Lab 0 Report

Group 3: Dennis Zang (704766877), Daniel Park (104809832),

Samuel Pando (904809319)

October 9, 2020

Part 1: (Sequential Circuit) BCD Counter - “bcdCounter.v”, “tick.v”

[Components and Implementation Logic]

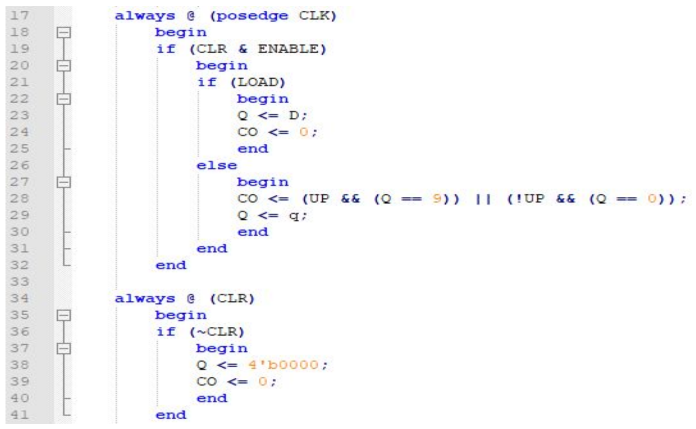
For the design of our sequential circuit, the circuit is a simple top-level module “bcdCounter” that makes use of a sub-level module *tick* to manually increment/decrement the current counter value. The top-level module manually operates the synchronous and asynchronous logic using always statements (flip-flops in an actual netlist/schematic) depending on the CLK, CLR, LOAD, ENABLE, and D signals to operate when to update Q and CO. The sub-level module handles the logic for incrementing/decrementing the current value of Q (the top-level module is what ultimately handles whether the output is loaded into Q in the next clock cycle though). Further details for each module are described below.

[top-level module: “bcdCounter”]

For this sequential circuit, there are two main issues that we needed to consider: updating the counter value synchronously and resetting the counter value asynchronously. To implement the module so that the counter value will trigger on the rising edge of a clock signal, an always block would be necessary (“always @ (posedge CLK)” in “bcdCounter.v”) in order to implement flip-flop behavior. To implement asynchronous reset, we would also need an always block that would trigger on any change in the CLR signal (“always @ (CLR)” in “bcdCounter.v”). We could have also implemented this using an assign statement, but this would interfere with the synchronous behavior required to address the first issue.

For the first always block that triggers for each rising edge of the clock, we would need to add an “if” clause to distinguish whether CLR and ENABLE are both set (as it is only when both are set that further action would proceed). The second conditional for LOAD would be fed into a multiplexor to determine which value would be fed into the flip-flop on the next clock cycle (to load D as the new Q, or to take the incremented/decremented value of the current Q as the next Q).

The second always block includes a single conditional to check if CLR is not set to then proceed to reset Q and CO. In a schematic/netlist, this would be represented as a multiplexer that determines whether a reset value would be fed into the registers for Q and CO (which would also be fed into the flip-flop represented by the first always block for synchronous behavior).



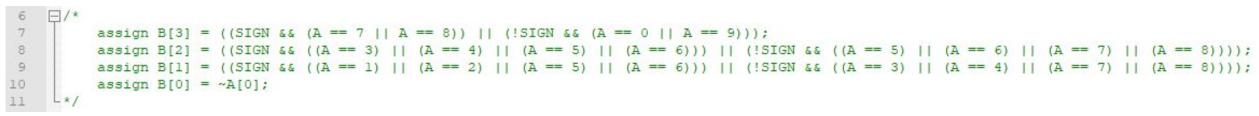
```verilog
always @ (posedge CLK)
    begin
    if (CLR & ENABLE)
        begin
        if (LOAD)
            begin
            Q <= D;
            CO <= 0;
            end
        else
            begin
            CO <= (UP && (Q == 9)) || (!UP && (Q == 0));
            Q <= q;
            end
        end
    end

always @ (CLR)
    begin
    if (~CLR)
        begin
        Q <= 4'b0000;
        CO <= 0;
        end
    end
```

Figure 1.1: The separate always blocks that process the synchronous and asynchronous logic in *bcdCounter.v*.

[sub-level module: *tick*]

Finally, there is also the issue that we need to increment/decrement the counter value manually using logic gates. We could implement an adder module, but it isn't required as we will be adding/subtracting 1 each time, so we implemented a "truth table" for each BCD bit. The logic here would be to consider the mod of the current input to a power of 2 (2^1 to 2^3) to see whether a bit would be set after an increment/decrement. For the 1st least-significant bit, after an increment/decrement, the bit will always be the bitwise-negation of the current value (x mod 2 will always be inverted after increment/decrement). For the 2nd bit: only the values 1, 2, 5, and 6 after an increment would have the 2nd bit set set; and only the values 3, 4, 7, and 8 after a decrement would have the 2nd bit set (x mod 4 = 2 or 3 after increment/decrement). For the 3rd bit: only the values 3, 4, 5, and 6 after an increment would have the 3rd bit set set; and only the values 5, 6, 7, and 8 after a decrement would have the 3rd bit set (x mod 8 = 4, 5, 6, or 7 after increment or decrement). The 4th bit depends if the final sum becomes an 8 or 9 after the increment/decrement, so we need to detect a 7 or 8 for increment, or a 0 or 9 for decrement. One important thing to note is that the behavior will break if a non-BCD value is inputted (any non-BCD 4-bit value from 1010-1111). So, the module depends on the D input being a proper BCD, otherwise the behavior will be undefined.



```
/*
    assign B[3] = ((SIGN && (A == 7 || A == 8)) || (!SIGN && (A == 0 || A == 9)));
    assign B[2] = ((SIGN && ((A == 3) || (A == 4) || (A == 5) || (A == 6))) || (!SIGN && ((A == 5) || (A == 6) || (A == 7) || (A == 8))));
    assign B[1] = ((SIGN && ((A == 1) || (A == 2) || (A == 5) || (A == 6))) || (!SIGN && ((A == 3) || (A == 4) || (A == 7) || (A == 8))));
    assign B[0] = ~A[0];
*/
```

