

# **Sequential Logic Design with Counters in Verilog Design Report**

Benjamin Betancourt

Lab Dates: February 10<sup>th</sup> – February 18<sup>th</sup>, 2025

Date of Submission: February 17<sup>th</sup>, 2025

# Table of Contents

<b>Introduction .....</b>	<b>3</b>
<b>Implementation.....</b>	<b>3-5</b>
<b>Results.....</b>	<b>5-7</b>
<b>Conclusion.....</b>	<b>7</b>

**Introduction:**

This project is a good introduction to the storage of values and state management. The requirement of a load case and store case for the GCD calculation introduced the requirement for previous state storage. This project also required the implementation of a GCD algorithm in c as well, so it offered a good comparison between traditional programming languages and hardware description languages. The developed algorithm was used in the implementation of the Verilog step as it allowed for expectations of what the code looks like to be drawn. The Verilog step made use of sequential blocks such as the always block and case statements to compute the result. Lastly, registers were used heavily to store previous states in the case that the load button is not pressed.

**GCD Algorithm:**

The steps of the provided algorithm are as follows:

- 1) Start with two positive numbers, A and B
- 2) If  $\min(A, B) == 0$ , jump to step 5
- 3) Subtract the smaller number from the larger number
- 4) Jump to step 2
- 5) The maximum of A and B is the GCD

The running of the algorithm can be seen below with each step and intermediary calculation included:

Algorithm given  $A=55, B=121$

```
1:  $A = 55, B = 121$ 
2:  $\min(55, 121) = 55$ ; miss branch
3:  $121 - 55 = 66 = B$ 
4: take branch to step 2
5:  $\min(55, 66) = 55$ ; miss branch
6:  $66 - 55 = 11 = B$ 
7: take branch to step 2
8:  $\min(55, 11) = 11$ ; miss branch
9:  $55 - 11 = 44 = A$ 
10: take branch to step 2
11:  $\min(44, 11) = 11$ ; miss branch
12:  $44 - 11 = 33 = A$ 
13: take branch to step 2
14:  $\min(33, 11) = 11$ ; miss branch
15:  $33 - 11 = 22 = A$ 
```

16: *take branch to step 2*  
17:  $\min(22, 11) = 11$ ; *miss branch*  
18:  $22 - 11 = 11 = A$   
19: *take branch to step 2*  
20:  $\min(11, 11) = 11$ ; *miss branch*  
21:  $11 - 11 = 0 = A$   
22: *take branch to step 2*  
23:  $\min(0, 11) = 0$ ; *take branch to step 5*  
24:  $\max(0, 11) = 11$  *which is the GCD*

### GCD Implementation in C:

My implementation in C using the algorithm and its output shown above is as follows:

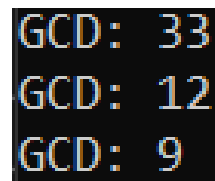
```
#include <stdio.h>

int max(int a, int b) {
    if (a >= b) {
        return a;
    }
    else {
        return b;
    }
}

int min(int a, int b) {
    if (a <= b) {
        return a;
    }
    else {
        return b;
    }
}

int GCD(int a, int b) {
    if (min(a, b) == 0) {
        return max(a, b);
    }
    while (min(a, b) != 0) {
        if (a - b >= 0) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return max(a, b);
}

void main(void) {
    printf("GCD: %d\n", GCD(165, 363));
    printf("GCD: %d\n", GCD(48, 180));
    printf("GCD: %d\n", GCD(27, 36));
}
```



