Ellis Chen
11/18/2022

ECPE 174 – Lab 9 Report

Problem Statement:

The goal of the lab is to design a system that stores instructions using a FIFO, computes two's complement arithmetic, computes logical comparisons, and displays data on a seven-segment display. This lab reuses several components from previous labs, which include: the add and subtract unit, the synchronizer, resets, seven segment display, FIFO memory, and FIFO control. The most significant additions to this lab are the logic unit that handles the comparisons and the logic responsible for parsing the data that is read out from the FIFO.

The system must take in two 6-bit two's complement numbers, A and B. The system also must take in a 3-bit value that determines what type of operation the ALU performs; the list of operations include: addition, subtraction, check if A is equal to B, check if A is greater than B, check if A is less than B, and check if A is equal to 0. The instructions are stored in a FIFO during a write, and the data will not be computed by the ALU until the user switches the FIFO to read mode. The seven-segment display shows the new results every time a valid instruction is read out from the FIFO. The seven-segment displays the two's complement result if the instruction is associated with arithmetic. Else for logical comparisons, the seven-segment displays 1 or 0 to indicate true or false respectively.

System Design:

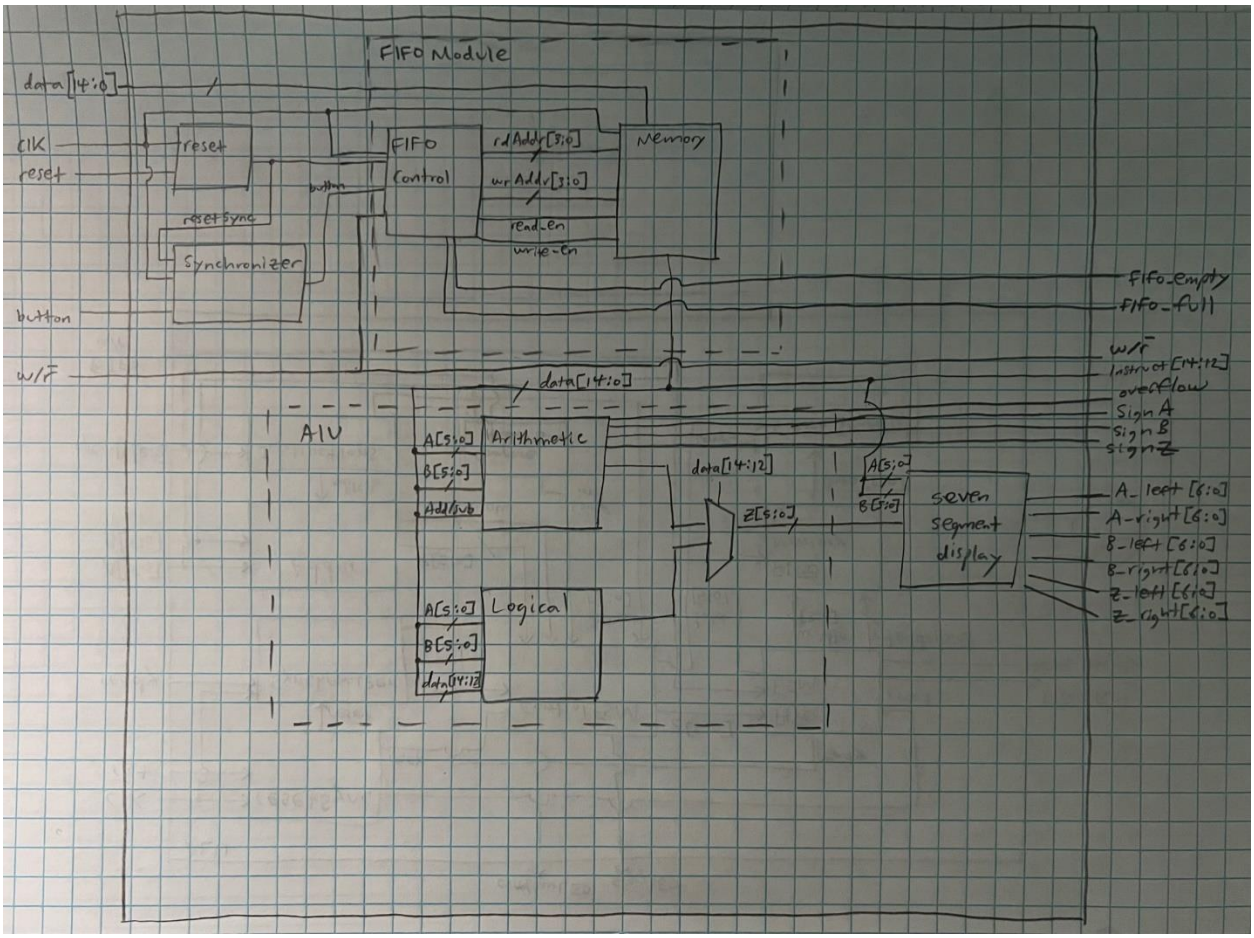

Figure 1. System Diagram shows all main components and their inputs/outputs

Ellis Chen
11/18/2022

The system was designed such that the FIFO, ALU, and seven segment display components were divided into their own components. In the problem statement, the 15 bits of data carry instructions and are stored in the FIFO during a write. The ALU does not compute arithmetic nor compare two's complement A and B numbers as long as the FIFO is in write mode. The user can store different 15-bit values for the instructions as long as the FIFO is not full. The ALU will only operate on A or B within the instructions when the FIFO is in read mode and the memory returns the stored 15-bit data. Since, the ALU has to wait for the data from the FIFO, the FIFO should be the component that comes first in the design. As shown in figure 1, the first stages of logic are the FIFO control, memory, reset, and synchronizers.

