Final Project Report: Repeated Subtraction Algorithm

Charlie Singer & Lilah Kelly

ECE 318

Professor Traver

June 5th, 2021

ABSTRACT

The goal of this project was to create a VHDL model of a division algorithm. This algorithm was created as both a high level state machine (HLSM) and as a control and datapath structure. Both were first created manually and then implemented into VHDL code. The code was successfully run, but not all test cases were able to be executed. This was due to the slowness of the algorithm selected.

DESIGN OBJECTIVE

For this project, the overall objective is to select a divider algorithm and implement it into VHDL. This naturally expands on the laboratories completed throughout this course. The particular algorithm chosen for this project was the repeated subtraction algorithm [1]. It takes two inputs, which are the numerator and denominator, and returns two outputs, which are the quotient and the remainder. In the original algorithm, two cases were handled. If the numerator was less than the denominator, then the algorithm would return a value of zero for the quotient and the value of the numerator for the remainder. Otherwise the denominator would be repeatedly subtracted from the numerator until the numerator was no longer greater in value than the denominator.

One of the disadvantages of this original algorithm was the lack of cases. In order to fully optimize this algorithm, two more cases were added. The first would handle when the denominator is equal to zero, and the second would handle when the denominator is equal to one. Even with the added cases, the modified algorithm still has some disadvantages. One of the more obvious drawbacks is that the repeated subtraction algorithm takes a significant amount of time to run when compared to other divider algorithms. However, this algorithm also comes with its advantages. One of the most notable advantages is the simplicity of the code, which meant a straightforward implementation for the HLSM, the datapath and control FSM. This also made the process of the code simple, meaning that it would be easy to explain to someone not familiar with algorithms or coding.

PROCEDURE

The first step in creating a divider circuit is to identify the algorithm that will be used. This project implemented the repeated subtraction algorithm [1]. Once an algorithm has been identified, creating a HLSM helps in understanding and planning the implementation of this algorithm. The completed HLSM needs special states to account for inputs that the algorithm is not designed to handle. For example, the repeated subtraction algorithm can not divide by zero, so there is a special state for that case.

The next step is to implement the HLSM into VHDL code. The VHDL design uses cases where each state will be its own case. After completing the design, two VHDL testbenches are created. The first testbench contains five tests to address extreme cases that the algorithm should be able to run. The second testbench tests all possible 8-bit value numerators for a denominator value of five. A key part of implementing the second testbench is ensuring that it runs within the EDA playground timing restrictions, which may require limiting the range of numerator values tested.

Next, the control and datapath should be extracted from the HLSM. The control is a finite state machine (FSM) that is very similar to the HLSM. It should have the same number of states and transitions for each state. The datapath is a combination of multiple components that will each perform their own operations based on the transitions and changes for the output values.

After the control and datapath are designed, they need to be implemented into VHDL code. The datapath and control are both separate VHD files, with the datapath being a structure composed of separate VHD files for each type of component needed. A top level entity is then used to facilitate the interactions between the datapath and control. The entity is then tested using

the same two testbenches from the HLSM implementation. If both of these testbenches are functional, the algorithm is successfully implemented.

DESIGN & RESULTS

For the divider circuit, the top level block diagram is fairly simple. To start with the obvious, a numerator and denominator are needed as inputs, and a quotient and remainder are needed as outputs. In order to make this an effective circuit, reset and clock inputs are added, as well as a start input and a done output. Additionally, because not everything can be divided, a zero flag output is needed, which will raise to indicate when the circuit is asked to divide by zero. The complete block diagram is pictured below.

