

EE 371 Lab 1

Sharyar Khalid <15727978>

Katie Neff <1168464>

Adolfo Pineda <1231310>

September 19, 2016

Contents

1	INTRODUCTION	1
2	DISCUSSION	1
2.1	Design Specification	1
2.1.1	Four Counters	1
2.1.2	C Program	2
2.2	Design Procedure	2
2.2.1	Counter 1	2
2.2.2	Counter 2	2
2.2.3	Counter 3	2
2.2.4	Counter 4	3
2.2.5	C Program	3
3	TESTING	4
3.1	Testing Strategy	4
3.1.1	Counters	4
3.1.2	C Program	4
3.2	Expected Results	4
3.2.1	Counters	4
3.2.2	C Program	5
4	RESULTS	5
4.1	Counter 1	5
4.2	Counter 2	6
4.3	Counter 3	6
4.4	Counter 4	7
4.5	C Program	7
5	ERROR ANALYSIS	8
5.1	Counter 1	8
5.1.1	Stuck At zero	8
5.1.2	Stuck At one	8
5.2	Counter 2	9
5.2.1	Stuck At zero	9
5.2.2	Stuck At one	9
5.3	Counter 3	10
5.3.1	Stuck At zero	10
5.3.2	Stuck At one	10
6	CONCLUSION	10
7	APPENDICES	11
7.1	Division of Work	11
7.2	Figures	11

List of Figures

2.1	Top level schematic for Counter 1	2
2.2	Block diagram design created in Quartus II software	3
4.1	Test bench results from Counter 1.	5
4.2	Test bench results from Counter 2.	6
4.3	Test bench results from Counter 3.	6
4.4	Test bench results from Counter 4.	7
4.5	Currency calculator program results	7
7.1	GTK Wave for Counter 1 (four bit ripple down counter).	11
7.2	GTK Wave for Counter 2 (four bit synchronous down counter, designed using a dataflow model).	12
7.3	GTK Wave for Counter 3 (four bit Johnson counter).	12
7.4	GTK Wave for Counter 4 (four bit synchronous down counter, designed using schematic level entry).	13
7.5	Signal Tap in Quartus during simulation for Counter 1.	13
7.6	Signal Tap in Quartus during simulation for Counter 2.	14
7.7	Signal Tap in Quartus during simulation for Counter 3.	14
7.8	Signal Tap in Quartus during simulation for Counter 4.	15

List of Tables

7.1	Division of work among group members	11
-----	--	----

List of Listings

1	D Flip Flop design used in counters 1 and 2.	1
2	Top level testbench for all counters	15
3	Clock divider for modeling on the SoC board	16
4	Top level module for modeling on the SoC board	17
5	Counter 1 Verilog Code	17
6	Counter 2 Verilog code	18
7	Counter 3 Verilog code	18
8	Counter 4 Verilog code generated by Quartus II from the block diagram	19
9	Currency Calculator C Program	22

1 INTRODUCTION

For this lab, we were to design and program four different counters based on different design models: gate, dataflow, behavioral and schematic models. We designed these counters using Verilog and Quartus II software. We created testbench models for each counter to test there output, and we programmed each onto a DE1-SoC board. Additionally, we designed a C program that converted dollar amounts to other types of currencies based on an exchange rate.

2 DISCUSSION

2.1 Design Specification

In this section we will describe the design specifications for both the four Verilog counters and the currency exchange program written in C.

2.1.1 Four Counters

Each of the four counters specified required a different model. The specification for each counter is as follows:

- Counter 1: A four stage ripple down counter, with active low reset, designed using a gate model. This counter should incorporate the DFlipFlop design specified in Listing 1.
- Counter 2: A four stage synchronous down counter, with active low reset, designed using a data flow model. This counter should incorporate the DFlipFlop design specified in Listing 1.
- Counter 3: A four stage synchronous Johnson down counter, with active low reset, designed using a behavioral model.
- Counter 4: A four stage synchronous down counter, with active low reset, designed using schematic entry.

Listing 1: D Flip Flop design used in counters 1 and 2.

```
module DFlipFlop(q, qBar, D, clk, rst);
    input D, clk, rst;
    output q, qBar;
    reg q;
    not n1 (qBar, q);
    always@ (negedge rst or posedge clk)
    begin
        if(!rst)
            q = 0;
        else
            q = D;
    end
endmodule
```

2.1.2 C Program

A currency calculator that allows a user to convert US currency into foreign currency and vice versa.

2.2 Design Procedure

Our four counters were first written in Verilog or designed in Quartus II. Then they were downloaded onto the De1-SoC board. On the SoC board, all of the counters incorporated the clock divider and top level code found in listings 3 and 4 in Section 7.2. The red LEDs 0 - 3 were used to model the bits of the counters, and SW9 was used as the reset (active low).

2.2.1 Counter 1

Counter 1 was designed using a gate model. First we designed out top level schematic using D flipflops as specified. Figure 2.1 shows our final design.

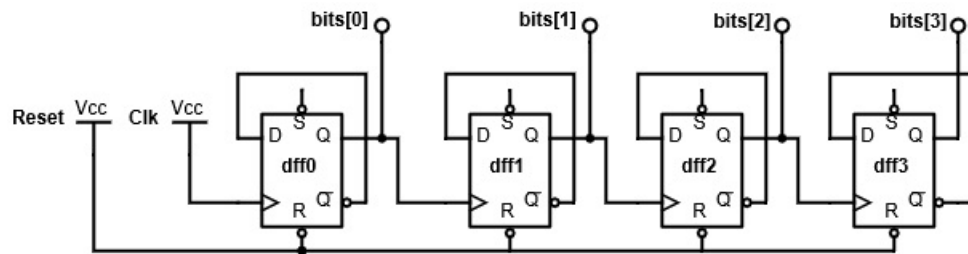


Figure 2.1: Top level schematic for Counter 1

We translated this to the Verilog code found in Listing 5 in Section 7.2.

2.2.2 Counter 2

Counter 2 was specified to be designed using the dataflow model. To do this, we had to figure out logic equations to relate previous state (PS) to next state (NS). We drew out the truth table for the previous and next state of each bit. We then simplified the expressions using K-maps. Our final logic expressions are defined in

```
NS0 = ~PS0
NS1 = PS0 & PS1 | ~PS0 & ~ PS1
NS2 = PS2 & PS0 | PS2 & PS1 | ~PS2 & ~PS1 & ~PS0
NS3 = PS3 & PS2 | PS3 & PS1 | PS3 & PS0 | ~PS3 & ~PS2 & ~PS1 & ~PS0
```

We incorporated these stated with four D flipflops to make the counter. The final Verilog code is found in Listing 6 in Section 7.2.

2.2.3 Counter 3

Counter 3 was specified as a Johnson counter designed using the behavioral model. In Verilog, this means using `assign` statements and specifying the systems behavior at every step. The

behavior of a Johnson counter is quite simple. At every clock cycle, the present bit takes the value of the next bit, except for the least significant bit. This bit takes that inverse value of the most significant bit. Additionally, the bits must be set to undefined when reset is low. We were able to model this behavior in an `always` statement. The final Verilog code is found in Listing 7 in Section 7.2.

2.2.4 Counter 4

Counter 4 was specified to be a synchronous down counter, just like Counter 2. However, we were specified to design it using block diagrams instead. In Quartus, we modeled the system with D flipflops and logic gates. We had already figured out logic equations for previous state and next state for this type of counter when we wrote Counter 2. We then translated these equations into a schematic using logic gates. Our final block diagram is shown in Figure 2.2.