GCD C program output

### **GCD Implementation in Quartus using Verilog:**

The implementation in C helped guide the design of the hardware implementation of a GCD calculator using Verilog. Due to Verilog describing hardware rather than simply being a program, some major changes needed to be made to the approach as things such as while loops and function calls inside of loops work differently in a hardware environment. While loops are unraveled by a HDL compiler as they simply represent repeated code. Any time I made use of a while loop in my design, I got an error due to the 150 iteration limit for this specific FPGA due to physical chip size limitations. So I made use of an always block with a parameter of the positive edge of the internal clock. I also made use of concatenation to combine both my reset and load buttons into one variable to be used as an input to a case statement. This case statement handles, reset, load, and computation logic. Given that the buttons operate at a default state of 1 on the MAX10 FPGA (0 is pressed, 1 is released), the default state for most computations is '11' in the state variable. I had to manually unravel the min and max modules as modules in always blocks are prohibited in Verilog due to space concerns. When 11 is seen in the case statement, an unraveled min module is called first. The result of this min module is used as a comparison parameter of a nested if else statement. The outer if statement checks for result of 0 from the min function, if this result is found, then we have our output. The output is assigned to the result of an unraveled max function, and the done output is set to 1. If the initial if statement is found to be false, the else part is ran. What the whole inner layer of the nested if statement does is it subtracts a from b if  $b > a$  and subtracts b from a if  $a > b$ . This part is repeated in following clock cycles until the initial if statement is met. This whole process completes the computation. The second case is if the reset button is pressed. A, B, done, and result are all set to zero therefore resetting the process back to zero. The third case is the set case. This case reads the inputs and sets A to its input and B to its input. The final case is the instance both buttons are pressed, here I prioritized reset and reimplemented that functionality. A default case is also implemented for circuit resilience's sake, and it simply emulates the reset case. I intentionally generated latches by assigning registers to themselves to prevent automatic latches from causing timing issues I also implemented on my own accord a five-bit seven segment display decoder that displays the result across two seven-segment displays in decimal. This module is called at the end. The code for both my implementation and seven segment decoders can be found on the following pages.

```

module gcd(input clk,
input reset,
input load,
input [4:0] aIn,
input [4:0] bIn,
output reg [4:0] result,
output [7:0] disp0,
output [7:0] disp1,
output reg done);

reg [4:0] a, b;
reg [4:0] mOut;
wire [1:0] state = {reset, load}; //remember to default these to 1

initial begin
    result <= 5'b00000;
    a <= 5'b00000;
    b <= 5'b00000;
end

always@ (posedge clk) begin

    case(state) //State dictates what the circuit does "11" the case when neither reset or set button is pressed
        2'b11 : begin
            mOut <= (a<b)? a : b; //min module translated to a statement synthesisable in an always statement
            if (mOut == 5'b00000) begin //checks for a minimum result of 0
                result <= (a>b)? a : b; //max module translated to a statement synthesisable in an always statement
                done <= 1'b1; //operation done
                a <= a; //intentional latch generation
                b <= b; //intentional latch generation
            end else begin
                if (a>=b) begin
                    a <= a-b;
                    b <= b; //Intentional latch generation
                    done <= 1'b0;
                    result <= result; //Intentional latch generation
                end else begin
                    a <= a; //intentional latch generation
                    b <= b-a;
                    done <= 1'b0;
                    result <= result; //Intentional latch generation
                end
            end
        end
        2'b01 : begin
            a <= 5'b00000;
            b <= 5'b00000;
            done <= 1'b0;
            result <= 5'b00000;
        end
        2'b10 : begin
            a <= aIn;
            b <= bIn;
            done <= 1'b0;
            result <= result; //Intentional latch generation
        end
        2'b00 : begin
            a <= 5'b00000;
            b <= 5'b00000;
            done <= 1'b0;
            result <= 5'b00000;
        end
        default : begin
            a <= 5'b00000;
            b <= 5'b00000;
            done <= 1'b0;
            result <= result; //Intentional latch generation
        end
    endcase

end

fiveBitSevenSegmentDecoder displayResults(result, disp1, disp0);

endmodule