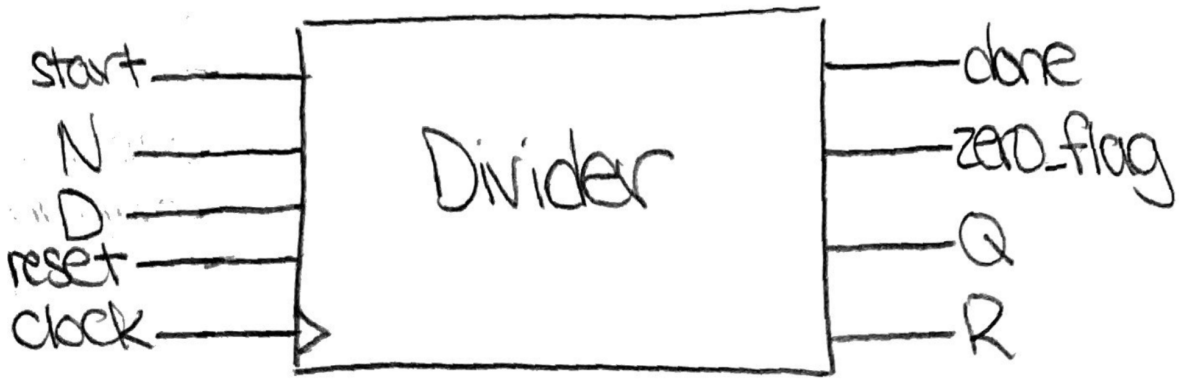

Figure 1: Divider Top Level Block Diagram

As seen in Figure 1, the numerator is represented with the letter N, the denominator is represented with the letter D, the quotient is represented by the letter Q, and the remainder is represented by the letter R. With this established, the algorithm can now be selected. The aforementioned repeated subtraction algorithm was chosen to design this circuit [1]. The full modified algorithm has been represented below in pseudo code.

REPEATED-SUB(start, n, d)

```
1.      zero_flag = 0
2.      if (start == 1) then
3.          if (d == 00000000)
4.              return zero_flag = 1, done, q, 00000000
5.          else if (d == 00000001)
6.              return zero_flag = 0, done, n, 00000000
7.          else if (n < d)
8.              return zero_flag = 0, done, 00000000, n
9.          let q == 000000000
10.         let r == n
11.         for (r >= d; r - d)
12.             let q == q + 000000001
13.         return zero_flag = 0, done, q, r
```

Figure 2: Modified Repeated Subtraction Algorithm

The zero flag is initialized as low in line 1 of Figure 2, and then the algorithm checks to see if start is high on line 2. All following procedures only occur if start is high, otherwise nothing happens. As the n and d inputs for this implementation are 8 bits, the values are represented as such. The lines that return the outputs, which are lines 4, 8, and 12, all return some value for the zero flag and the done, q and r outputs. By taking each if statement within the initial if statement on line 2, the states for the HLSM can be visualized. An additional three states are formed for the for loop from lines 11 to 12, the corresponding return statement on line 13, and the initial state to store the initialization of all variables and outputs. In total, six states are needed to accurately represent the HLSM. To avoid using a global reset for timing reasons, the

start input is used to indicate when a previously terminal state should return to the initial state. An illustration depicting this is shown below.