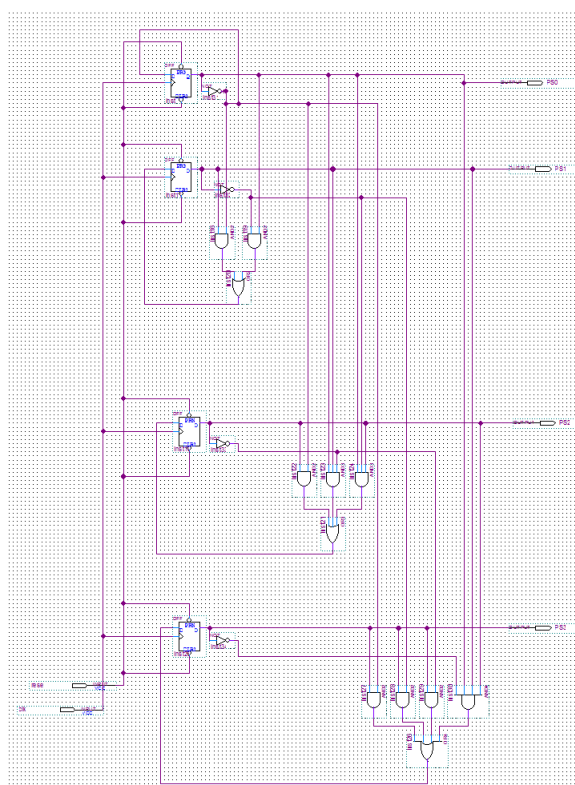


Figure 2.2: Block diagram design created in Quartus II software

For testing purposes, we used a program provided by Quartus II to translate this block diagram to Verilog HDL code. This code is found in Listing 8 in Section 7.2.

2.2.5 C Program

This program is required to have the following functionalities: specifying the exchange rate, converting an amount in dollars into the foreign currency, and converting an amount of the foreign currency into dollars. We decided to allow the user to specify the exchange rate and the currency amount by using `scanf` statements in our program, which prompted the user for these values.

3 TESTING

3.1 Testing Strategy

3.1.1 Counters

Our testing strategy for the counters involved a two pronged approach. Our first approach was using Icarus Verilog and writing a testbench program that would provide outputs at each clock cycle and a waveform in GTKWave. Our second approach was downloading the counters to the DE1-SoC board and observing the behavior using the Signal Tap program to make distinct observations.

Our test bench code is found in Listing 2 in Section 7.2. The test bench was designed to test every possible state in the counter, as well as the reset functionality. The tester starts with all inputs undefined. Next, reset is pulled high and the clock iterated through four cycles. Then reset is pulled low. If reset is functional, the output counter should reset to its starting value. Next, reset is pulled high again and the clock iterates through 17 cycles. The first 16 cycles are to test the output of the counter, and the last cycles is to test the "wrap around" functionality (if the counter restarts at it's original value when it's through counting). All counters were run through this test and the outputs and GTKWaves were observed for correctness.

Next we used the SoC board to test the counters. Each counter was downloaded onto the board and the reset and counting functionality were observed. We started by switching reset high (switch 9) and letting the counter run for a few clock cycles. We pulled reset low and observed if the LED outputs were undefined. Then we pulled reset high again to see if the counter started at the correct beginning state. Next, we let the counter run for a number of clock cycles to observe the counting and wrap around functionality. After this general observation, we used Signal Tap to model the output waveforms in real time.

3.1.2 C Program

Our group designed the currency calculator to allow the user to input any numeric value when prompted for one. This means that the currency amount and the exchange rate can be as large as the user chooses it to be, and the program will correctly convert it. It will also correctly convert negative values, but it seems unlikely that a user would like to convert a negative currency amount (but they could if they wanted to). And finally, when asked for character input (i.e. would you like to convert another value), we decided to let the user to have a second, third, or as many chances as necessary for them to either input a 'y' for yes or a 'n' for no.

3.2 Expected Results

3.2.1 Counters

The expected results of our counters are similar for each one. For counters 1, 2 and 4 who model down counters, the outputs bits should be undefined when reset is low. When reset is high, the counters should first output 0000. Then on the next clock cycle, they will jump to 1111 (15) and decrement by 1 every subsequent clock cycle. Once they reach zero, they should "wrap around" back to 1111 on the next clock cycle and start decrementing once again.

3.2.2 C Program

Taking into consideration of all of the possible inputs the user can put in, we expect the program to correctly execute its expected functionalities as long as the user follows the instructions.

4 RESULTS

4.1 Counter 1

The test bench program gave the output found in Figure 4.1 below.

bits	rst	clk	bits	rst	clk	bits	rst	clk
xxxx	x	x	1110	1	1	0110	1	1
0000	0	0	1110	1	0	0110	1	0
0000	0	1	1101	1	1	0101	1	1
0000	1	0	1101	1	0	0101	1	0
1111	1	1	1100	1	1	0100	1	1
1111	1	0	1100	1	0	0100	1	0
1110	1	1	1011	1	1	0011	1	1
1110	1	0	1011	1	0	0011	1	0
1101	1	1	1010	1	1	0010	1	1
1101	1	0	1010	1	0	0010	1	0
1100	1	1	1001	1	1	0001	1	1
0000	0	0	1001	1	0	0001	1	0
0000	0	1	1000	1	1	0000	1	1
0000	1	0	1000	1	0	0000	1	0
1111	1	1	0111	1	1	1111	1	1
1111	1	0	0111	1	0	1111	1	0

Figure 4.1: Test bench results from Counter 1.

Additionally, this step produced the GTKWave found in Figure 7.1. Both these outputs appear to follow the expected outputs specified in Section 3.2.1.

Additionally, we tested in Quartus using Signal Tap. The results of this test are shown in Figure 7.5. This waveform is similar to the GTKWaveform and is consistent with the expected output.

4.2 Counter 2

bits	rst	clk	bits	rst	clk	bits	rst	clk
xxxx	x	x	1110	1	1	0110	1	1
0000	0	0	1110	1	0	0110	1	0
0000	0	1	1101	1	1	0101	1	1
0000	1	0	1101	1	0	0101	1	0
1111	1	1	1100	1	1	0100	1	1
1111	1	0	1100	1	0	0100	1	0
1110	1	1	1011	1	1	0011	1	1
1110	1	0	1011	1	0	0011	1	0
1101	1	1	1010	1	1	0010	1	1
1101	1	0	1010	1	0	0010	1	0
1100	1	1	1001	1	1	0001	1	1
0000	0	0	1001	1	0	0001	1	0
0000	0	1	1000	1	1	0000	1	1
0000	1	0	1000	1	0	0000	1	0
1111	1	1	0111	1	1	1111	1	1
1111	1	0	0111	1	0	1111	1	0

Figure 4.2: Test bench results from Counter 2.

4.3 Counter 3

bits	rst	clk	bits	rst	clk	bits	rst	clk	bits	rst	clk
xxxx	x	x	1000	1	1	0000	1	1	0001	1	1
xxxx	0	0	1000	1	0	0000	1	0	0001	1	0
0000	0	1	1100	1	1	1000	1	1	0000	1	1
0000	1	0	1100	1	0	1000	1	0	0000	1	0
1000	1	1	1110	1	1	1100	1	1	1000	1	1
1000	1	0	1110	1	0	1100	1	0	1000	1	0
1100	1	1	1111	1	1	1110	1	1			
1100	1	0	1111	1	0	1110	1	0			
1110	1	1	0111	1	1	1111	1	1			
1110	1	0	0111	1	0	1111	1	0			
1111	1	1	0011	1	1	0111	1	1			
1111	0	0	0011	1	0	0111	1	0			
0000	0	1	0001	1	1	0011	1	1			
0000	1	0	0001	1	0	0011	1	0			

Figure 4.3: Test bench results from Counter 3.