Figure 1.2: The bit assignment in "tick.c" is equivalent to the above.

[Waveforms/Testbench]

For the testbench *bcdCounterTB.v*, we ran test cases for the possible following scenarios:

1) Loading and storing a BCD D for a given period of clock cycles (prove LOAD works) (refer to [0ns, 40ns) in waveform)
2) Incrementing the counter over all integer digits (prove Q and CO acts as they should) (refer to [40ns, 140ns) in waveform)
3) Disabling ENABLE to prove functionality of the ENABLE input with incrementing (refer to [140ns, 170ns) in waveform)
4) Decrementing the counter over all integer digits (prove Q and CO acts as they should) (refer to [170ns, 270ns) in waveform)
5) Disabling ENABLE to prove functionality of the ENABLE input with decrementing (refer to [270ns, 300ns) in waveform)
6) Reset Q asynchronously by unsetting CLR (when CO = 0 and CO = 0) (refer to [311ns, 350ns) in waveform)
7) Reset Q and CO asynchronously by unsetting CLR (when Q = 0 or 9 and CO = 1) (refer to [381ns, 420ns) and [451ns, 480ns) in waveform)

   (Please refer to the code comments in *bcdCounter.v* for test cases and expected values of Q and CO with their respective timestamps.)

   The waveform is as shown below.

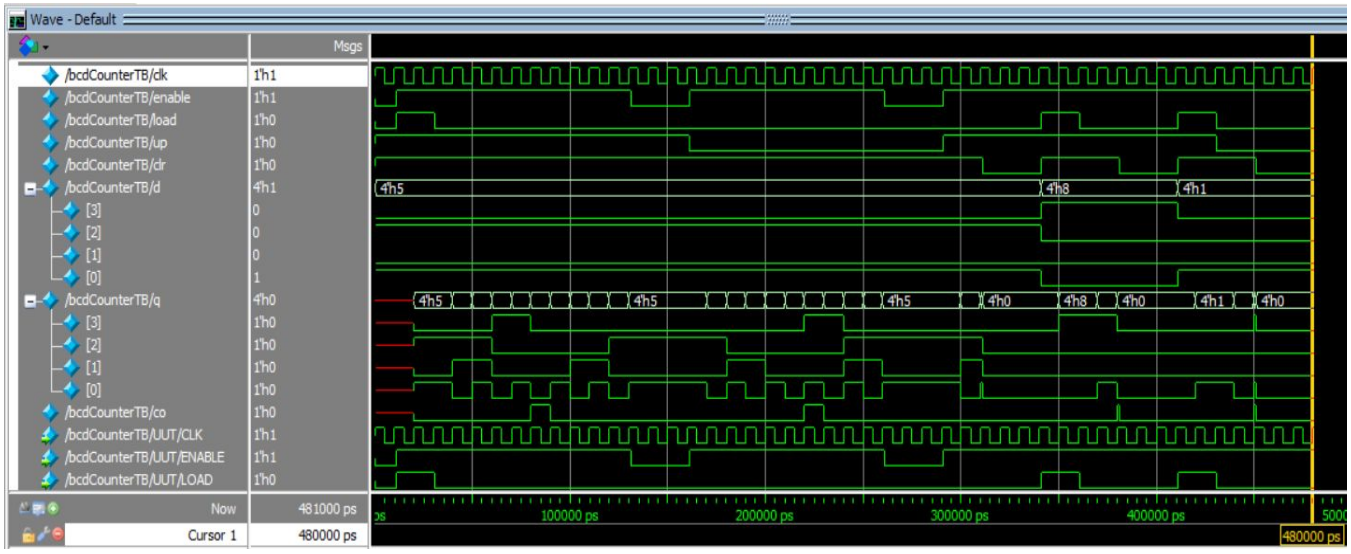


Figure 1.3: The waveforms of the inputs and outputs to the module "bcdCounter". The inputs and outputs being monitored are the lowercase inputs and outputs of the testbench *bcdCounterTB*.

[Challenges Faced]

The main challenges presented in Lab 0 are reviewing what we had learned in COM SCI M152A (a while back) and getting used to Mentor Graphics Modelsim. Specifically, the main troubles were relearning Verilog and looking up how to install/use Modelsim instead of Xilinx ISE.

For the implementation of Part 1 of Lab 0, the two main difficulties were getting the updates to registers Q and CO to work synchronously with the clock signal CLK while resetting Q and CO asynchronously on CLR regardless of signal CLK. In addition, as we are only allowed to use basic logic gates, multiplexers, and flip-flops, we needed to create a separate module for the sole purpose of incrementing/decrementing based on what BCD is currently being processed (done manually using a truth table approach).