The ALU components come after the FIFO components. The ALU contains the arithmetic unit and the logic unit. It is at this phase of the design that the data read from the FIFO gets parsed into 3 segments: a 3-bit instruction code and two 6-bit numbers A and B. My design uses combinational logic to generate the add/subtract logic value for the arithmetic component as shown in figure 2. The 6-bit two's complement result and logical result have to be generated because they run concurrently in hardware. So, according to the code from figure 3, my design contains combinational logic to select between the arithmetic and logic outputs to display on the seven-segment.

```
assign a = data_in[11:6];
assign b = data_in[5:0];
assign opcode = data_in[14:12];

always_comb
begin
    if (opcode == 3'b001)
        addsub <= 1'b1;
    else
        addsub <= 1'b0;
end
```

Figure 2. Systemverilog code determines addition or subtraction for twos' complement arithmetic

```
always_comb
begin
    if (opcode == 3'd0 || opcode == 3'd1)
        z <= arith_z;
    else
        z <= logical_z;
end
```

Figure 3. Systemverilog code determines which output to display to seven-segment

Ellis Chen
11/18/2022

Testing Approach:

For simulation, I first tested the FIFO component with random testcases. To start, I tested the FIFO would remain empty when I attempted to read from it. I then generated a random 15-bit number to wrote the data into the FIFO memory for 10 iterations. The FIFO should become full by the 8^{th} iteration, and the FIFO should not accept any more writes until a read occurs. I then tested if the random 15-bit numbers were properly read out from the FIFO. I continued to read for 10 iterations, and I expect the FIFO to stop reading out data by the 8^{th} iteration. Then I alternated randomly between writing and reading with random inputs to conduct a more exhaustive test on the FIFO. When I verified the FIFO works properly, the next component to test would be the logic unit introduced in this lab.

The testing approach for the logic unit was straightforward; the goal was to verify that the logic unit made correct comparisons and returned 1 or 0 as appropriate to its inputs. The inputs were all randomized, and each testcase ran through 10 iterations. For A = B, A < B, and A > B, both A and B were randomized. For A = 0, only A was randomized. The testing approach also considered instructions for arithmetic operations, and the testbench should assert that the logic unit does not support arithmetic instructions.