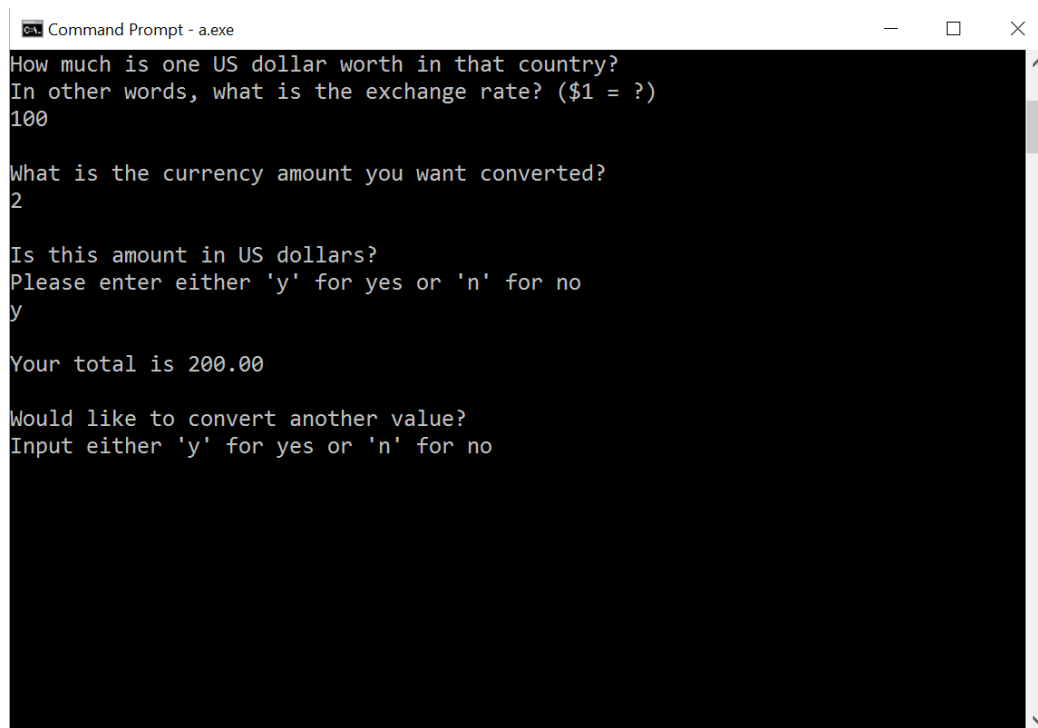
4.4 Counter 4

bits	clk	rst	bits	clk	rst	bits	clk	rst
xxxx	x	x	1101	1	1	0101	1	1
1111	0	0	1101	0	1	0101	0	1
1111	1	0	1100	1	1	0100	1	1
1111	0	1	1100	0	1	0100	0	1
1110	1	1	1011	1	1	0011	1	1
1110	0	1	1011	0	1	0011	0	1
1101	1	1	1010	1	1	0010	1	1
1101	0	1	1010	0	1	0010	0	1
1100	1	1	1001	1	1	0001	1	1
1100	0	1	1001	0	1	0001	0	1
1011	1	1	1000	1	1	0000	1	1
1111	0	0	1000	0	1	0000	0	1
1111	1	0	0111	1	1	1111	1	1
1111	0	1	0111	0	1	1111	0	1
1110	1	1	0110	1	1	1110	1	1
1110	0	1	0110	0	1	1110	0	1

Figure 4.4: Test bench results from Counter 4.

4.5 C Program

Results are shown in Figure 4.5.



```
Command Prompt - a.exe
How much is one US dollar worth in that country?
In other words, what is the exchange rate? ($1 = ?)
100

What is the currency amount you want converted?
2

Is this amount in US dollars?
Please enter either 'y' for yes or 'n' for no
y

Your total is 200.00

Would like to convert another value?
Input either 'y' for yes or 'n' for no
```

Figure 4.5: Currency calculator program results

5 ERROR ANALYSIS

5.1 Counter 1

Counter 1 has two inputs (`clk` and `rst`) and one output (the bits). An important distinction of this counter is that each flip flop after the first flip flop is clocked by the output of the previous flip flop. The first flip flop is clocked by a `clk`.

5.1.1 Stuck At zero

If `rst` is stuck at zero, the output will always be undefined. This would be a fatal error because the counter would never output anything and would not follow any specifications for output. This is likely to happen during the test bench, if we forget to pull `rst` high while simulating, or on the SoC board, if the switch we choose is broken or never pulled high.

If `clk` is stuck at zero, this will cause the system to be stuck in its current state. Since the first D flip flop is clocked by `clk`, and the D flip flop model we use uses an `always @(posedge clk)` statement to change states, the state would never change because there would never be a positive edge on the clock. The rest of the flip flops are clocked by the output of the first flip flop. This output would be stuck in its current state, and none of the other flip flops would be clocked. This would be a fatal error, as the module would never change states and would never decrement by one every clock cycle as specified.

The output bits are wired to the outputs of each flip flop. If any of these outputs are stuck at zero, then the following flip flops will never be clocked, and the state will never change. This would be cause a similar result as the one caused if `clk` is stuck at zero. This would be a fatal error as the module would be stuck in its present state and would never decrement by one as specified.

However, this is an impossible state. Since the input to each flip flop is wired to its own inverse output, the outputs would never be stuck at zero.

5.1.2 Stuck At one

If `rst` is stuck at one, the system would run without trouble and would produce the expected output. However, there would be no way to reset the system, and the specification required a means to reset. Therefore, this error would not be fatal to the system but it would need to be addressed in order to meet the system requirements.

If `clk` is stuck at one, there would be a similar result to the one caused if `clk` is stuck at zero. The `always @(posedge clk)` in the first D flip flop would never be triggered and its state would never change. Since the output of the first D flip flop clocks the next D flip flop, this flip flop would also be stuck in its current state and would be unable to clock the next D flip flop. None of the flip flops in the series would be clocked and the entire system would be stuck in its current state. This would be a fatal error to the system as the output would never decrement by one as required.

If any of the outputs wired to the outputs of each D flip flop are stuck at one, then a similar problem to the one described above is produced. Each of the D flip flops after the first one is

clocked by the output of the previous, so if this output is stuck at one, the `always @(posedge clk)` block in each D flip flop is never triggered and the state never changes. This would cause the entire system to be stuck in its current state.

However, this state would be impossible. Since the input of the each D flip flop is wired to its own inverse output, the output of every flip flop would never be stuck at one.

5.2 Counter 2

Counter 2 has two inputs (`clk` and `rst`) and one output (the bits), just like counter 1. Also like counter 1, it uses four d flip-flops to help do the counting. However, this counter uses synchronized clocks and dataflow logic (assign statements) to count down by one.

5.2.1 Stuck At zero

If `rst` is stuck at zero, the output will always be undefined since this counter uses active low reset to do its counting. Counter 2 would never begin counting in this case, and this is a possible reality if we were to leave the `rst` variable at zero in our test bench, or the `rst`-defined switch at zero on the DE1-SoC board.

If `clk` is stuck at zero, the value of the counter would never change since the d flip-flops it uses change the values when there is a positive edge (`always @(posedge clk)`). Because the counter is never decremented by one in this case, this would be considered a fatal error since it does not count down by one.

If the output bits were to be stuck at zero, then the counter would not function like it is supposed to (obviously). The output bits are assigned to a variable called `PS`, and this variable is manipulated by using logic to emulate a synchronous down counter. Since the output bits will always be at zero, the `PS` variable will also always be at zero.

5.2.2 Stuck At one

If `rst` is stuck at one, then we would have no problems if we were to want counter 2 to forever count down by one. However, we would never be able to reset the counter, meaning that we can only start counting from the initial state once and only once.

If `clk` is stuck at one, then, similar to `clk` being stuck at zero, it will not count correctly. This counter requires d flip-flops for outputs, outputs that depend on the positive edge of the clock (`always @(posedge clk)`). Since it will have only one positive edge, then the counter will count down once and be stuck at that state for the remainder of its run.

If the output is stuck at one, then, due to how we implemented the logic for the output bits, it would not change value at all. The output is assigned to a variable called `PS` which only uses AND and OR logic for computation, never NOT. Because of this, the output bits will always be stuck at one, something we do not want counter 2 to do.