The biggest difficulty for Part 1 was not the implementation, but the testbench. Writing a comprehensive testbench to test all possible inputs and outputs while checking the waveform for correctness for each time cycle (and when CLR = 0) wasn't too difficult, but tedious.

Part 2.1: (Combinational Circuit) 4-Digit BCD Divisible by 3 - "Multiple3.v"

[Components and Implementation Logic]

The result of this combinational circuit is whether the input is divisible by 3. The singular input's expected form is 16-bits of binary coded decimal (BCD). Deviating from these input specifications results in an unreliable result which should not be interpreted as indicative of either result. The output is a single bit which follows standard boolean rules. 0 is considered a multiple of 3. For a diagram of this workflow, see Figure 2.1.1.

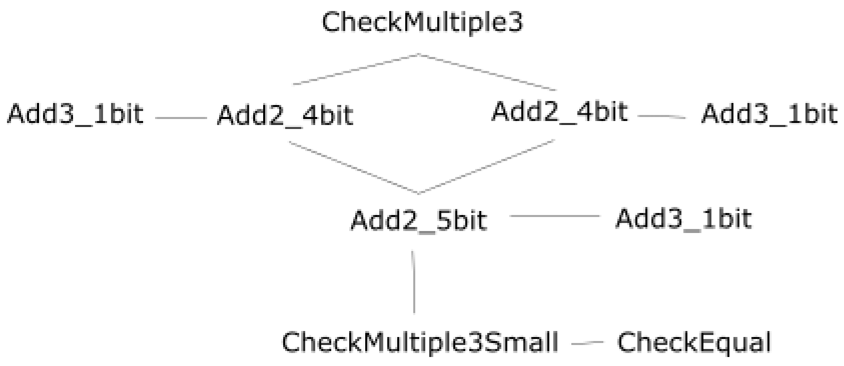
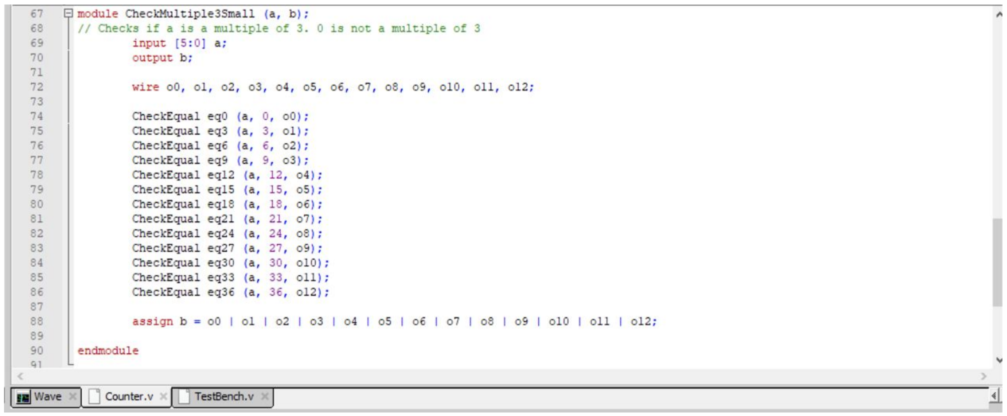


Figure 2.1.1 Module Workflow. Horizontal lines denote the inner module calls/utilizes the outer module. Vertical lines denote a module passes its output (in part or in full) as input to the next module.

The module utilizes the divisibility rule for 3, namely the digits of a number divisible by 3 sum to a multiple of 3. Since the input is provided in BCD, the module adds the first two digits together, the last two digits together, and then the results of both additions together to get a single sum. That sum is then checked to be a multiple of 3. That result is then wired to the output.

"Multiple3.v" holds Verilog code of all modules used for this circuit and is included in the project folder. The highest level module is named *CheckMultiple3*. It calls modules *Add2_4bit*, *Add2_5bit*, and *CheckMultiple3Small*. *Add2_4bit* and *Add2_5bit* are adders, but differ in the number of bits they accept as inputs (4 and 5 bits respectively). There are multiple adders since the number of bits increases with each addition. (The sum of 2 4-bit numbers requires 5 bits and the sum of 2 5-bit numbers requires 6 bits). Both call *Add3_1bit* for the adding component. *Add3_1* in turn utilizes XNOR, OR, and AND gates to map 3 inputs to a 2 bit sum output. *Add2_4bit* modules first add the initial BCD digits together. *Add2_5bit* then adds the resulting sums together and passes its 6-bit output to *CheckMultiple3Small.* This module checks if its input is a multiple of 3, but differs from the outer *CheckMultiple3* module in its scale and expected input. It only accepts a 6-bit number in binary form and checks for divisibility by 3 using the *CheckEqual* module to compare its input to 0, 3, 6, 9, 12, ... stopping at 36 which is the maximum sum for 4 single digit numbers. The results are all given as inputs to a OR gate and the result is then passed as *CheckMultiple3*'s output. For an example of *CheckEqual*'s code, see Figure 2.1.2.



```verilog
module CheckMultiple3Small (a, b);
// Checks if a is a multiple of 3. 0 is not a multiple of 3
        input [5:0] a;
        output b;

        wire o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12;

        CheckEqual eq0 (a, 0, o0);
        CheckEqual eq3 (a, 3, o1);
        CheckEqual eq6 (a, 6, o2);
        CheckEqual eq9 (a, 9, o3);
        CheckEqual eq12 (a, 12, o4);
        CheckEqual eq15 (a, 15, o5);
        CheckEqual eq18 (a, 18, o6);
        CheckEqual eq21 (a, 21, o7);
        CheckEqual eq24 (a, 24, o8);
        CheckEqual eq27 (a, 27, o9);
        CheckEqual eq30 (a, 30, o10);
        CheckEqual eq33 (a, 33, o11);
        CheckEqual eq36 (a, 36, o12);

        assign b = o0 | o1 | o2 | o3 | o4 | o5 | o6 | o7 | o8 | o9 | o10 | o11 | o12;

endmodule
```