When results of testing show that the logic unit works, the next step in the test approach is to combine the arithmetic and logic units into one module and test it on a higher level. The higher-level module is referred to as the ALU component. The ALU should take in the 15-bit data that is expected from reading out the data from the FIFO, and the ALU should expect to output either the arithmetic or logic result to the seven-segment display. So, in the testcases, both A and B are always randomized. The testbench seeks to verify the behavior of the ALU component in the following order: addition, subtraction, A = B, A > B, A < B, and A = 0. The 3-bit instruction code changes accordingly to the type of operation. Upon verification of the ALU component, the final step in the testing approach is to combine all components and test with a full set of inputs and outputs.

The last testbench combines all components: the FIFO modules, the ALU modules, the seven-segment display module, and other miscellaneous modules. My testing approach started with hardcoding various 15-bit values for input data. I wrote out the 15-bit values such that the testbench covers all arithmetic and logic operations. I tested these values for writing and then reading. I then conducted a few write testcases and then tested the reset component. After the reset, I wrote data to the system again and read from it to ensure the system resumed functionality. After verifying the system worked with hardcoded values, I generated randomized 15-bit inputs for writing and reading for a more exhaustive test. I verified that the system functions as expected by looking at the 7-bit values for the seven-segment display.

After I verified the system functions as expected, the next step was to implement the design onto a board and test that the FIFO writes data in and reads data out to the ALU components, and I would also need to test the behavior of the ALU components by looking at the results shown on the seven-segment display.

Ellis Chen
11/18/2022

Results:

In simulation, the FIFO component showed promising results based on randomized inputs. Figure 4 shows the waveform of the of the FIFO status and data written to and read out of the FIFO. Firstly, the testbench attempts to read from FIFO, but no data should be read out since FIFO is empty. Secondly, the testbench wrote random data to FIFO to fill up 8 entries, and the FIFO did indicate that it was full. Thirdly, the system read from FIFO 8 times until the FIFO indicated it was empty. The results from figure 4 show that the FIFO behaves properly and data can be written to and read from given the available space initialized in memory.