5.3 Counter 3

Counter 3 has two inputs (`clk` and `rst`) and one output (the bits) just like the previous two counters. For this counter, called the Johnson down counter, the most significant bit depends on the least significant bit to go through a NOT gate, and then that bit is passed through the rest of the bits.

5.3.1 Stuck At zero

If `rst`, defined by SW[9], is stuck at zero, the output would always be a 4 bit number 0, since the reset is defined as active low in this counter. If this program was mapped on to the DE1-SoC board and the reset was set to 0 all of the LEDs defined in the program would be turned off, the counter itself would never begin.

If `clk` is stuck at zero, the counter would never change states since it is clocked by `clk` and the counter only changes states when it sees the positive edge of `clk`. If this was simulated in a gtk wave or model sim, there would just be a straight line and no edges, because of which the behavioral model logic for the Johnson counter would not have anything to react to.

If any one of the bits are stuck at zero, then all the bits will eventually go to zero since the output of one bit is assigned to the other. However, unless the `rst` is set to zero or, in the case of DE1-Soc board, SW[9] is set to zero, it is highly unlikely that this case would ever happen since the output from the least significant bit is connected to an inverter. Any time all of the bits are being changed to zero (since one of the bits will be stuck on zero), by the time it gets to the least significant bit, that zero will be flipped to one due to the inverter, and will then be passed on as one to the most significant bit.

5.3.2 Stuck At one

If `rst` is stuck at one, there wouldn't be an issue in terms of the counter being stuck in one state. However, there would not be a way to set the counter back to its initial state. Also, whenever we turn on the DE1-SoC board, we would always be facing a challenge of not knowing which state the bits are going to start from.

If `clk` is stuck at 1, we would face similar challenges as in the case when `clk` is stuck at 0, the bits will not have anything to react to and, whichever state the bits are in, they would stay stuck in that state.

If the output is stuck at one, then a similar thing would happen as in the case with the outputs stuck at zero. If any one of the bits get stuck at one, then the bit will trickle down to all of the other bits being stuck at one. However, as mentioned above, it is highly unlikely since the inverter on the end will end up flipping that zero bit to one and all of the other bits will quickly get out of that state.

6 CONCLUSION

In conclusion, the lab went extremely smoothly, and work was delegated equally to meet all requirements mentioned in the design specification. As a team we were able to help each other

understand key components and teach one another whenever we were stuck. There were some concepts that were new to all of us, but using online tutorials and documentation found online, we were able to overcome all challenges and successfully build and test our lab components. We learnt each other's strengths and weaknesses and look forward to the future labs.

7 APPENDICES

7.1 Division of Work

Table 7.1: Division of work among group members

Name	Tasks	Hours
Katie Neff	Wrote Verilog modules, wrote test bench,	
Adolfo Pineda	Wrote and tested C program	
Sharyar Khalid	Debugged, improved, and annotated C program	5

7.2 Figures

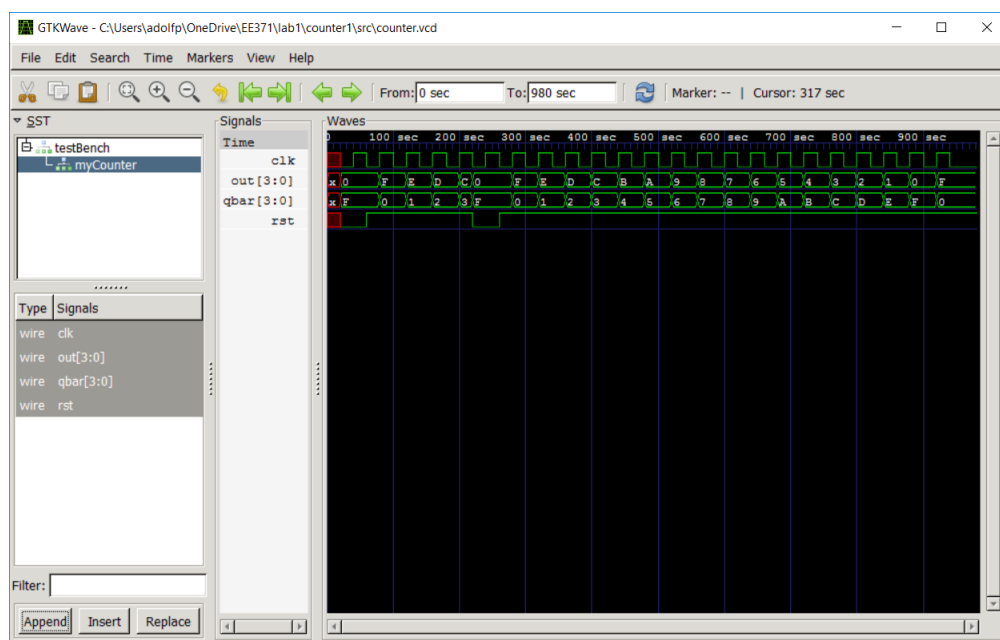


Figure 7.1: GTK Wave for Counter 1 (four bit ripple down counter).

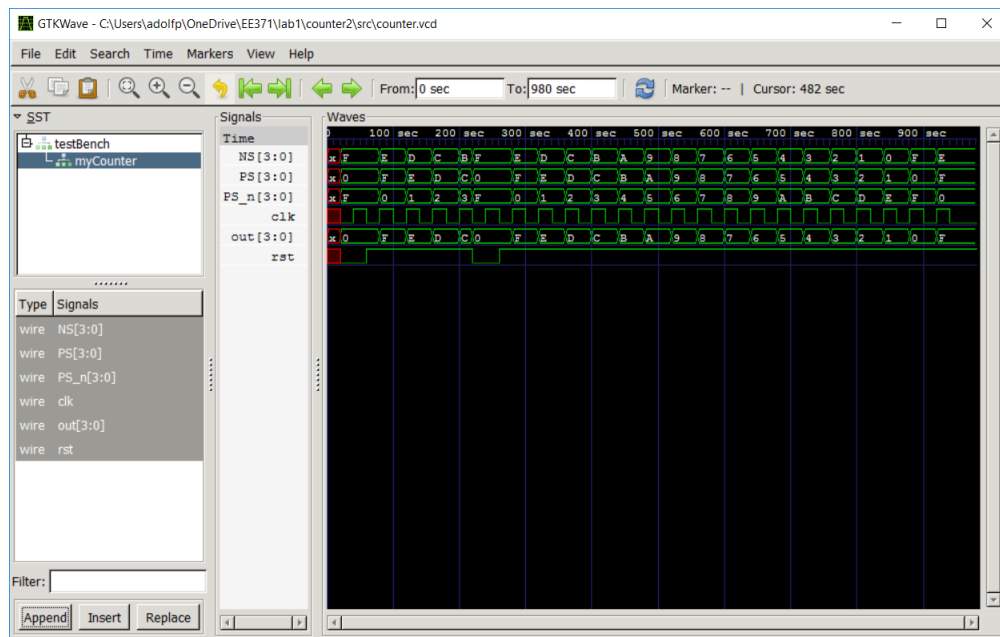


Figure 7.2: GTK Wave for Counter 2 (four bit synchronous down counter, designed using a dataflow model).