Figure 2.1.2 *CheckMultiple3Small* Code. The Verilog code compares the input to different multiples of 3 using the *CheckEqual* module. The wires connect the *CheckEqual* modules' outputs to the equation assigning the *CheckMultiple3Small* output.

[Waveforms/Testbench]

"Multiple3TestBench.v" holds individual testbenches for each of the modules listed in "Multiple3.v." All of them assign different values as inputs and check for the proper output. For Adder modules, the inputs are changed every few nanoseconds to test a wide variety of values with the exception of *Add3_1bit* which is small enough that every possible input can be tested. The inputs are purposefully diversified in bit-length to examine each part of the adders, and outputs are compared to their theorized answer as a check for correct behaviour.

For modules with boolean outputs, the testing process is similarly rigorous with a mix of correct and incorrect inputs. For example, the testbench for the module *CheckEqual* (*CheckEqualTB*) occasionally assigns inputs which should yield false since the module's correct behaviour should differentiate between equal and unequal values. The outputs are also compared to their theoretical correct results and the result of these comparisons are output to a register called "pass." A value of 1 in this register indicates the module's output matches the expected output. The waveforms during the testbench for *CheckMultiple3* (*Multiple3TB)* can be found in Figure 2.1.3. For waveforms of the testbenches for the other modules, check the project folder.

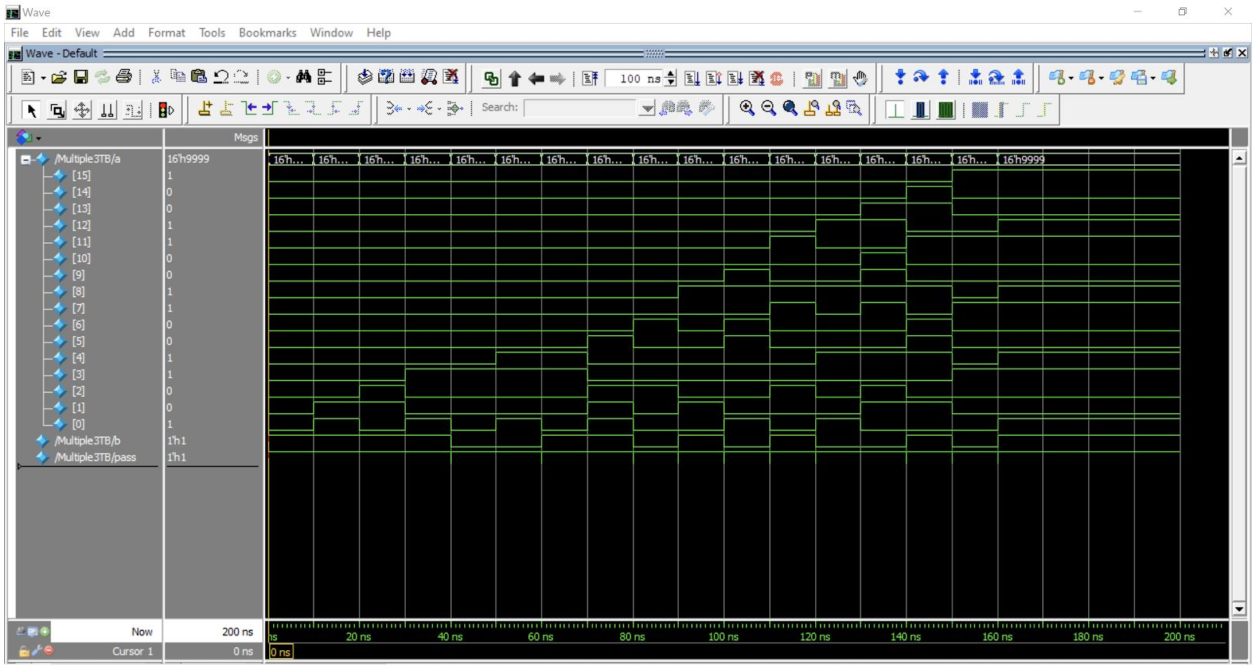


Figure 2.1.3 *Multiple3TB* Waveform. The resulting waveform when testing *Multiple3* for correct behavior. Note the pass register remains 1, signalling the module's output matches the expected output.

[Challenges Faced]

The main difficulties when designing this module stemmed from unfamiliarity with the Modelsim software and the Verilog code. Problems with the software included navigating through its window layout, retrieving the proper student license, and understanding error logs. Problems with Verilog included naming conventions (Verilog does not allow function names to start with numbers), syntax errors, and differentiating between registers and wires.