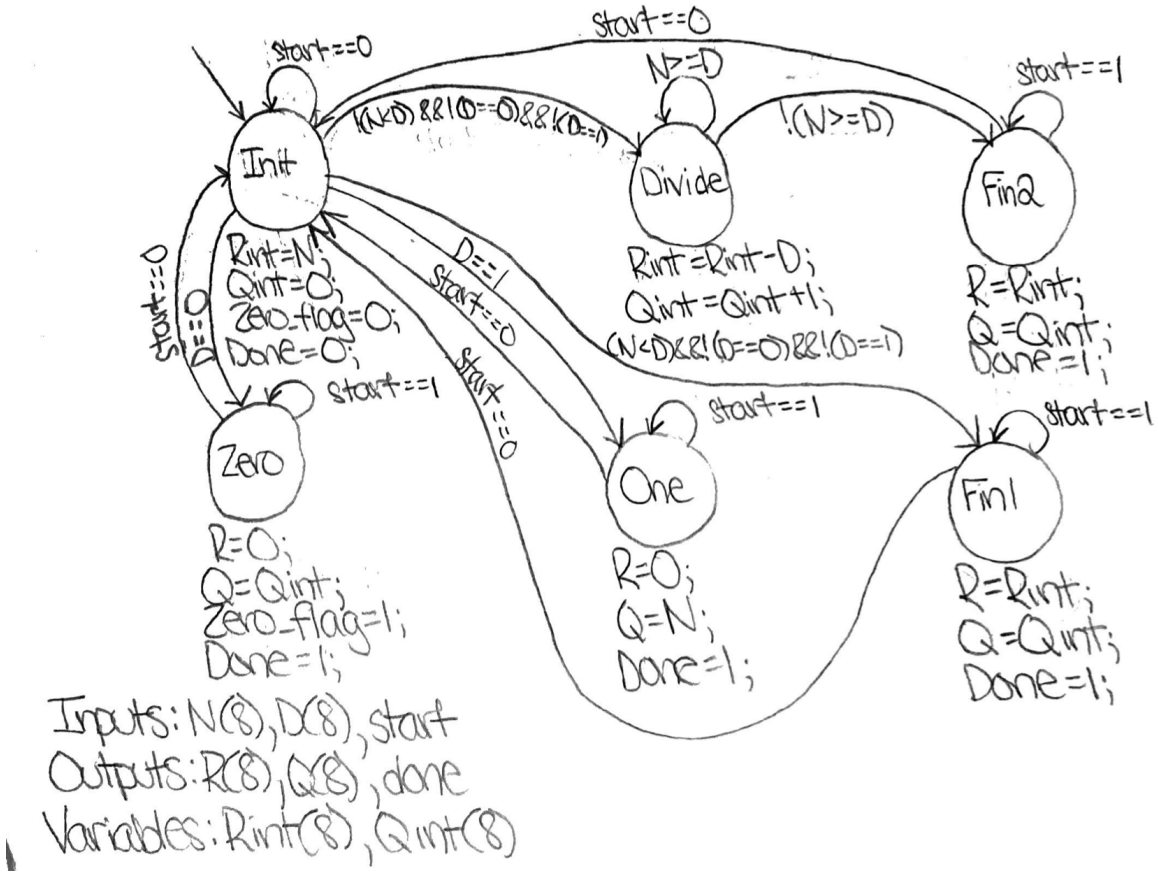

Figure 3: HLSM

Figure 3 shows that the six states are named Init, Divide, Fin2, Zero, One and Fin1, which accurately describes each of their purposes. The Zero state is visited only if the algorithm is asked to divide by zero, and the One state is visited only if the algorithm is asked to divide by one. The Fin1 state is visited if the algorithm is asked to divide by a value larger than the numerator. Otherwise, the Divide state is visited, which represents the for loop seen on lines 11 and 12 in Figure 2. This state is then left and the Fin2 state is visited when the numerator is no longer greater than or equal to the denominator value. All states returning a done value of 1 were

the original terminal states which have since been modified with a condition involving the start input being low to return to the Init state. As done in previous assignments in this course, the HLSM is then implemented into VHDL using the EDA playground [2].

The implementation is accomplished by first writing the entity of the HLSM, which includes all of the inputs and outputs shown in Figure 1. The architecture is then created, with two signals declared to represent the variables in Figure 3. All six of the states are then declared, and the state transitions and state outputs for each individual state are coded using Figure 3 as a reference point. Once the clock process concludes, the two signals representing the variables are assigned to the R and Q outputs respectively. Using the testbench created to test the required cases for Project Assignment #2, the success of the VHDL implementation can be demonstrated. Pictured below are the first two of these cases, which are 1/200 and 5/0 respectively.

