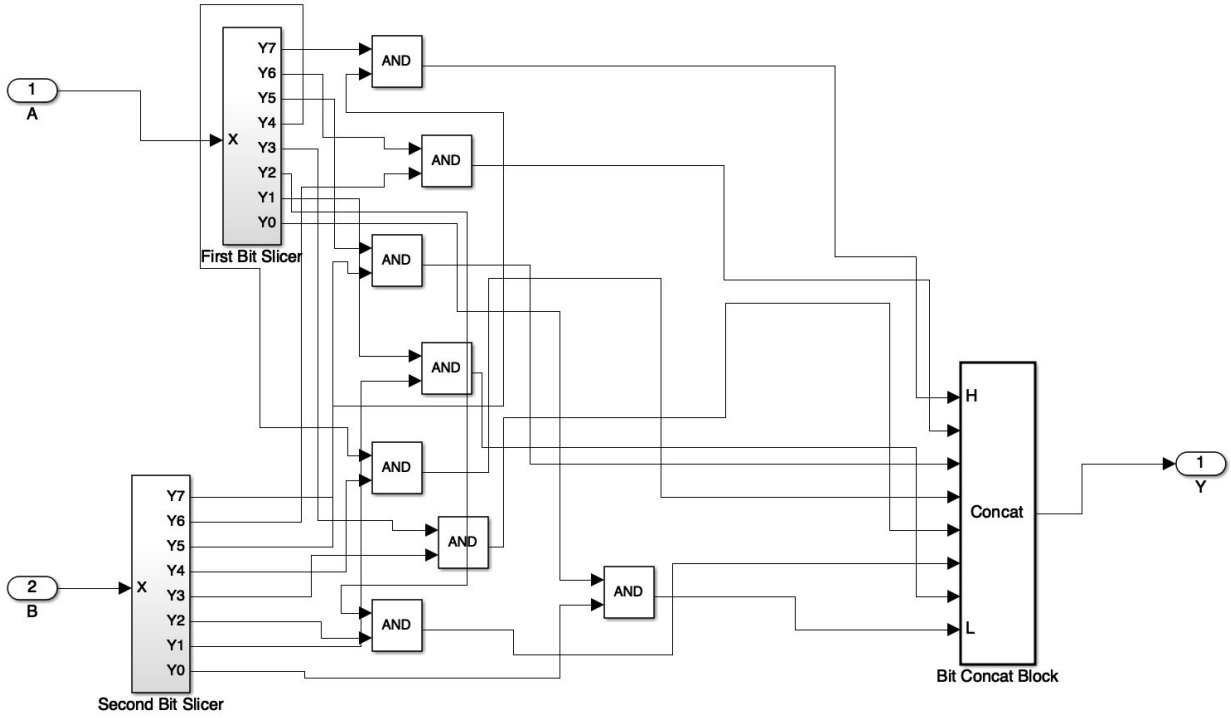


Figure 10: "AND" Operator Subsystem

First, the inputs A and B are converted to their 8-bit binary format using the bit-slicer circuits. Next, the bits of A and B in the same positions are compared using the 8 AND gates. Finally, the resulting 8-bit number is concatenated and outputted. Figure 11 shows a sample simulation of the circuit.

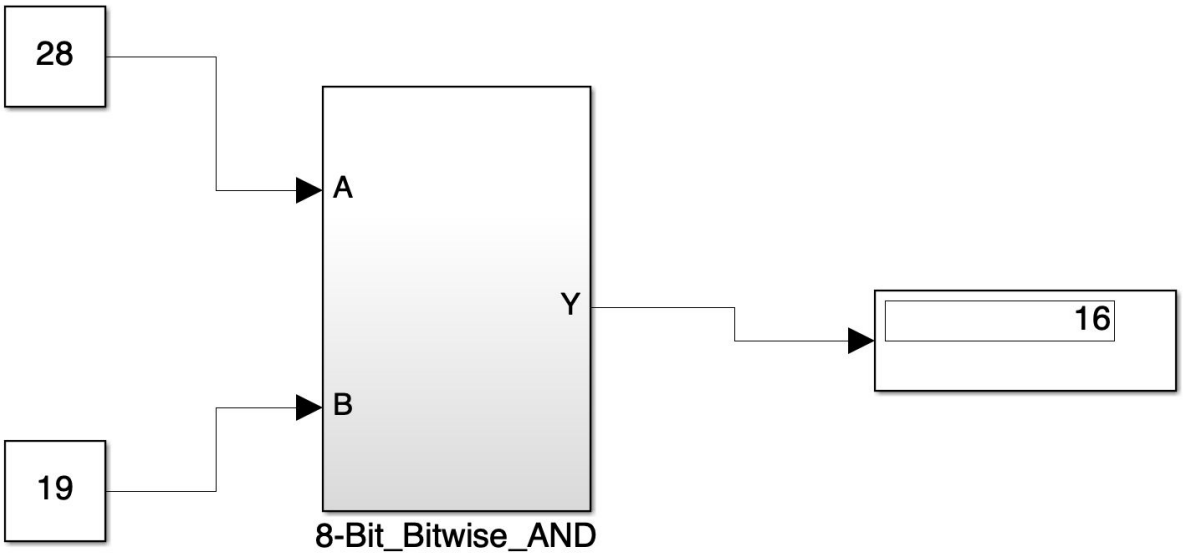


Figure 11: "AND" Operator Sample Simulation

Looking at the sample simulation, the correct outputs were given by the circuit with the respective inputs. Since the binary of 28 is 00011100 and binary of 19 is 00010011, when comparing the bits in the same positions with the AND gates, the output binary is 00010000, which is 16 in decimal/uint8.