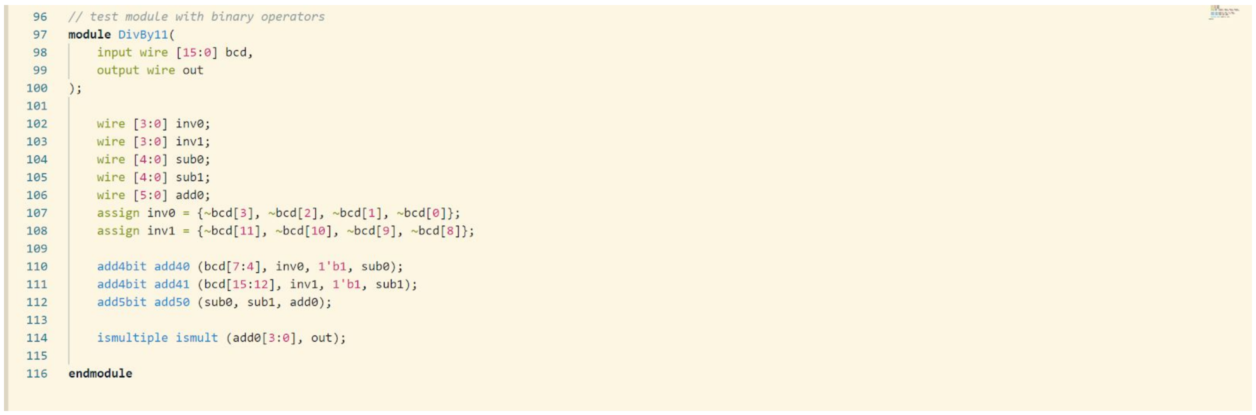
Part 2.2: (Combinational Circuit) 4-Digit BCD Divisible by 11 - "DivBy11.v"

[Components and Implementation Logic]

This combinational circuit is meant to determine whether the number represented by the 16-bit BCD input is a multiple of 11. If true, the circuit outputs a 1, and it outputs a 0 otherwise. The BCD is a multiple of 11 if the sum of the difference of alternating digits ((4th - 3rd) + (2nd - 1st)) is a multiple of 11 (for example, 0 or 11).

Given this knowledge on how to find out if the BCD was a multiple of 11, I took an approach very similar to the one my teammate took on the previous section. That is to say that I tried to implement a full adder (since binary operators were banned). In my case, instead of addition (though I had to do some of that too) I needed to do subtraction, but as it turns out, a full adder can do subtraction too. For example, 5 - 3 can also be taken as 5 + (-3). So to subtract with a full adder, I needed to add the inverse of the negative term, which in this case happened to be the 2's complement of the binary number. I made the module *add4bit* to accomplish this. It took two BCD digits and one carry in value, returning their sum as a 5-bit binary number. When I needed to subtract, I inverted the second input and set the carry in bit to 1.

After this, I would then add the results of the two subtractions using the module *add5bit*, which added 5-bit binary numbers and output 6-bit ones. I was worried that with all the 2's complement subtraction and ever expanding bit count, the least significant bits of my output numbers would begin to lose meaning. They did, but as it turned out, only the four least significant bits mattered, and of those four least significant bits, there were only two combinations that would reveal a 16-bit BCD to be a multiple of 7: 1011 (11 or -11) and 0000. Finally, I would use my *isequal* module to check if one of these values was in the four least significant bits of my sum and return the result.



```verilog
// test module with binary operators
module DivBy11(
    input wire [15:0] bcd,
    output wire out
);

    wire [3:0] inv0;
    wire [3:0] inv1;
    wire [4:0] sub0;
    wire [4:0] sub1;
    wire [5:0] add0;
    assign inv0 = {~bcd[3], ~bcd[2], ~bcd[1], ~bcd[0]};
    assign inv1 = {~bcd[11], ~bcd[10], ~bcd[9], ~bcd[8]};

    add4bit add40 (bcd[7:4], inv0, 1'b1, sub0);
    add4bit add41 (bcd[15:12], inv1, 1'b1, sub1);
    add5bit add50 (sub0, sub1, add0);

    ismultiple ismult (add0[3:0], out);

endmodule
```

Figure 2.2.1 "DivBy11.v" *divby11* module. The alternating digits of the BCD must be inverted before being input to the full adder, and the carry bit must also be set to 1 to fulfill 2's complement.

[Waveforms/Testbench]

I tested each of my modules, but the magnitude of the tests varied. For modules like *add4bit* (my full adder), *add5bit* (an extension of the full adder), *ismultiple* (the final check for the sum being a multiple of seven), and *divby11* (the encapsulating module that runs the full circuit) there were many test covering all of the edge cases possible. However, there were a few less significant modules like *add2bit* (a supplementary half adder) I will spare most of the details since there is significant overlap between this section and the previous one (including the testing). But I will expand on the subtraction aspect, since this is unique to my section.

In “DivBy11_tb.v” I did a lot of testing on the full adder (both 4 and 5 bit) to make sure that subtraction would actually give me the values that I wanted. I specifically tested subtraction resulting in a positive number, subtraction resulting in a negative number, the highest and lowest values possible through subtraction, and subtracting from and by 0. I also did many tests on the 5 bit adder to see what the initially 4-bit digits would look like after going through 2’s complement subtraction and gaining two extra bits. As previously stated, in the transition from 3 to 4 to 5 bit, the most significant bits lost some of their meaning, but this ended up being to my benefit. Through testing I discovered that the least significant bits (which happened to contain the values I needed) stayed consistent in their ability to denote a multiple of 11.