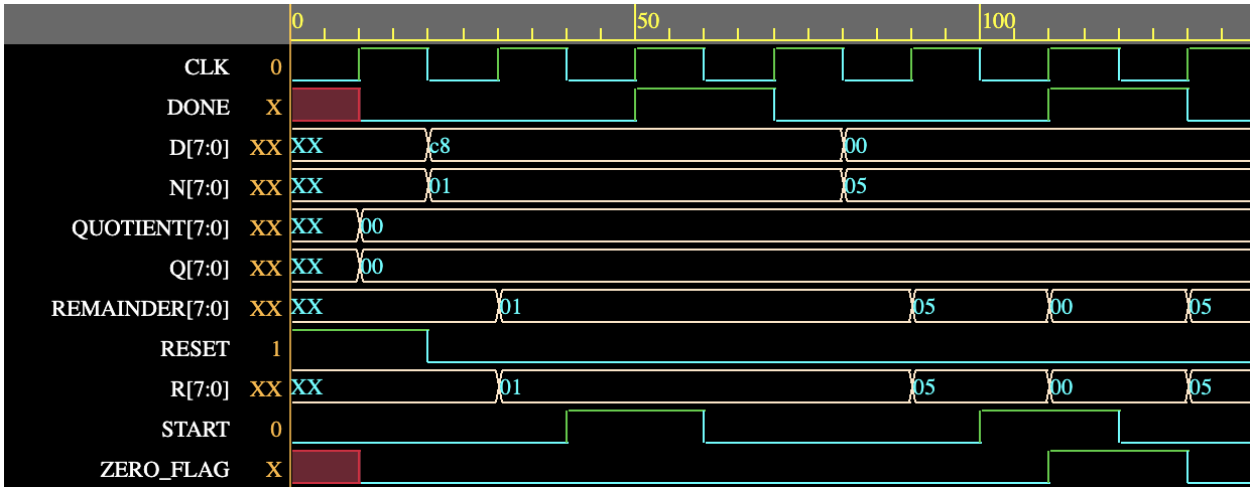

Figure 4: Screenshot of HLSM simulation

These two test cases can be found from 0 to 140ns in the EPWave, with the signals shown in hexadecimal. The test case of 1/200 is seen from 20ns to 80ns, and as seen by the R[7:0] output, the HLSM implementation successfully entered the Fin1 state. The test case of 5/0 is

seen from 80ns to 140ns. The ZERO_FLAG signal goes high from 110ns to 130ns, indicating that for this test case, the HLSM implementation successfully entered the Zero state.

With the success of the HLSM implementation in the VHDL verified, the next method of implementation can then be used. This method mirrors what was done in Lab 5 of this course, where the HLSM is broken down into a control FSM and a datapath. Using Figure 3 as a guide, the components necessary for the datapath include two load registers and two 2x1 muxes, with one each for the Q output and the R output. Additionally, the Q output needs an incrementer for when in the Divide state, and the R output needs both a subtractor for when in the Divide state and a comparator for the transitions components needed are two comparators, as gathered from the state transitions from the Divide state. The datapaths for outputs Q and R are shown below respectively.

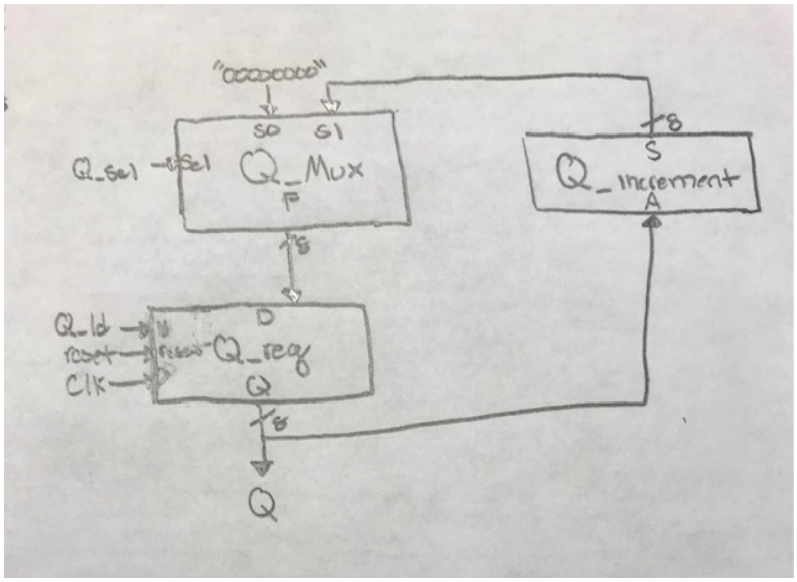

Figure 5: Q Output Datapath

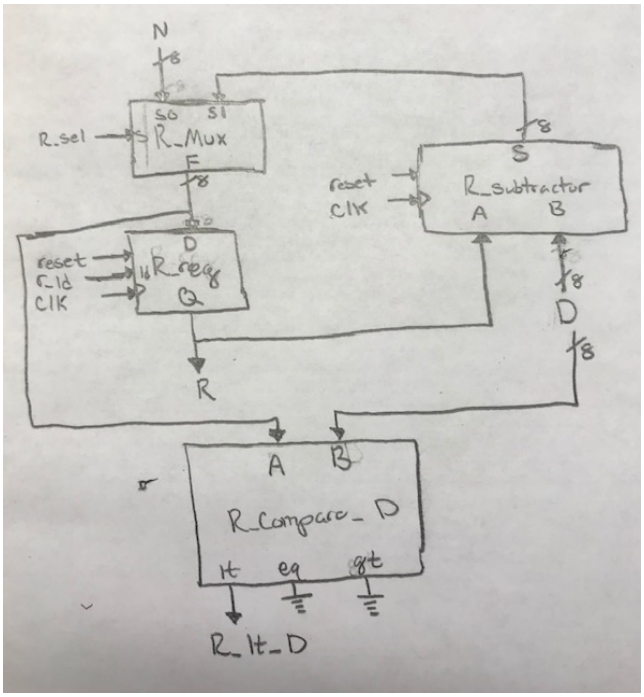

Figure 6: R Output Datapath

The overall structure of the Q output datapath is straightforward, with the 2x1 Mux selecting between a value of zero or incrementation of Q, the incrementer to provide the