Afterwards, a 1-bit 4-to-1 multiplexer circuit was created to choose one of four inputs depending on the value of a 2-bit selector input. In this case, the input for the selector was given as a decimal number from 0-3, which was then converted to its 2-bit representation. Figure 12 displays the circuit design.

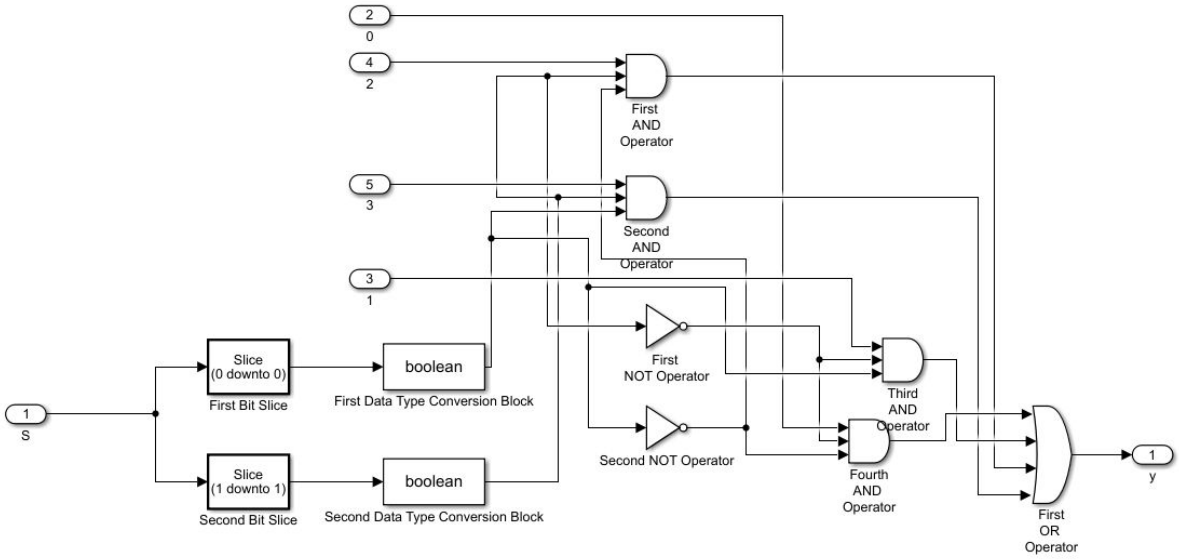


Figure 12: 1-bit 4-to-1 Multiplexer Subsystem

Looking at the subsystem, the selection input is converted into its 2-bit format using the bit-slicers. Next, using AND gates and OR gate, the 4-1 multiplier logic is applied. If the input to the selector is 0 (00), then the first input will be set as the output. If the selector input is 1 (01), then the second input will be the output. If the selector input is 2 (10), then the third input will be the output. Finally, if the selector input is 3 (11), then fourth input will be the output. Figures 13 and 14 show sample simulations with varying values for the selector input.