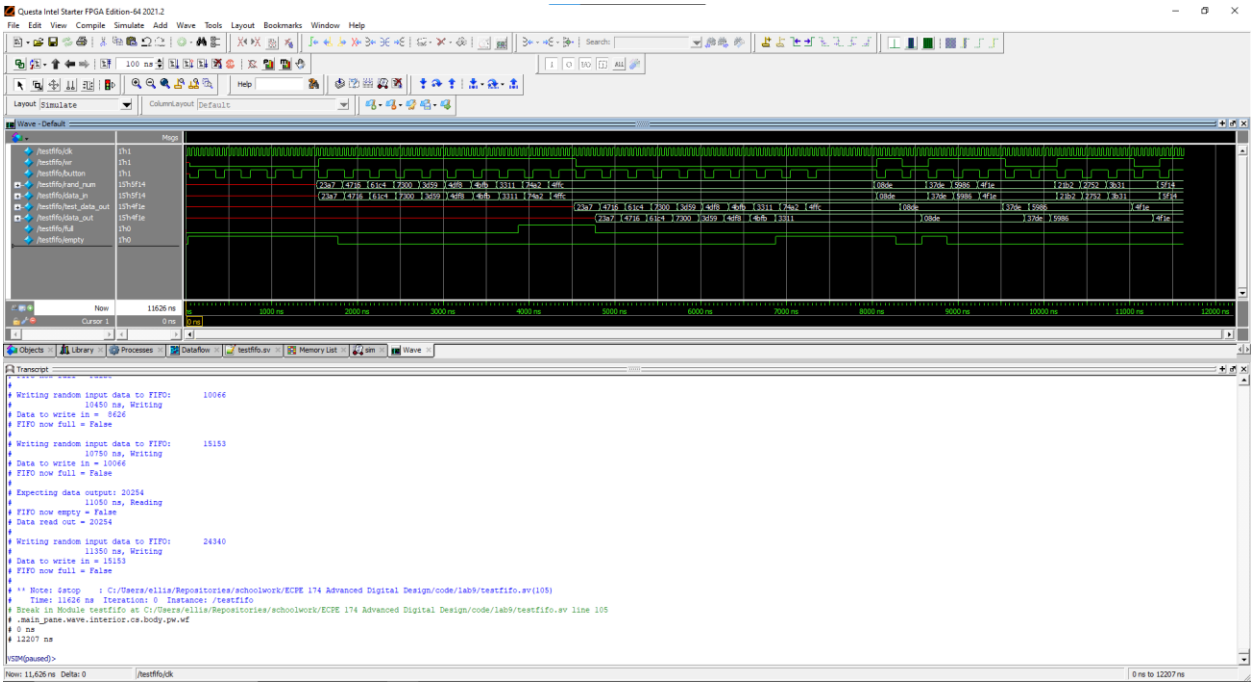

Figure 4. FIFO testbench

The second component to verify in my testing approach was the logic unit. I tested my logic unit with randomized inputs, so A and B were all randomly generated. As shown in figure 5, the testbench was supposed to assert any errors when the logic unit encounters invalid inputs. In this case, the testbench was able to return an error message saying the logic unit cannot perform arithmetic operations as suggested by an invalid 3-bit instruction code. Next, the waveform from figure 5 shows tests comparing A = B with random inputs, then it shows A > B with random inputs, then tests comparing A < B with random inputs, and then finally comparing A = 0 with random inputs. Overall, the results prove that the logic computes 1 or 0 correctly according to its inputs.