incrementation, and the load register to update the value of Q. The connection between the load register and the incrementator does not need to be changed, as the one-clock-cycle delay that occurs with variable registers does not impact the variable assignments of this datapath. The structure for the R output datapath is slightly more complex, with the 2x1 Mux selecting between the value of N and the subtraction of D from R, the subtractor to provide said subtraction, a comparator and a load register to update the value of R. In this case, the comparator takes the input of R from the 2x1 Mux instead of the load register as it needs to avoid the one-clock-cycle delay from the load register for accurate results. Outside of these two datapaths, two more components are needed to make the full datapath comprehensive. These components are both comparators, which are needed for the transition from the Init state to the Zero state and for the transition from the Init state to the Fin1 state, which are shown below respectively.

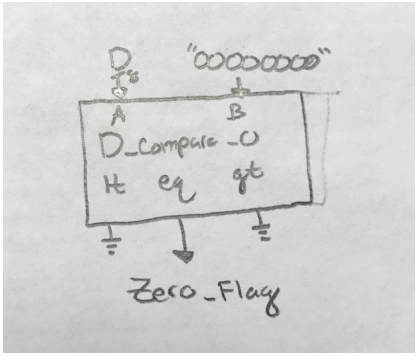

Figure 7: Zero State Comparator

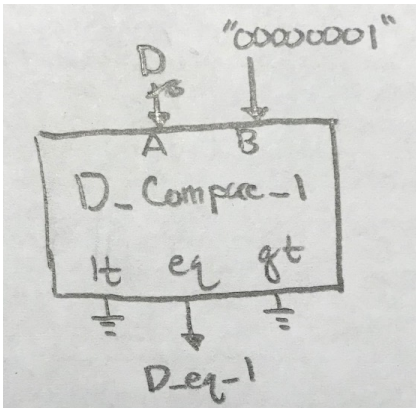

Figure 8: Fin1 State Comparator

As seen in Figures 5 and 6, there are two 8-bit registers in the datapath. These registers are the only examples of latency present in the algorithm. They are not directly connected to each other, so the maximum latency is only one clock cycle.

SUMMARY

The design process for modelling the repeated subtraction algorithm in VHDL required several steps. The first was to create the algorithm as a HLSM, followed by simulating it in VHDL code. These steps illustrated some of the major flaws of the repeated subtraction algorithm. This project modified the algorithm to address a few of its most glaring inadequacies by introducing new states. Next, the datapath and control were extracted. This project skipped defining the control and status signals, and instead immediately modeled the datapath in VHDL. The control signals could then be easily defined and the FSM successfully extracted. The FSM was simple to model in VHDL code, along with the top level structure. The final simulation of the top level structure was plagued by simple syntax errors. Following a session of thorough debugging, the structure functioned perfectly.

A notable advantage of this algorithm is that there is only a single clock cycle of latency, which is also the minimum amount required if any data is meant to be stored. In other words, the latency value is the most optimal option. However, it is unknown how this compares to other algorithms. What is known is that the repeated subtraction algorithm is significantly slower than other dividing algorithms. The reasoning behind this is that this particular algorithm is linear in size, whereas many others are constant. This means that those with constant size run much faster overall. Because this algorithm runs slower than most, not all test cases for the second testbench could be added. This means that after testing 174 different numerator values from 0 to 173, the maximum running time for the EDA playground was reached. Thus, not all 256 numerator values for the denominator value of 5 could be tested. However, as mentioned before, linearity comes with its advantages as well, including easier implementation and understanding of the algorithm.

APPENDIX

Citation:

[1] S. Pauli, “MAT 112 Ancient and Contemporary Mathematics,” *Division Algorithm*. [Online]. Available: https://mathstats.uncg.edu/sites/pauli/112/HTML/secdivalg.html. [Accessed: 06-Jun-2021].

EDA Playground:

[2] https://www.edaplayground.com/x/mKL5