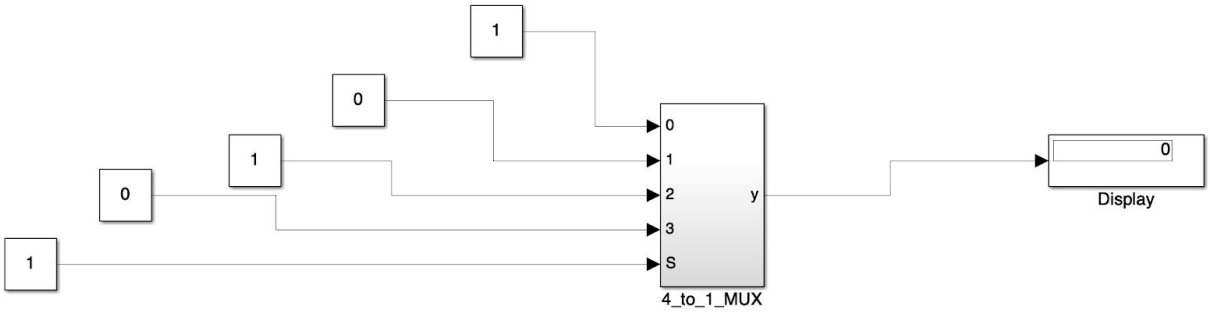


Figure 13: 1-bit 4-to-1 Multiplexer Sample Simulation

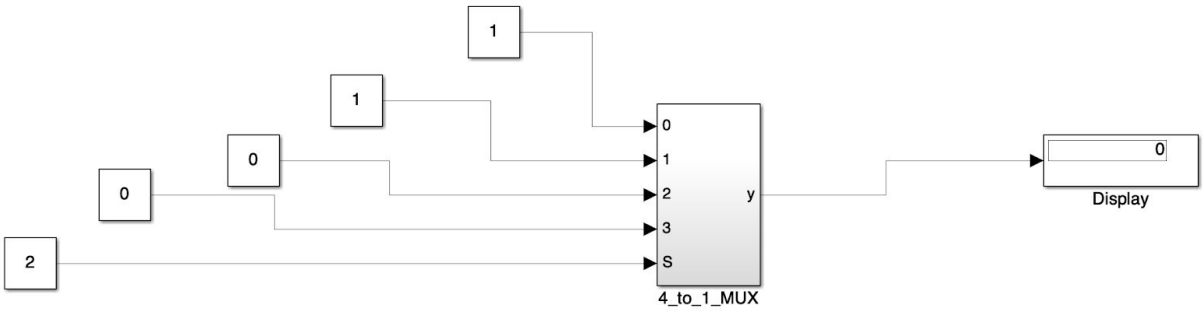


Figure 14: 1-bit 4-to-1 Multiplexer Sample Simulation

Looking at Figure 13, the output is 0 because the selection input is 1, which selects the second input (0) as the output. Figure 14 shows how the output is 0 because the selection input is 2, which selects the third input (0) as the output.

A 8-bit 4-1 multiplexer circuit was designed. The circuit is able to take in four 8-bit inputs and the 2-bit selection input to get an 8-bit output. The selector input is inputted as a decimal/uint8 form between 0-3 and the four 8-bit inputs are inputted as their respective decimal/uint8 format. These inputs are later converted to their binary formats in the circuit. The circuit design can be seen in Figure 15.