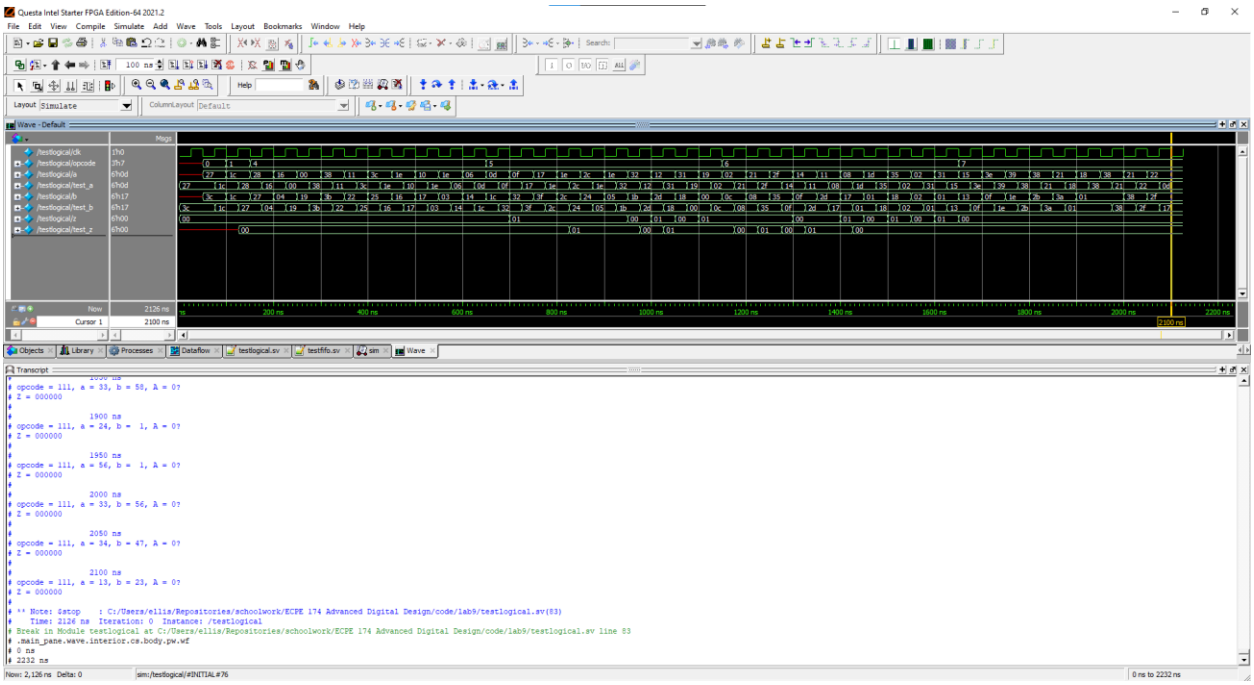

Figure 5. Logical operation testbench

The third component to test was the ALU component, which housed the arithmetic unit with the logic unit. During testing, I expect the ALU to return an output appropriate to its 15-bit input; I also expect the ALU to choose its output correctly between the two's complement result or the logical comparison result depending on the 3-bit instruction code. Figure 6 shows the test in the following order: addition, subtraction, A = B, A > B, A < B, and A = 0. All inputs were randomized for the testcases. Results from the simulation show that the ALU component works as expected. With the FIFO components working as well, the next step was to combine the FIFO and ALU components with the seven-segment to complete the full system design.

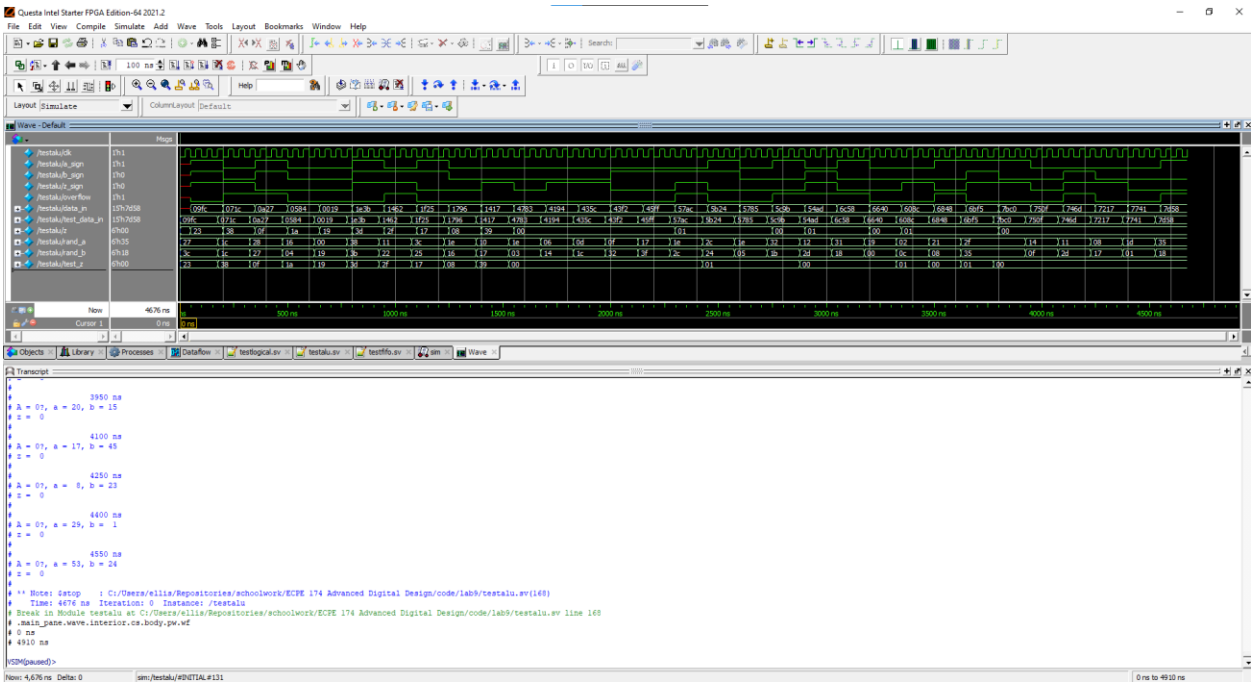

Figure 6. Arithmetic and Logical operations (ALU component)