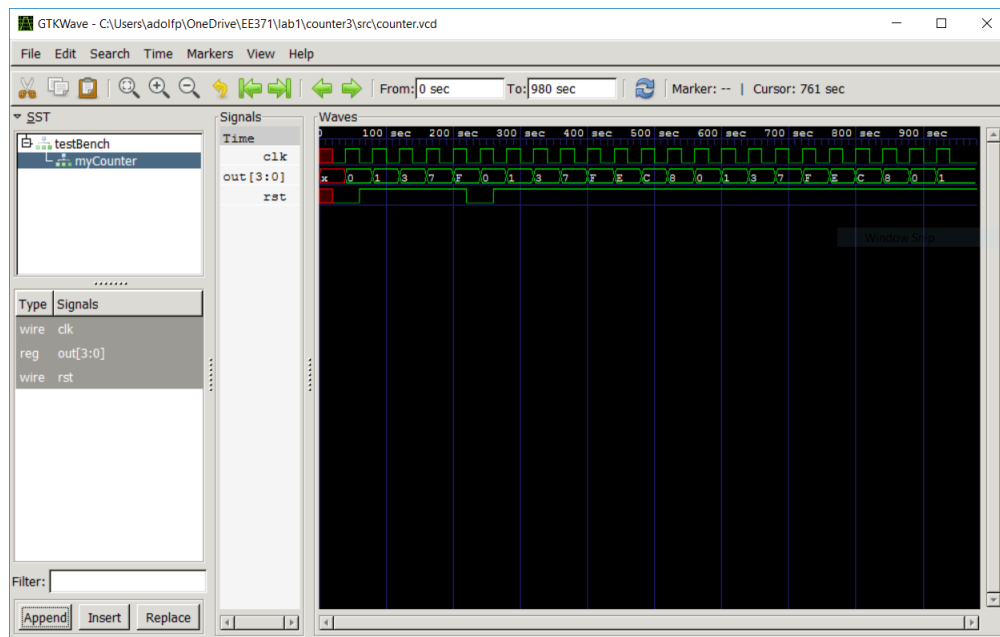


Figure 7.3: GTK Wave for COunter 3 (four bit Johnson counter).

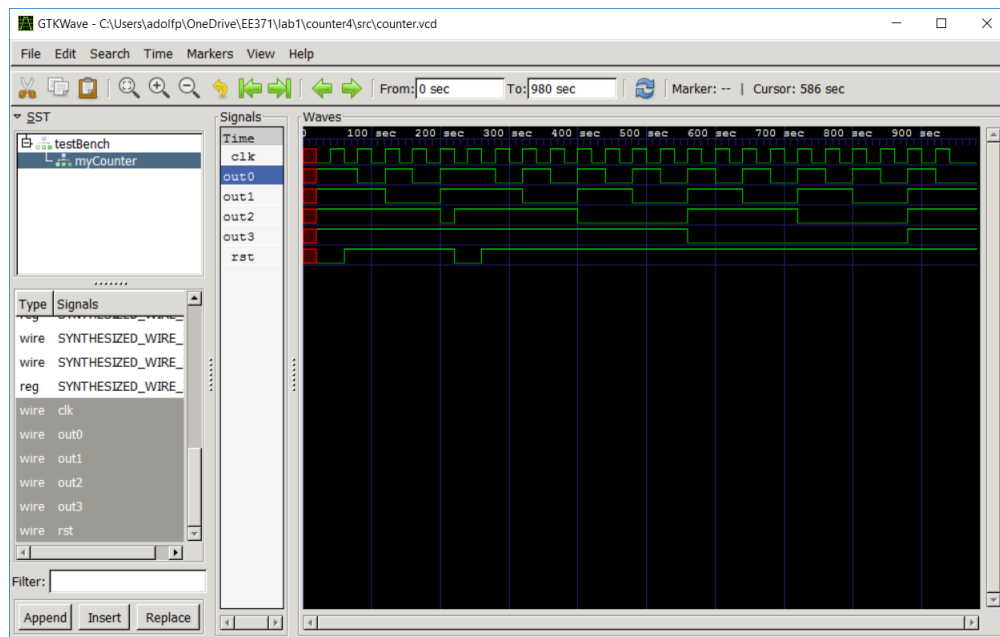


Figure 7.4: GTK Wave for Counter 4 (four bit synchronous down counter, designed using schematic level entry).

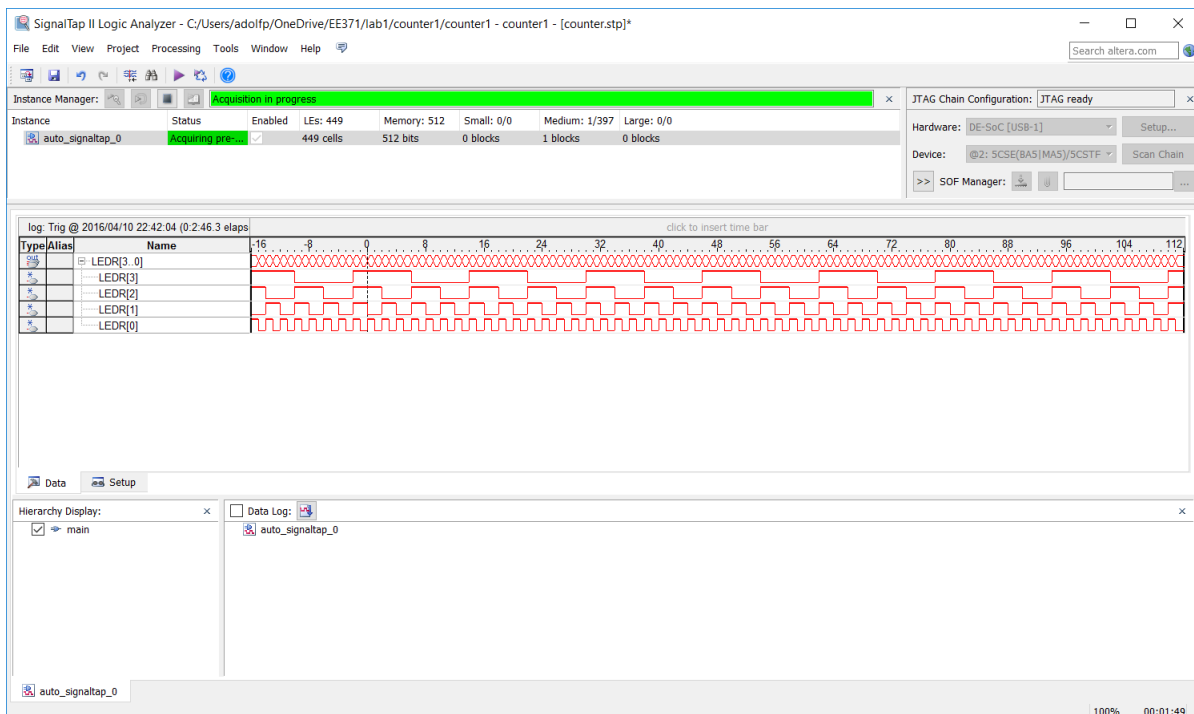


Figure 7.5: Signal Tap in Quartus during simulation for Counter 1.

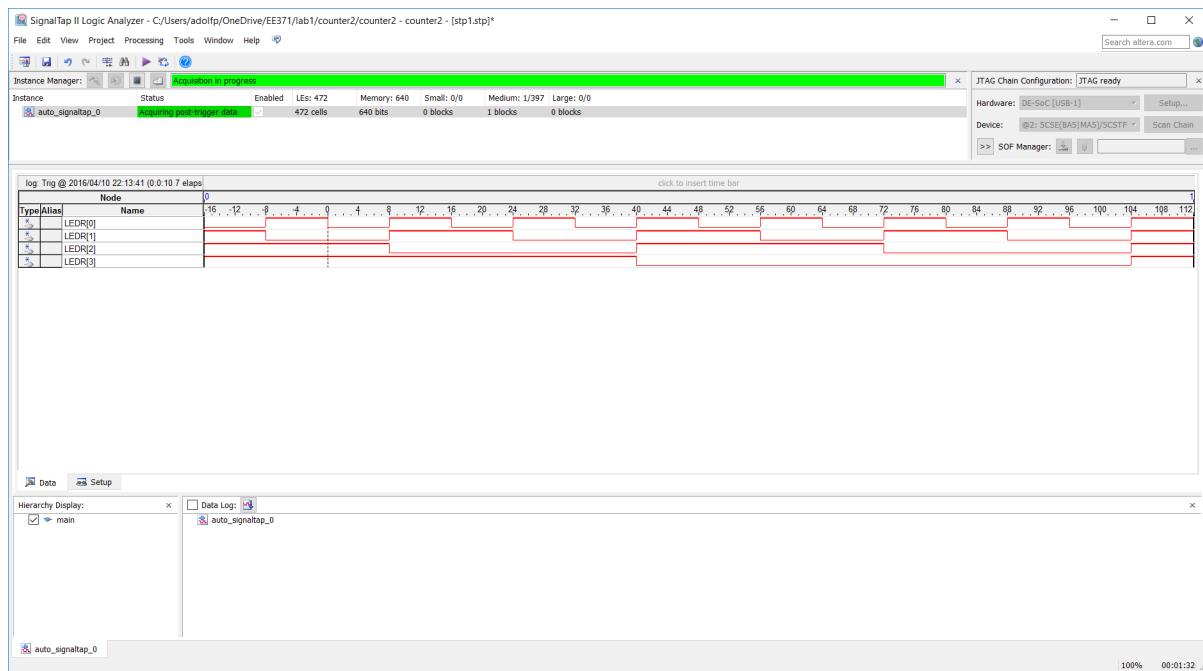


Figure 7.6: Signal Tap in Quartus during simulation for Counter 2.