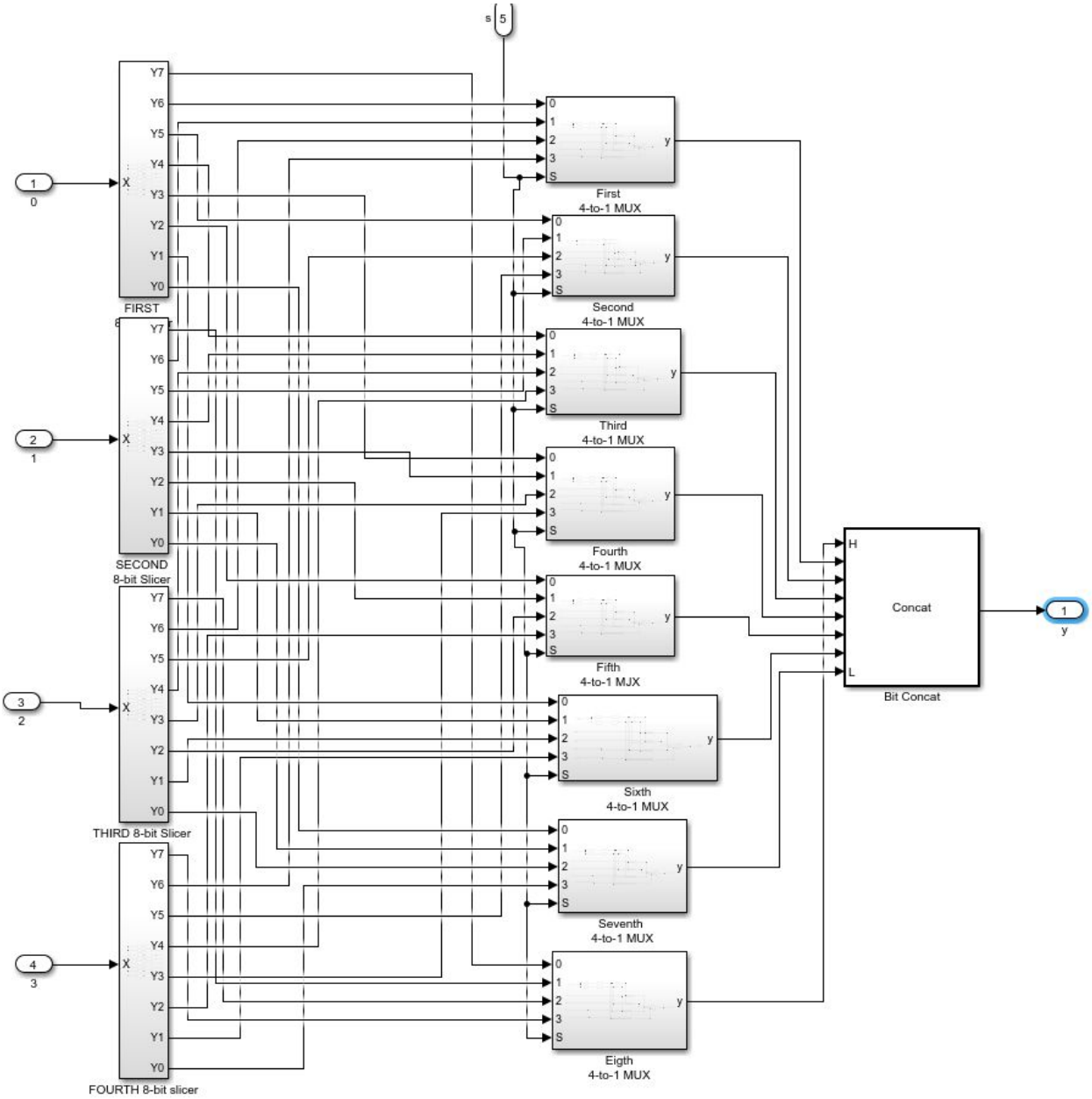


Figure 15: 8-bit 4-to-1 Multiplexer Subsystem

First, the four inputs are converted from decimal (uint8) to a binary format using multiple bit slicers. Then, eight 4-to-1 multiplexers were used to choose each bit of the 8-bit output "y". For example, if the selector input "S" is 0 (00), each of the eight 4-1 multiplexers would choose their

first input, which would be the bits of input zero. These would allow the resulting output “y” to be composed of all the bits of input zero**.** However, if the selector input “S” is 3 (11), each of the eight 4-1 multiplexers would choose their fourth input or the bits of input three. The output “y” would then be composed of the bits of input three. Figures 16 and 17 show sample simulations.

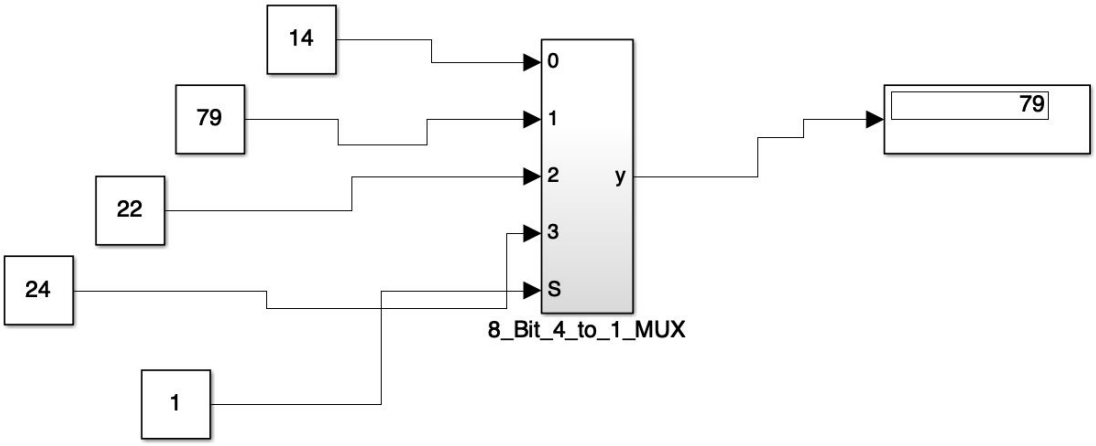


Figure 16: 8-bit 4-to-1 Multiplexer Subsystem Sample Simulation

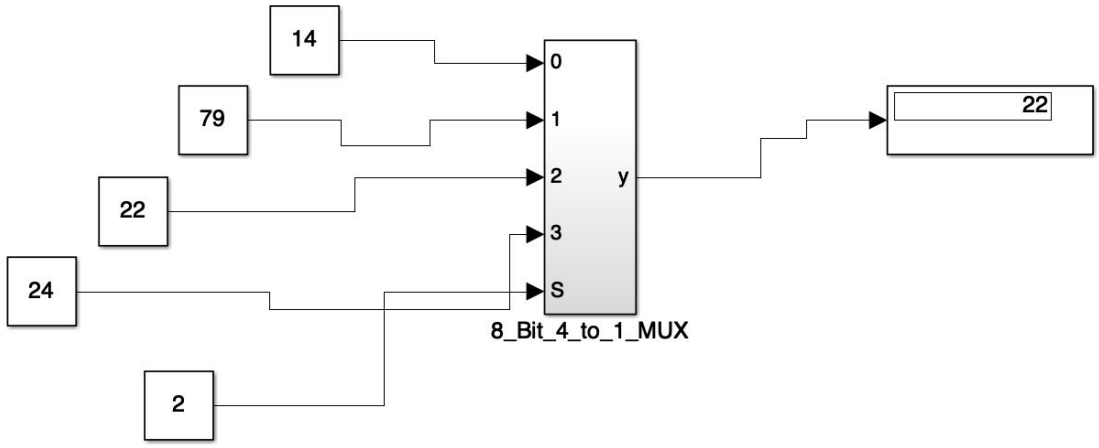


Figure 17: 8-bit 4-to-1 Multiplexer Subsystem Sample Simulation

Based on Figure 16, the selection input was 1 and this means that input one should be the output, which it was. For Figure 17, the selection input was 2 and this means that input two should be output, which it was.