Finally, I tested the resulting circuit for all significant multiples of 7 of four digits. In the end, there were only 3 real values that could be multiples of 11: those whose differences added up to 11, those whose differences subtracted to negative 11, and those whose differences canceled to 0. After that, all I had to do was test a value that definitely did not fit any of the above criteria and make sure it failed. After doing that (and testing some of the trickier subtraction values) I was satisfied.

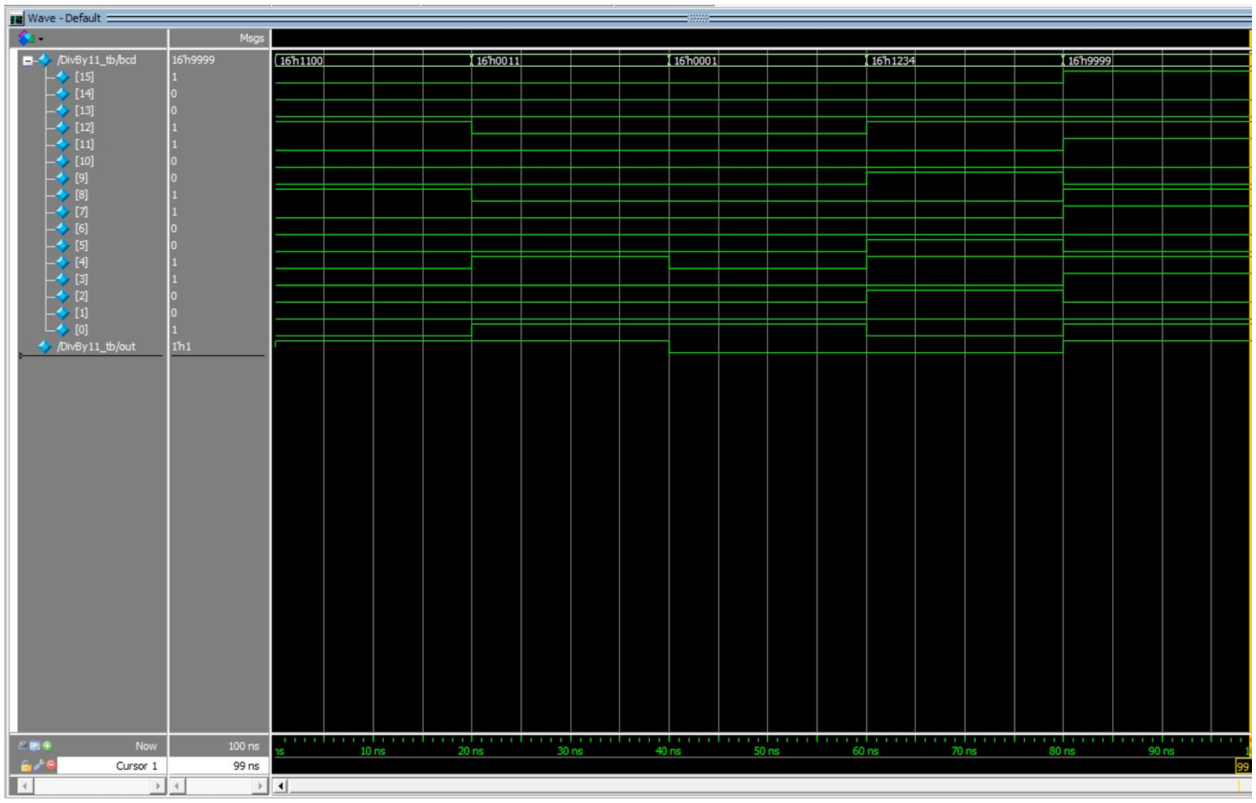


Figure 2.2.2 *DivBy11* Waveform. The module takes the BCD as an input and returns a 1 as true or 0 as false. This testbench tests the most significant edge cases of the circuit and the conditions of adding to 11, subtracting to -11, or canceling to 0.

[Challenges Faced]

First off, as a mac user, it was very difficult to actually run Modelsim at all. At first, I tried to run it inside a virtual machine, but as it was proving too difficult for the effort, I abandoned this route. In the end, I found enough space to dual boot my mac with windows and run modelsim in there.

The modelsim interface was also a bit of a challenge. I had worked with Modelsim and verilog code briefly in the past, but it was long ago enough that the immediate layout was unrecognizable. However, after various tutorials and guides, I was able to figure out how to compile and simulate (which is the bare minimum necessary for this particular lab). I look forward to learning the system in greater detail in the future.

As for the lab itself, my most significant challenge was implementing subtraction. I was particularly worried about how I would maintain a concrete representation of what negative even was since many of my modules used different bit counts for calculations. It wasn't until I discovered that I only really needed to check for 3 multiple values (11, 0, and -11) that I began to make some headway. Finally, upon realizing that I only needed to check the four least significant bits (since 11 and -11 are the same in that regard), I was able to get the simulation running as intended.