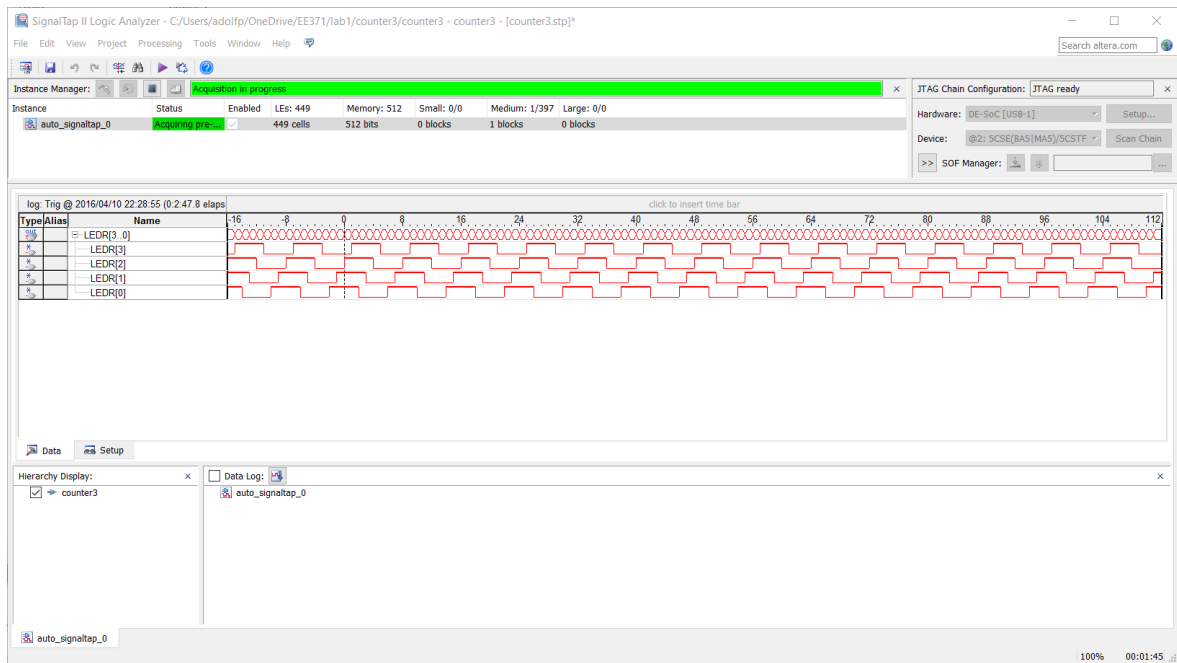


Figure 7.7: Signal Tap in Quartus during simulation for Counter 3.

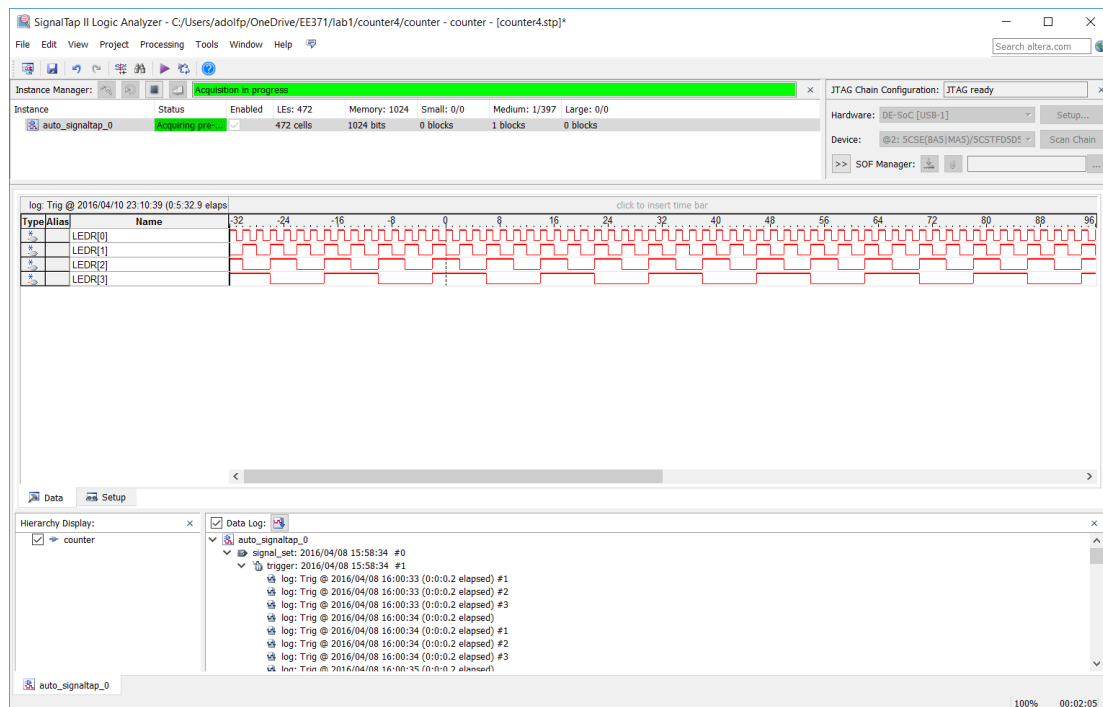


Figure 7.8: Signal Tap in Quartus during simulation for Counter 4.

Listing 2: Top level testbench for all counters

```

/*
Author:  Katie Neff
Title:   counterTop.v
Summary: Test bench module that drives the tests for a counter
*/

module testBench;

    wire [3:0] out;
    wire clk, rst;

    // Declare an instance of the counter module
    counter myCounter(out[3:0], clk, rst);

    // Declare an instance of the Tester module
    Tester aTester(out[3:0], clk, rst);

    // File for gtkwave
    initial
    begin
        $dumpfile("counter.vcd");
        $dumpvars(1, myCounter);
    end

endmodule

```

```

/*
Author:  Katie Neff
Title:   Tester
Summary: Testing module for the counter to show what it is counting.
*/

module Tester(out, clk, rst);
    input [3:0] out; // 4-bit data of the counter as input
    output reg clk, rst; // Clock and Reset for the counters

    parameter stimDelay = 20;

    initial
    begin
        // Displays the data and labels for the counter
        $display("\t\t bits \t rst \t clk");
        $monitor("\t\t %b\t %b\t %b", out, rst, clk);
    end

    integer i;
    initial
    begin
        #stimDelay;
        rst = 'b0; clk = 'b0;
        #stimDelay clk = 'b1;
        #stimDelay clk = 'b0; rst = 'b1;

        // 4 iterations to test reset
        for (i = 0; i < 4; i = i + 1) begin
            #stimDelay clk = 'b1;
            #stimDelay clk = 'b0;
        end
        rst = 'b0;
        #stimDelay clk = 'b1;
        #stimDelay clk = 'b0; rst = 'b1;

        // 16 iterations to test counter
        for (i=0; i < 16; i = i + 1) begin
            #stimDelay clk = 'b1;
            #stimDelay clk = 'b0;
        end

        #stimDelay clk = 'b1;
        #stimDelay clk = 'b0;

        #(2*stimDelay);
        $finish;
    end

endmodule