An 8-bit register circuit was designed to store an 8-bit number. In this case a parallel register was created and the initial value of the register was stored as 0. In order to construct the circuit, eight D Flip-Flops were used as each stores 1-bit. Furthermore, a clock signal with period 1 was applied to the circuit. The design can be seen in Figure 18.

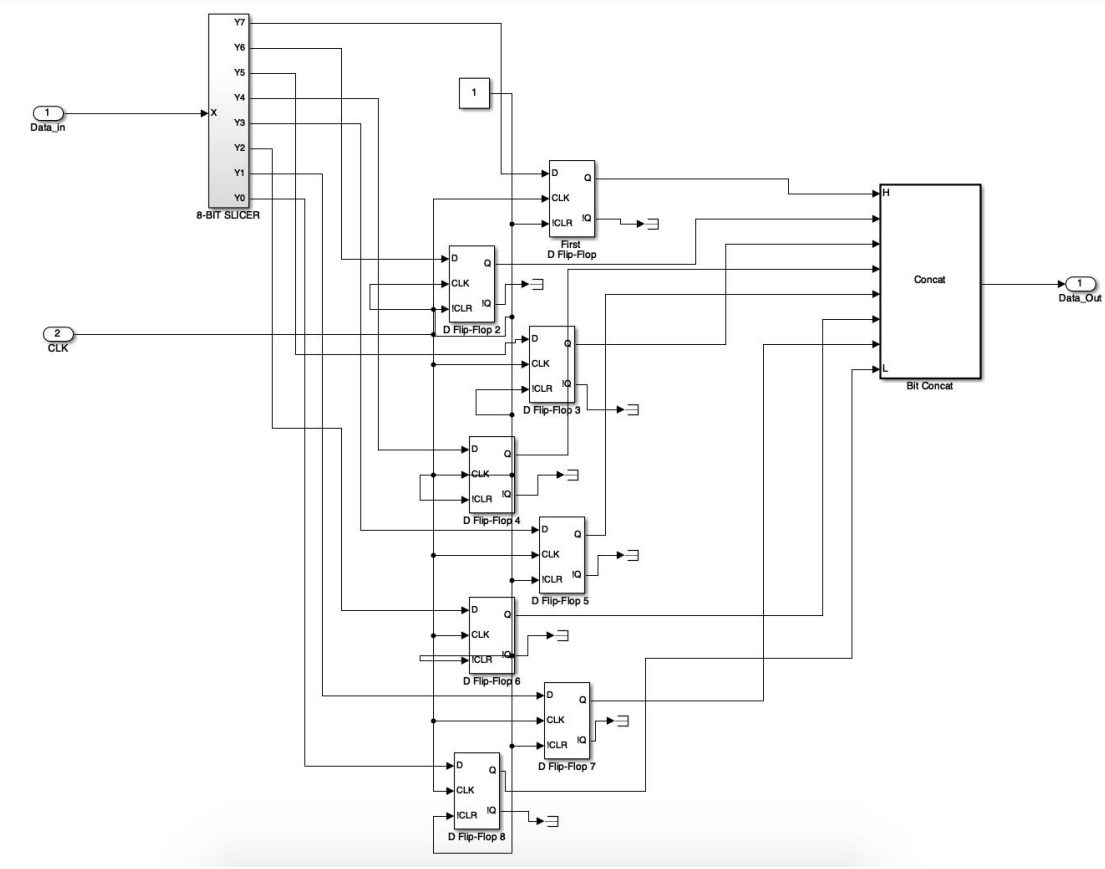


Figure 18: Register Circuit Subsystem

The clear input (!CLR) was set to a constant value of 1 as it is active low and we wanted the register to not be cleared. The output !Q of the D Flip-Flop was connected to a terminator because it was not needed. In order to test the 8-bit register circuit, a sample simulation was run. The simulation circuit and scope output of this simulation can be seen in Figures 19 and 20.