Appendix

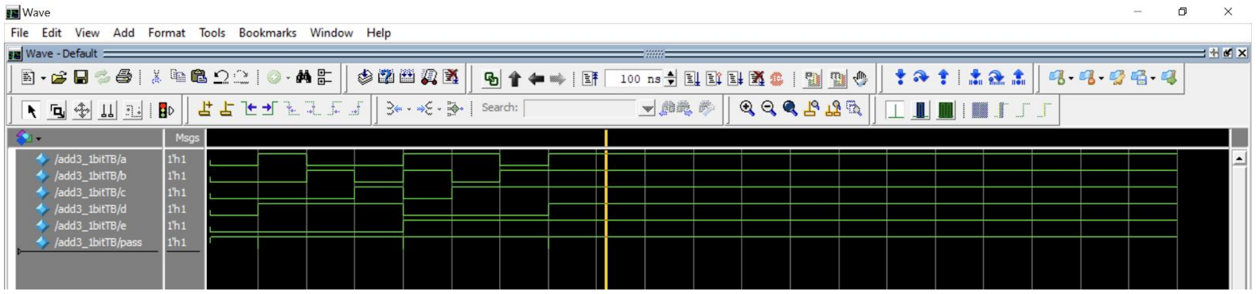


Figure 2.1.4 ***Add3_1bitTB*** **Waveform. The module takes 3 inputs a, b, c and returns the output sum (d) and a carry value (e). Since the inputs were few in number and 1-bit each, the testbench for this waveform tests all possible inputs.**

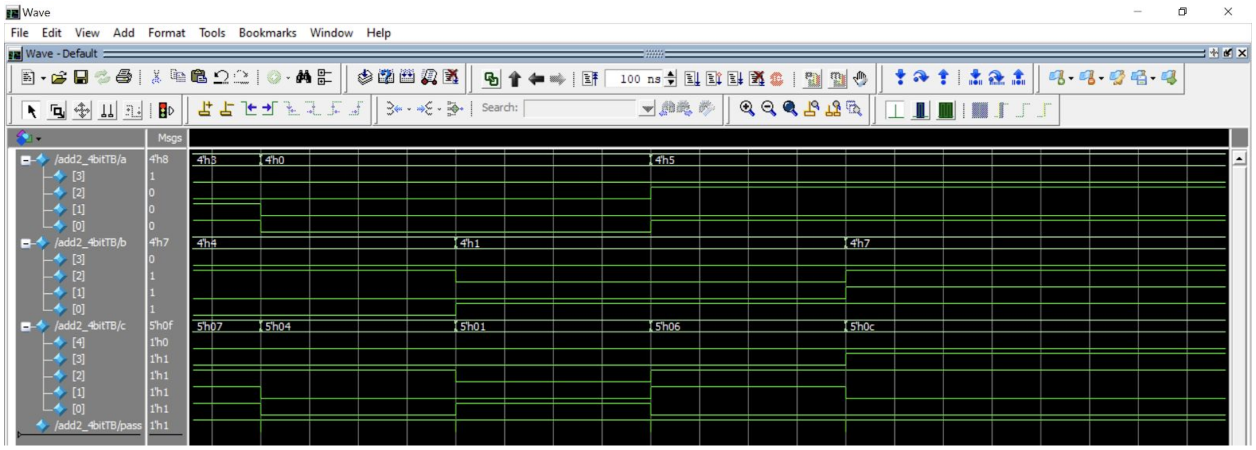


Figure 2.1.5 ***Add2_4bitTB*** **Waveform. This module takes 2 4-bit numbers and returns their 5-bit sum. An additional bit is required in case the last bits produce a carry. The testbench for this waveform can be found in the project folder.**