```

Listing 3: Clock divider for modeling on the SoC board

```

% * <neffk9@gmail.com> 2016-04-14T21:56:07.525Z:
%

```

```

% ^.
% * <neffk9@gmail.com> 2016-04-14T21:56:05.902Z:
%
% ^.
/*
Author:  Katie Neff
Title:   clockdiv.v
Summary: This divides the clock to slow it down. divided_clocks[0] = 25MHz,
        [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz,
        ...
*/

module clock_divider (clk, divided_clocks);
    input clk; // Clock that is passed in
    output reg [31:0] divided_clocks; // Outputs a slower clock

    initial
        divided_clocks = 0;

    // Division logic
    always @(posedge clk)
        divided_clocks = divided_clocks + 1;

endmodule

```

Listing 4: Top level module for modeling on the SoC board

```

/*
Author:  Katie Neff
Title:   main
Summary: Main code for DEC-Sol board implementation
        that shows the counter in action
*/

module main(LED_R, CLOCK_50, SW);
    input [9:9] SW; // Switch for reset
    input CLOCK_50; // 50 MHz clock
    output wire [3:0] LED_R; // LEDs for displaying the counter

    wire [31:0] clk_out; // For clock division

    clock_divider clockdiv(CLOCK_50, clk_out); // Divides the 50 MHz clock
        for slower clocks
    counter cnt(LED_R, clk_out[24], SW); // Starts the counting using the
        divided clock

endmodule

```

Listing 5: Counter 1 Verilog Code

```

/*
Author:  Katie Neff
Title:   counter.v
Summary: 4-bit ripple down counter with active low reset
        that uses gate model logic.

```

```

*/

module counter(out, clk, rst);
    output [3:0] out;
    input clk, rst;
    wire [3:0] qbar;

    // D flip-flops connected in such a way that emulates the
    // 4-bit ripple down counter
    DFlipFlop dff0(out[0], qbar[0], qbar[0], clk, rst);
    DFlipFlop dff1(out[1], qbar[1], qbar[1], out[0], rst);
    DFlipFlop dff2(out[2], qbar[2], qbar[2], out[1], rst);
    DFlipFlop dff3(out[3], qbar[3], qbar[3], out[2], rst);

endmodule

```

Listing 6: Counter 2 Verilog code

```

/*
Author:  Katie Neff
Title:   counter
Summary: 4-bit synchronous down counter with active low reset
         that uses dataflow model logic.
*/

module counter(out, clk, rst);
    output wire [3:0] out;
    input clk, rst;
    wire [3:0] PS, NS, PS_n;

    // Synchronized d flip-flops that do the counting
    DFlipFlop dff0(PS[0], PS_n[0], NS[0], clk, rst);
    DFlipFlop dff1(PS[1], PS_n[1], NS[1], clk, rst);
    DFlipFlop dff2(PS[2], PS_n[2], NS[2], clk, rst);
    DFlipFlop dff3(PS[3], PS_n[3], NS[3], clk, rst);

    // Counter logic
    assign out = PS;
    assign NS[0] = PS_n[0];
    assign NS[1] = PS[0] & PS[1] | PS_n[0] & PS_n[1];
    assign NS[2] = PS[2] & PS[0] | PS[2] & PS[1] | PS_n[2] & PS_n[1] &
        PS_n[0];
    assign NS[3] = PS[3] & PS[2] | PS[3] & PS[1] | PS[3] & PS[0] | PS_n[3]
        & PS_n[2] & PS_n[1] & PS_n[0];

endmodule

```

Listing 7: Counter 3 Verilog code

```

/*
Author:  Katie Neff
Title:   counter
Summary: 4-bit synchronous Johnson down counter with active low reset
         that uses behavioural model logic.
*/

```

```

module counter(out, clk, rst);
    output reg [3:0] out;
    input clk, rst;

    // Counter logic that emulates a Johnson down counter
    always @(posedge clk) begin
        if (!rst) begin
            out = 4'b0000;
        end else begin
            out[0] <= ~out[3];
            out[1] <= out[0];
            out[2] <= out[1];
            out[3] <= out[2];
        end
    end
endmodule

```

Listing 8: Counter 4 Verilog code generated by Quartus II from the block diagram

```

// Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, the Altera Quartus II License Agreement,
// the Altera MegaCore Function License Agreement, or other
// applicable license agreement, including, without limitation,
// that your use is for the sole purpose of programming logic
// devices manufactured by Altera and sold by Altera or its
// authorized distributors. Please refer to the applicable
// agreement for further details.

// PROGRAM                "Quartus II 64-Bit"
// VERSION                "Version 14.1.0 Build 186 12/03/2014 SJ Full Version"
// CREATED                "Fri Apr 08 15:51:10 2016"

module counter4(
    clk,
    rst,
    out0,
    out1,
    out2,
    out3
);

input wire      clk;
input wire      rst;
output wire     out0;
output wire     out1;
output wire     out2;
output wire     out3;

```

```

wire    SYNTHESIZED_WIRE_22;
wire    SYNTHESIZED_WIRE_1;
wire    SYNTHESIZED_WIRE_2;
wire    SYNTHESIZED_WIRE_3;
reg     SYNTHESIZED_WIRE_23;
reg     SYNTHESIZED_WIRE_24;
wire    SYNTHESIZED_WIRE_4;
wire    SYNTHESIZED_WIRE_5;
wire    SYNTHESIZED_WIRE_6;
reg     SYNTHESIZED_WIRE_25;
wire    SYNTHESIZED_WIRE_26;
wire    SYNTHESIZED_WIRE_27;
wire    SYNTHESIZED_WIRE_10;
wire    SYNTHESIZED_WIRE_11;
wire    SYNTHESIZED_WIRE_12;
wire    SYNTHESIZED_WIRE_13;
wire    SYNTHESIZED_WIRE_17;
reg     SYNTHESIZED_WIRE_28;
wire    SYNTHESIZED_WIRE_20;
wire    SYNTHESIZED_WIRE_21;

assign  out0 = SYNTHESIZED_WIRE_22;
assign  out1 = SYNTHESIZED_WIRE_1;
assign  out2 = SYNTHESIZED_WIRE_2;
assign  out3 = SYNTHESIZED_WIRE_3;


always@(posedge clk or negedge rst)
begin
if (!rst)
begin
SYNTHESIZED_WIRE_23 <= 0;
end
else
begin
SYNTHESIZED_WIRE_23 <= SYNTHESIZED_WIRE_22;
end
end


always@(posedge clk or negedge rst)
begin
if (!rst)
begin
SYNTHESIZED_WIRE_24 <= 0;
end
else
begin
SYNTHESIZED_WIRE_24 <= SYNTHESIZED_WIRE_1;
end
end
end