The final testbench was geared towards verifying all components work connected together. Figure 7 shows the waveform for the behavior of the full system. The first round of tests was conducted without random numbers. The tests were as follows: write entire 15-bit instruction to FIFO multiple times, read and check seven segment display multiple times, write multiple times and reset the system, attempt to read after the reset, write then read to verify system is no longer in reset. The second round of tests used randomized inputs, writing multiple times and then reading multiple times. Results of testing showed that all test cases, with and without randomized inputs, yielded expected outputs. The full system with all of its components worked together, The FIFO was able to store the instructions until they were read out to the ALU, and the ALU yielded outputs appropriate to the instruction set. The reset was also tested on the system, and the FIFO was empty when the reset was active low; the system resumed full operation when the reset was high again.

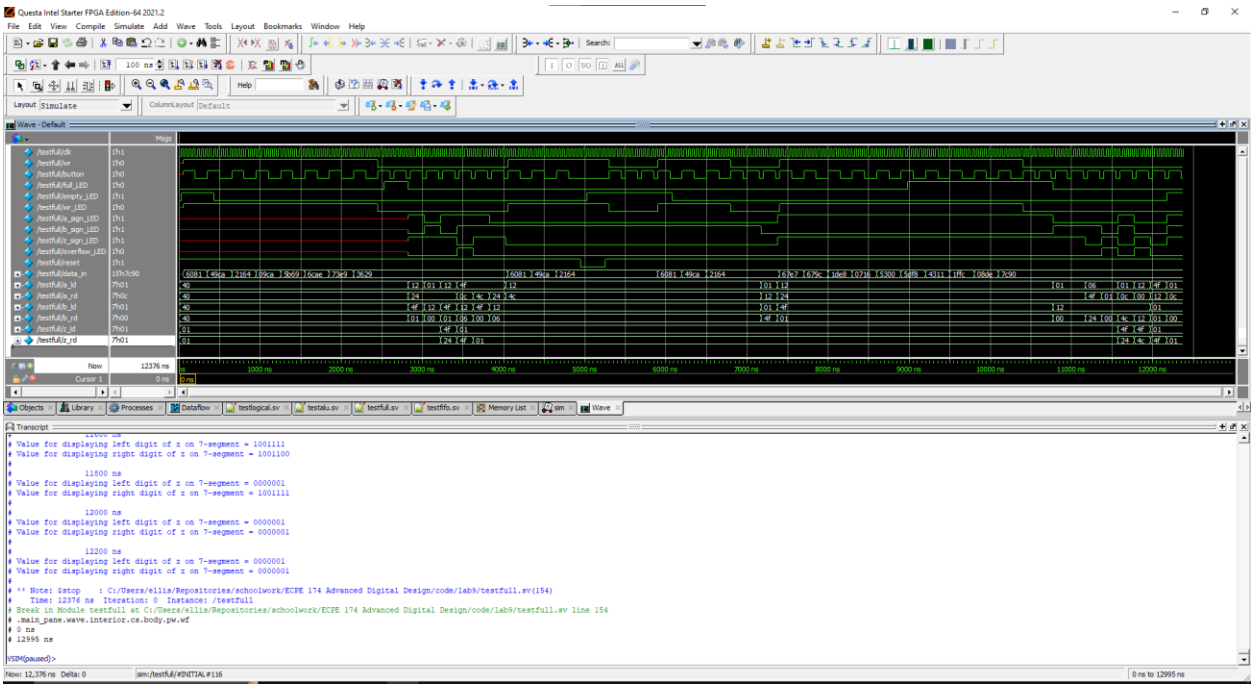

Figure 7. Full system, including the FIFO, ALU, and seven segment

Analysis:

My test procedure, which involved testing the FIFO first, followed by the logic unit, followed by encapsulating the logic unit with the arithmetic unit, and followed by testing the system with all of its components connected, worked as I proposed. Each test proved that the individual logic components were ready to be integrated into a larger, higher-level component. No modifications to the design or testing procedure was required during the design and testing phase. The complete build in Quartus consisted of 511 logic elements and 161 registers. Just by observing the ratio between these two quantities, it is apparent that my design contains more always_comb logic than always_ff logic. This observation also tells me that I may have more combinational logic in between FFs, which may also mean that the critical path of my design is relatively large.