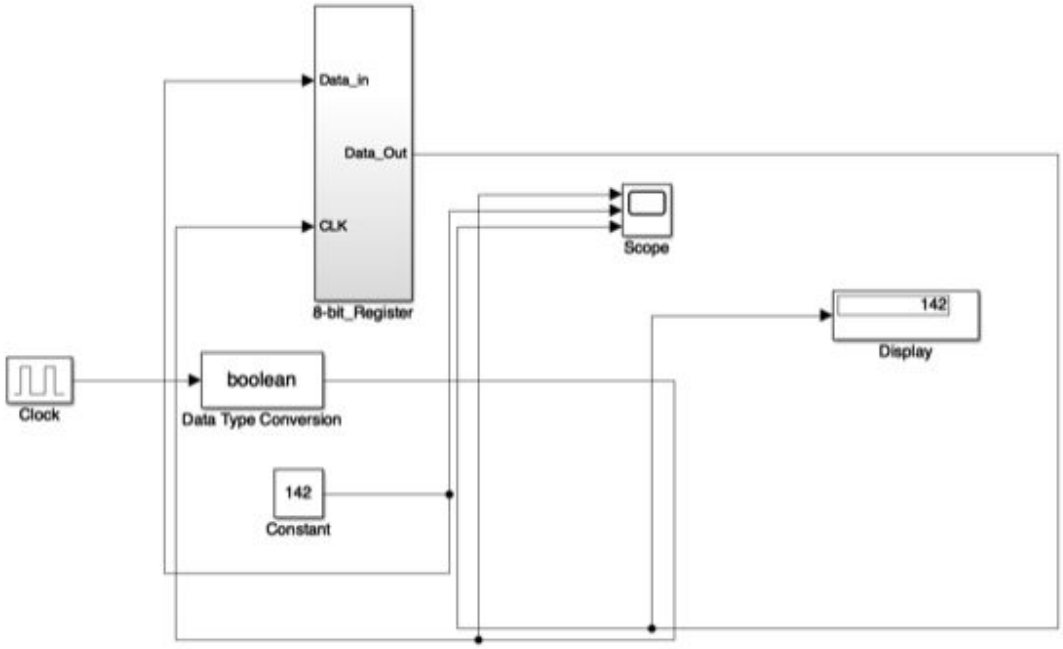


Figure 19: Register Sample Simulation

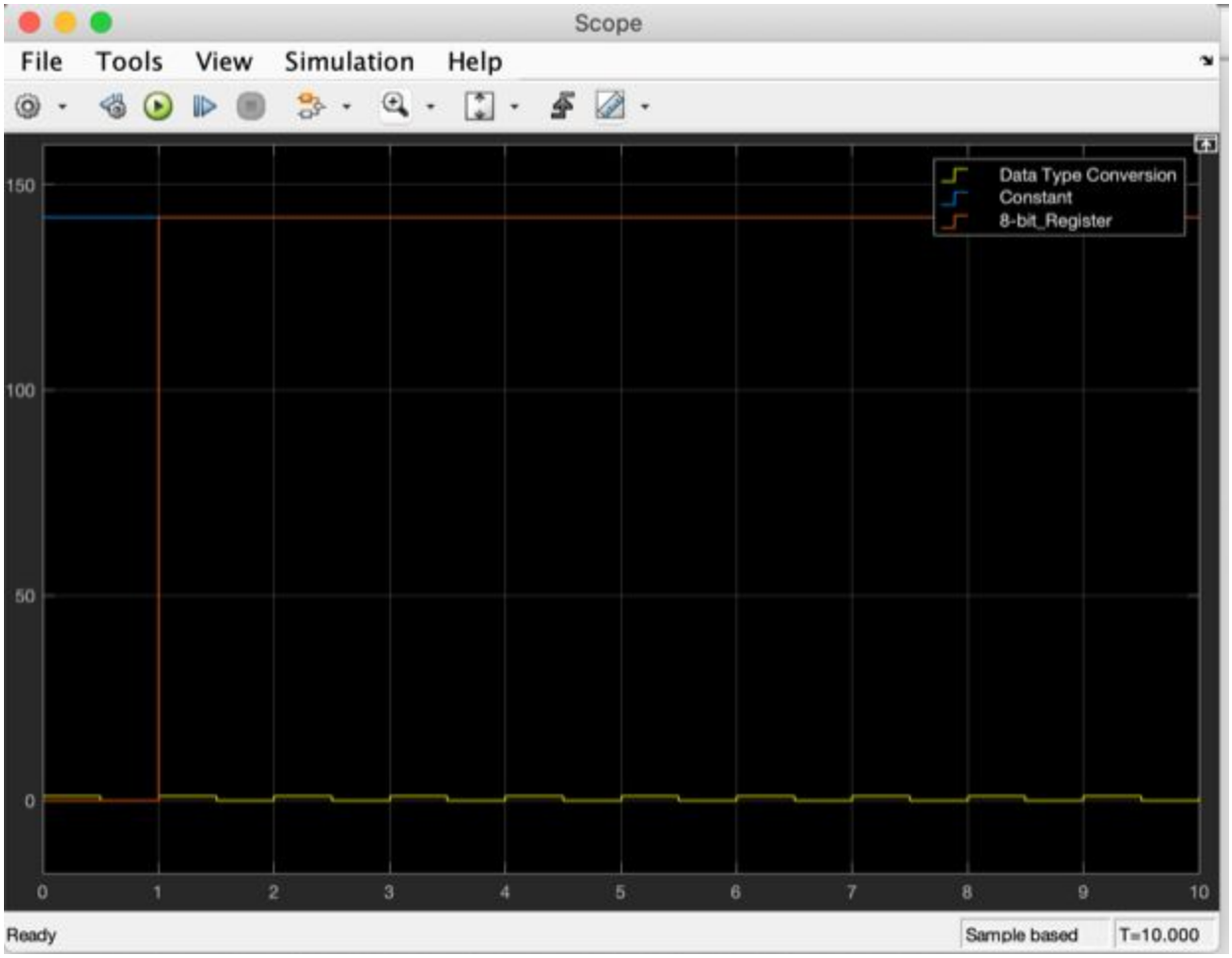


Figure 20: Sample Simulation Scope Output

Looking at Figure 20, the blue line shows the input data value of 142, which remains constant. The yellow line displays the clock signal with a period of 1. The red line shows the output of the circuit, which is correct. The output of the circuit stays at 0 for the first cycle and on the next rising edge, the input data value is loaded to the register and the output of the circuit becomes the input data value of 142.