```

```

always@(posedge clk or negedge rst)
begin
if (!rst)
begin
SYNTHESIZED_WIRE_25 <= 0;
end
else
begin
SYNTHESIZED_WIRE_25 <= SYNTHESIZED_WIRE_2;
end
end

always@(posedge clk or negedge rst)
begin
if (!rst)
begin
SYNTHESIZED_WIRE_28 <= 0;
end
else
begin
SYNTHESIZED_WIRE_28 <= SYNTHESIZED_WIRE_3;
end
end

assign SYNTHESIZED_WIRE_21 = SYNTHESIZED_WIRE_23 & SYNTHESIZED_WIRE_24;

assign SYNTHESIZED_WIRE_2 = SYNTHESIZED_WIRE_4 | SYNTHESIZED_WIRE_5 |
SYNTHESIZED_WIRE_6;

assign SYNTHESIZED_WIRE_6 = SYNTHESIZED_WIRE_23 & SYNTHESIZED_WIRE_25;

assign SYNTHESIZED_WIRE_4 = SYNTHESIZED_WIRE_24 & SYNTHESIZED_WIRE_25;

assign SYNTHESIZED_WIRE_5 = SYNTHESIZED_WIRE_22 & SYNTHESIZED_WIRE_26 &
SYNTHESIZED_WIRE_27;

assign SYNTHESIZED_WIRE_3 = SYNTHESIZED_WIRE_10 | SYNTHESIZED_WIRE_11 |
SYNTHESIZED_WIRE_12 | SYNTHESIZED_WIRE_13;

assign SYNTHESIZED_WIRE_12 = SYNTHESIZED_WIRE_22 & SYNTHESIZED_WIRE_26 &
SYNTHESIZED_WIRE_27 & SYNTHESIZED_WIRE_17;

assign SYNTHESIZED_WIRE_10 = SYNTHESIZED_WIRE_23 & SYNTHESIZED_WIRE_28;

assign SYNTHESIZED_WIRE_13 = SYNTHESIZED_WIRE_24 & SYNTHESIZED_WIRE_28;

assign SYNTHESIZED_WIRE_11 = SYNTHESIZED_WIRE_25 & SYNTHESIZED_WIRE_28;

assign SYNTHESIZED_WIRE_22 = ~SYNTHESIZED_WIRE_23;

assign SYNTHESIZED_WIRE_26 = ~SYNTHESIZED_WIRE_24;

assign SYNTHESIZED_WIRE_27 = ~SYNTHESIZED_WIRE_25;

assign SYNTHESIZED_WIRE_17 = ~SYNTHESIZED_WIRE_28;

```



```

assign SYNTHESIZED_WIRE_20 = SYNTHESIZED_WIRE_22 & SYNTHESIZED_WIRE_26;

assign SYNTHESIZED_WIRE_1 = SYNTHESIZED_WIRE_20 | SYNTHESIZED_WIRE_21;

endmodule

```

Listing 9: Currency Calculator C Program

```

/*
////////////////////////////////////////////////////////////////////

Author: Adolfo Pineda
       Katie neff
       Sharyar Khalid

Title: Currency Calculator

Abstract:      The following program allows the user to enter the rate of
               change
               of currency as compared with the U.S dollar. It then asks the user
               how much the user wanted to convert to either to dollars or
               from dollar to which ever country's currency the user entered.

Introduction:   Ever wonder how much an item from the United States costs in
               another country? Or how about wanting to know how much an item
               from another country costs in the United States? The currency
               calculator can help you deduce these costs using the Currency
               Calculator. All you need to do is input the exchange rate for
               that country and follow the instructions.

Inputs: (double) exchange rate - There are no limits set to this input, but
       the
       user should be aware that negative or zero conversion rates do
       not exist; This the exchange rate used between countries.

       (double) currency amount - There are no limits set to this input, but
       the
       user should be aware that negative dollars do not exist. Also,
       if a non-integer is inputed, then the program does not compute
       anything and asks the user if they would like to calculate
       another value; This is the money amount the user wants converted.

Outputs: (double) total amount - There are no limits set to the size of the
       conversion
       rate or the currency amount, but the output will be rounded to 2
       decimal
       places (in other words, the nearest cent);

Major Functions: The only major function used was the 'main' function where
       most of the
       above mentioned uses of the program were implemented.

*///

```



```

printf("\n");

/* The following while loop executes everytime and breaks if and only
   if the user enters Y for yes or N for no. This ensures if the user
   enters anything other than Y or N a warning would be displayed
   reminding the
   user to only enter Y or N
*/
while(s){

/* The following if statment only executes if and only if the user enters
   yes for the question above asking if the amount entered was in dollars
   or some other currency and if the ammount entered above is in dollars
   the if statment multiplies that ammount with the rate of change and
   converts the dollars to which every currency the user entered rate of
   change of. It also sets the condition for the while loop to false so
   that the while loop breaks.
*/
if(isDollars == 'y' || isDollars == 'Y') {
    total = amount * rate;
    printf("Your total is %.2lf\n", total);
    s = 0;

/* The following else if statement executes if and only if the user enters
   no to the question above and it takes the ammount entered by the user
   and divides it by the rate of change to convert what ever currency was
   entered by the user to U.S dollars. It also sets the condition for the
   while loop to false so that the while loop breaks
*/
    } else if (isDollars == 'n' || isDollars == 'N'){
        total = amount / rate;
        printf("Your total is $%.2lf\n", total);
        s = 0;

/* The following else statment executes if the user enters letters or
   numbers other than Y or N and it gives a warning to the user and
   reminds the user to only enter Y or N depending on what they want
   to do with the ammount entered above
*/
    } else {
        printf("Input is invalid\n");
        printf("Please enter either 'y' for yes or 'n' for no\n");
        scanf("%s", &isDollars);
    }
}

// The following three statements asks the user if they
// would like to convert another value or not.
printf("\n");
printf("Would like to convert another value?\n");
printf("Please enter either 'y' for yes or 'n' for no\n");
scanf("%s", &anotherValue);
printf("\n");
r = 1; // sets the condition for the following while loop to true

/* The following while loop executes everytime and breaks if and only

```

```

    if the user enters Y for yes or N for no. This ensures if the user
    enters anything other than Y or N a warning would be displayed reminding
    the
    user to only enter Y or N
*/
while (r){
    /* The following if statment runs if the user enters Y for the
    question above asking if they would like to convert another
    amount. If they do enter Y the if statement also asks if they
    would like to keep the same rate of change for another value.
    */

    if ( anotherValue == 'Y' || anotherValue == 'y'){
        printf("Would you like to use the same exchange rate?\n");
        printf("Please enter either 'y' for yes or 'n' for no\n");
        scanf("%s", &sameRate);
        printf("\n");
        h = 1; // sets the condition for the following while loop to true.
        r = 0; // sets the condition for the while loop to false.

        /* The following while loop executes everytime and breaks if and
        only if the user enters Y for yes or N for no. This ensures
        if the user enters anything other than Y or N a warning would
        be displayed reminding the user to only enter Y or N
        */
        while (h)
        {

            /* the following if statement executes if and only if the
            user enters Y for the question above that is if the user
            wants to convert another value by keeping the same rate
            of change or change the rate of change. If the user
            enters Y which means the user would like to convert the
            value by keeping the same rate of change as before
            and it sets the condition for the first if statement to false
            */
            if(sameRate == 'Y' || sameRate == 'y'){
                diffConversion = 0;
                h = 0;

                /* the following else if statement executes if and only
                if the user enters N for the question above. It sets
                the condition for the first IF statement to true and
                that if statment executes asking the user for another
                rate of change they would like to use
                */
            }else if (sameRate == 'N' || sameRate == 'n'){
                diffConversion = 1;
                h = 0;

                /* The following else statment executes if the user enters
                letters or numbers other than Y or N and it gives a warning
                to the user and reminds the user to only enter Y or N
                depending on if they would like to use same rate of chane
                or not
                */
            }
        }
    }
}

```

```

    }else {
        printf("Input is invalid\n");
        printf("Please enter either a 'y' for yes or 'n' for no\n");
        scanf("%s", &sameRate);
    }
}

/* The following else if statement executes if and only if the user
   enters
   N to the question of whether or not they would like to convert
   another
   value if they entered N then the condition for the first while
   loop is
   set to 0 and the program ends\
*/
else if (anotherValue == 'n' || anotherValue == 'N'){
    printf("Goodbye have a nice day!\n");
    repeat = 0;
    r = 0;

    /* The following else statment executes if the user enters letters or
       numbers
       other than Y or N and it gives a warning to the user and reminds the
       user
       to only enter Y or N depending on if they would like to convert
       another
       value or not
    */
}else{
    printf("Input is invalid\n");
    printf("Please enter either a 'y' for yes or 'n' no \n");
    scanf("%s", &anotherValue);
    h = 1;
}
}
}

return 0;
}

```