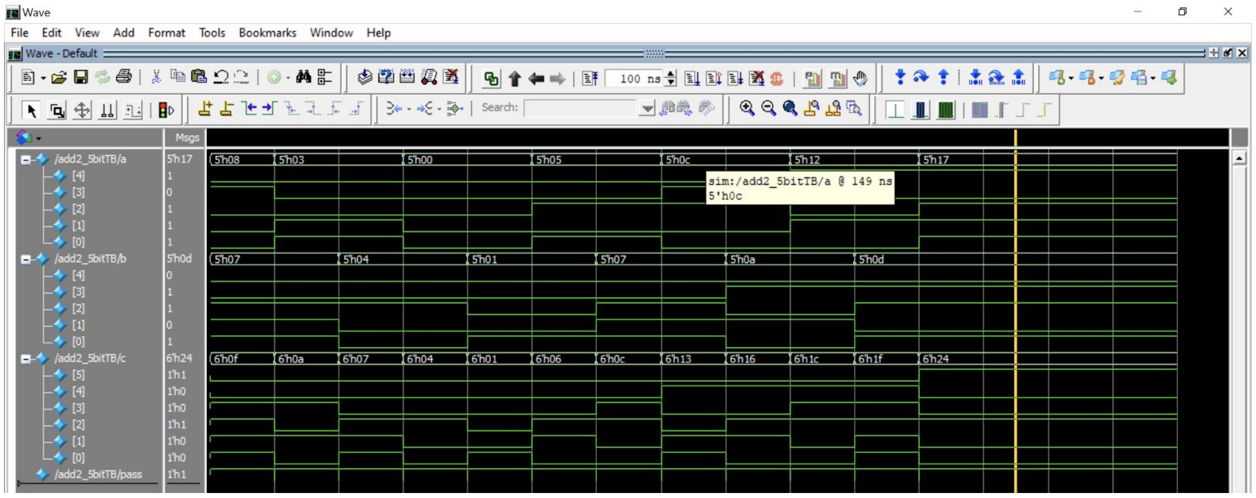

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
/add2_5bitTB/a 5'h17
[4] 1
[3] 0
[2] 1
[1] 1
[0] 1
/add2_5bitTB/b 5'h0d
[4] 0
[3] 1
[2] 1
[1] 0
[0] 1
/add2_5bitTB/c 6'h24
[5] 1'h1
[4] 1'h0
[3] 1'h0
[2] 1'h1
[1] 1'h0
[0] 1'h0
/add2_5bitTB/pass 1'h1
5'h08 5'h03 5'h00 5'h05 5'h0c 5'h12 5'h17
5'h07 5'h04 5'h01 5'h07 5'h0a 5'h0d
6'h0f 6'h0a 6'h07 6'h04 6'h01 6'h06 6'h0c 6'h13 6'h16 6'h1c 6'h1f 6'h24
sim:/add2_5bitTB/a @ 149 ns
5'h0c

Figure 2.1.6 ***Add2_5bitTB*** **Waveform. This module takes 2 5-bit numbers and returns their 6-bit sum. An additional bit is required in case the last bits produce a carry even if the largest expected number while following the expected input rules is 36. The testbench for this waveform can be found in the project folder.**

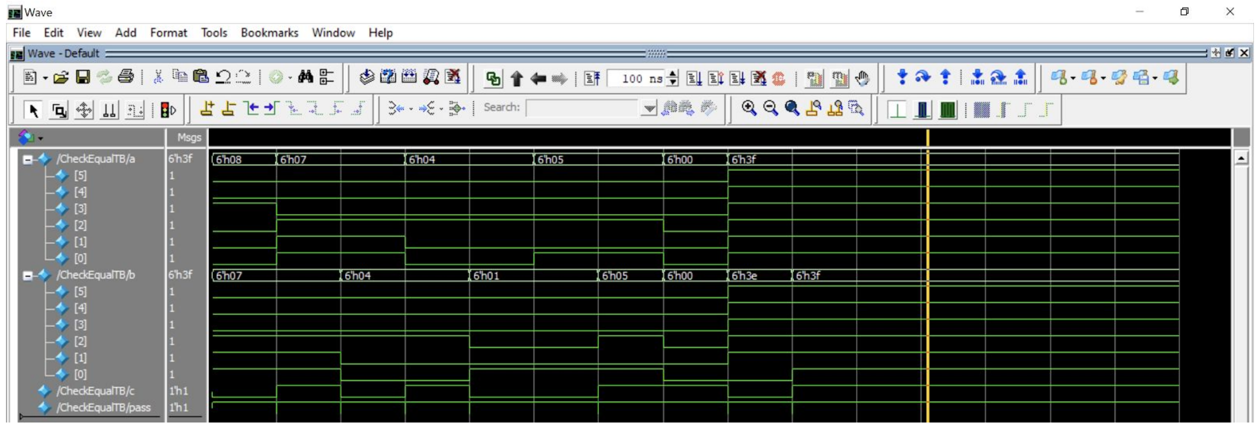


Figure 2.1.7 *CheckEqual* Waveform. This module takes 2 6-bit numbers as inputs and compares their values. If the first input (a) has the same value as the second input (b), a 1 is output. Otherwise a 0 is output. The testbench for this waveform can be found in the project folder.

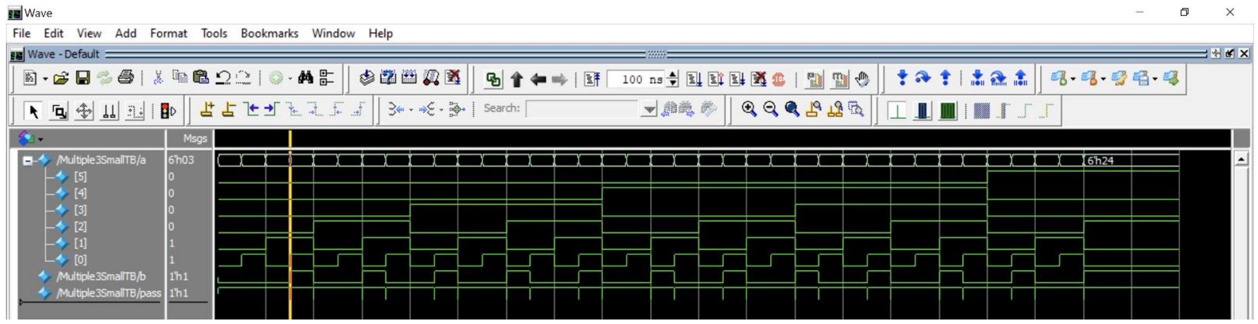


Figure 2.1.8 *Multiple3SmallTB* Waveform. This module takes a single 6-bit number as its input and outputs whether that number is divisible by 3. Its input must be in binary rather than BCD. The testbench for this waveform can be found in the project folder.

Member Contributions:

Dennis Zang:

- Completed Lab 0 Part 1 and wrote the report analysis for his part.

Daniel Park:

- Completed Lab 0 Part 2.1 and wrote the report analysis for his part.

Samuel Pando:

- Completed Lab 0 Part 2.2 and wrote the report analysis for his part.