Assignment 8

For the final portion of the project, an 8-bit processor, or ALU, circuit was developed from the circuits created in assignments one-seven. There are three inputs and one output of the circuit. The first input is the data value B. The second input is the code or selection input. This selection input is set to an HDL counter that counts in the following order: 0, 1, 2, 3, 0, 1, 2, 3, 0, and 1.The third and final input is the clock signal with period one. The design of the circuit can be seen in Figure 21. A sample simulation can also be seen in Figures 22 and 23.

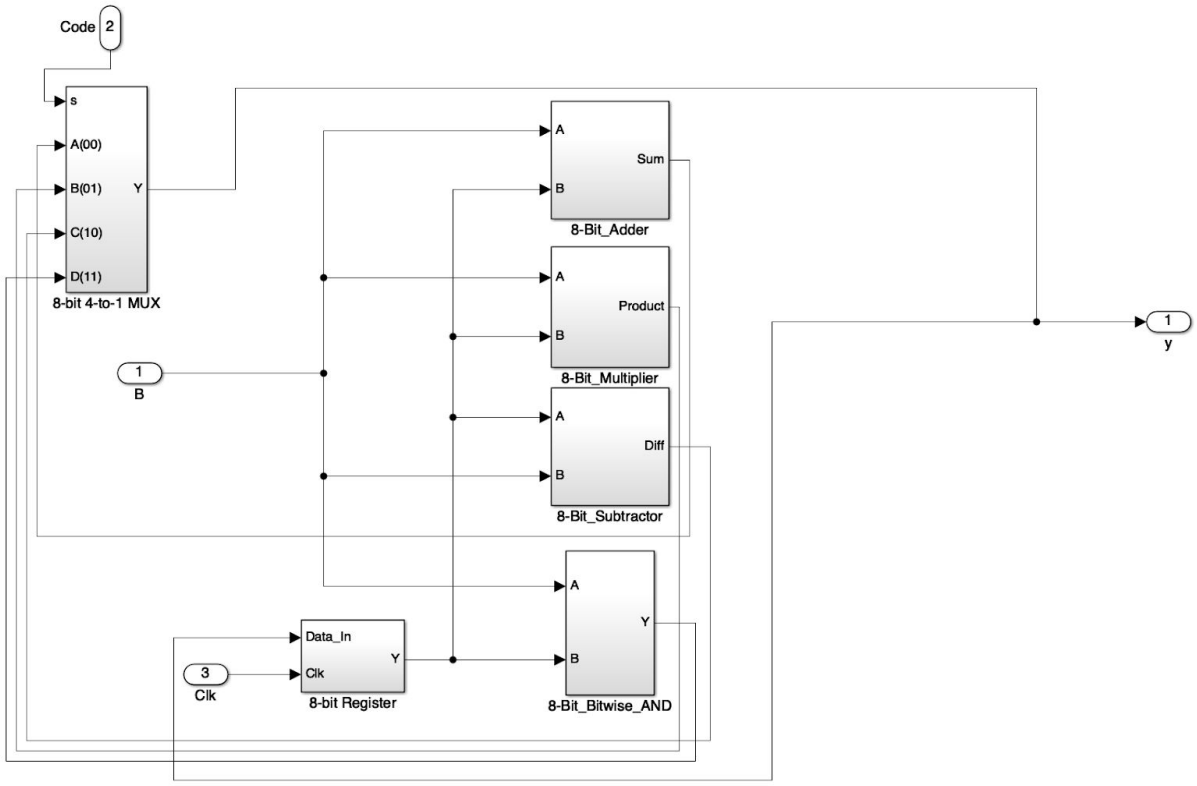


Figure 21: 8-bit Sequential Processor ALU

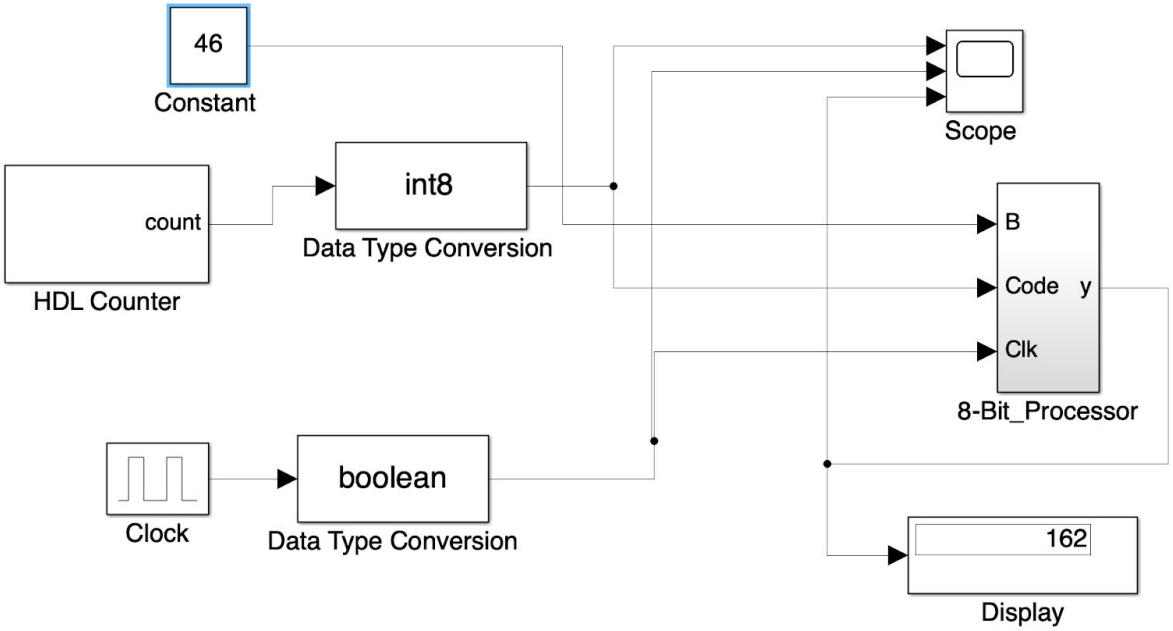


Figure 22: Sequential Processor Sample Simulation

This displayed output of 162 is the correct output given the input value of 46.

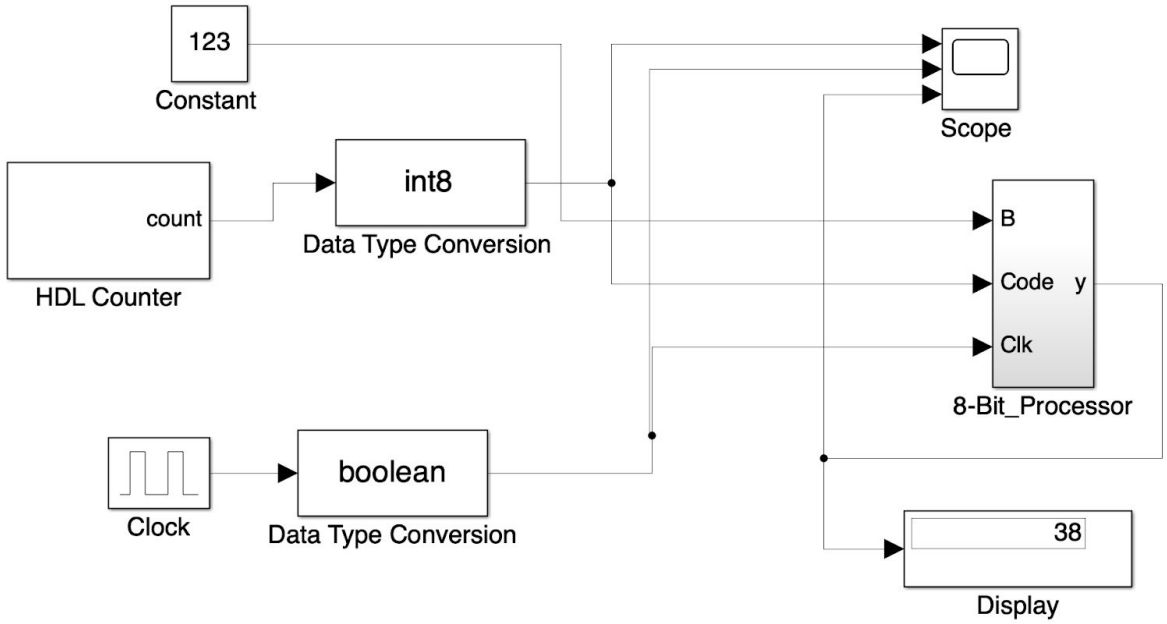


Figure 23: Sequential Processor Sample Simulation

This displayed output of 38 is the correct output given the input value of 123.

Conclusion

In conclusion, this lab is able to reveal the diverse usability of logical components in digital logic design, the basis to a processor's development. The ability to create an ALU from scratch further proves the ability for combinational logic to conduct arithmetic operations and binary manipulation. This can be scaled from 8-bit to any number of bits, generally 2^n bits. Most conventional computers are 32 and 64 bits!

Acknowledgments: I would like to thank Gabriel Peter for providing information regarding the concept of operation of the ALU and Simulink (found in the *Introduction*, *Simulink Discussion*, and *Conclusion* sections), which placed invaluable context and scope to the project. I would also like to thank Rishi Patel for assisting me in the debugging process.