Since the board runs on a 50MHz clock, the period would be 20 ns. In the timing analyzer, as shown in figure 8, the waveform on the right shows the timings for the critical paths. The path shown in figure 8 is from the output of the FIFO memory to the output of the seven-segment displays. This makes sense because the FFs for reading and writing to the memory were the last FFs in the entire design; the rest of the system runs entirely on combinational logic. According to figure 8, I assume that the setup time would be roughly 3 ns after the rising edge of the clock, and the hold time would be 14.4 ns after the rising edge of the clock. My design leaves me with 2.53 ns of slack, which I know can be improved upon. To improve the speed of my design, I would have to add FFs within the ALU component to divide up the delay from all the combinational logic. I would also add more FFs between the ALU and the seven-segment display, essentially replacing always_comb blocks with always_ff blocks to reduce the critical paths. As of my current design with no changes, I could clock my design slightly faster to around 57MHz before my design runs into issues like metastability.

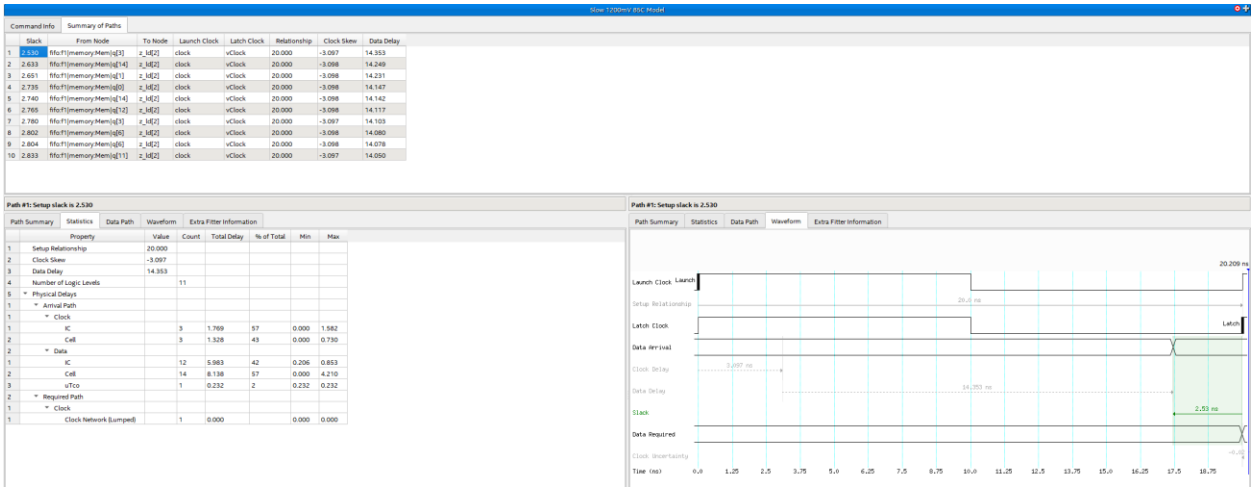

Figure 8. Timing Analyzer

The arithmetic unit takes up more space than the logic unit. In my design, the arithmetic unit consists of 6 ripple carry adders, each adder containing multiple stages of combinational logic. In contrast, the logic unit has a smaller number of stages in its logic. Since the space requirements are higher for the arithmetic unit than the logic unit, the time requirements for the arithmetic unit would be greater than the logic unit as well. More combinational logic would lead to a greater critical path, so the setup and hold times would also be greater.

To complete this design, I assumed that the only components that require FFs would be the FSM that controls the FIFO, the FFs for the memory, and the FFs for the reset and synchronizer. I assumed that all other components did not require any FFs to function. These assumptions were true, but those assumptions led to suboptimal timings in my design. If I had planned the design differently, I would have taken measures to place more always_ff logic to reduce the longest propagation delay path, and subsequently increasing the maximum clock frequency that can run my system.