```

```

module fiveBitSevenSegmentDecoder(input [4:0] in,
output [7:0] disp1, output [7:0] disp0);

reg [7:0] wout1;
reg [7:0] wout0;

always@ (*) begin
    case(in) //NOTE: wout1/disp1 = hex1 and wout0/disp0 = hex0 on MAX10 FPGA
        0: begin wout1 = 8'b11000000; wout0 = 8'b11000000; end
        1: begin wout1 = 8'b11000000; wout0 = 8'b11111001; end
        2: begin wout1 = 8'b11000000; wout0 = 8'b10100100; end
        3: begin wout1 = 8'b11000000; wout0 = 8'b10110000; end
        4: begin wout1 = 8'b11000000; wout0 = 8'b10011001; end
        5: begin wout1 = 8'b11000000; wout0 = 8'b10010010; end
        6: begin wout1 = 8'b11000000; wout0 = 8'b10000010; end
        7: begin wout1 = 8'b11000000; wout0 = 8'b11111000; end
        8: begin wout1 = 8'b11000000; wout0 = 8'b10000000; end
        9: begin wout1 = 8'b11000000; wout0 = 8'b10010000; end
        10: begin wout1 = 8'b11111001; wout0 = 8'b11000000; end
        11: begin wout1 = 8'b11111001; wout0 = 8'b11111001; end
        12: begin wout1 = 8'b11111001; wout0 = 8'b10100100; end
        13: begin wout1 = 8'b11111001; wout0 = 8'b10110000; end
        14: begin wout1 = 8'b11111001; wout0 = 8'b10011001; end
        15: begin wout1 = 8'b11111001; wout0 = 8'b10010010; end
        16: begin wout1 = 8'b11111001; wout0 = 8'b10000010; end
        17: begin wout1 = 8'b11111001; wout0 = 8'b11111000; end
        18: begin wout1 = 8'b11111001; wout0 = 8'b10000000; end
        19: begin wout1 = 8'b11111001; wout0 = 8'b10010000; end
        20: begin wout1 = 8'b10100100; wout0 = 8'b11000000; end
        21: begin wout1 = 8'b10100100; wout0 = 8'b11111001; end
        22: begin wout1 = 8'b10100100; wout0 = 8'b10100100; end
        23: begin wout1 = 8'b10100100; wout0 = 8'b10110000; end
        24: begin wout1 = 8'b10100100; wout0 = 8'b10011001; end
        25: begin wout1 = 8'b10100100; wout0 = 8'b10010010; end
        26: begin wout1 = 8'b10100100; wout0 = 8'b10000010; end
        27: begin wout1 = 8'b10100100; wout0 = 8'b11111000; end
        28: begin wout1 = 8'b10100100; wout0 = 8'b10000000; end
        29: begin wout1 = 8'b10100100; wout0 = 8'b10010000; end
        30: begin wout1 = 8'b10110000; wout0 = 8'b11000000; end
        31: begin wout1 = 8'b10110000; wout0 = 8'b11111001; end
        default: begin wout1 = 8'b01111111; wout0 = 8'b01111111; end //invalid returns
a decimal place only
    endcase
end

assign disp1 = wout1;
assign disp0 = wout0;

endmodule

/* Reference table dec
0 = 8'b11000000
1 = 8'b11111001
2 = 8'b10100100
3 = 8'b10110000
4 = 8'b10011001
5 = 8'b10010010
6 = 8'b10000010
7 = 8'b11111000
8 = 8'b10000000
9 = 8'b10010000
end reference table */

```

## FPGA and Simulation Implementation and results:

The code ran as expected on both the FPGA and the Questasim, however, the waveform simulation uncovered some bugs not found on the hardware implementation that were corrected. The FPGA implementation worked fully; however, the waveforms would occasionally show unknown values from bad latch implementations. I corrected these issues by manually invoking the auto latch generation that Quartus is capable of creating. This allowed me to prevent undefined behavior by manually assigning values where applicable and assigning itself when I simply wanted a variable to be stored. The end result is the code seen in the previous section and the test bench/waveforms seen below.

```

module gcd_TB();
//inputs
reg clk, reset, load;
reg [4:0] aIn, bIn;
//outputs
wire [4:0] result;
wire done;
wire [7:0] disp0, disp1;

gcd DUT(clk, reset, load, aIn, bIn, result, disp0, disp1, done);

always begin
    #2 clk = ~clk;
end

initial begin
    clk = 0; //set initial clock

    //start test cases
    reset = 1; load = 0; aIn = 30; bIn = 10; #4; // a = 30, b = 10
    reset = 1; load = 1; aIn = 30; bIn = 10; #100; //release store button and give some
compute cycles, result should be 10

    reset = 0; load = 1; aIn = 30; bIn = 10; #4; //press reset
    reset = 1; load = 1; aIn = 30; bIn = 10; #100; //wait, result should be back to 0

    reset = 1; load = 0; aIn = 15; bIn = 25; #4; //a = 15, b = 25  b>a case
    reset = 1; load = 1; aIn = 15; bIn = 25; #100; //release load and wait for compute,
result should be 5

    reset = 1; load = 0; aIn = 15; bIn = 6; #4; //a = 15, b = 6  a>b case
    reset = 1; load = 1; aIn = 15; bIn = 6; #100; //release load and wait for compute,
result should be 3

    reset = 1; load = 1; aIn = 5; bIn = 27; #100; //change input values without pressing
load button, result shouldnt change

    reset = 0; load = 1; aIn = 5; bIn = 27; #4; //press reset again
    reset = 1; load = 1; aIn = 5; bIn = 27; #100; //wait, result should be back to 0

    reset = 1; load = 0; aIn = 0; bIn = 4; #4; //a = 0, b = 4  zero case
    reset = 1; load = 1; aIn = 0; bIn = 4; #100; //release load and wait for compute,
result should be 4
    $stop; //complete
end
endmodule

```



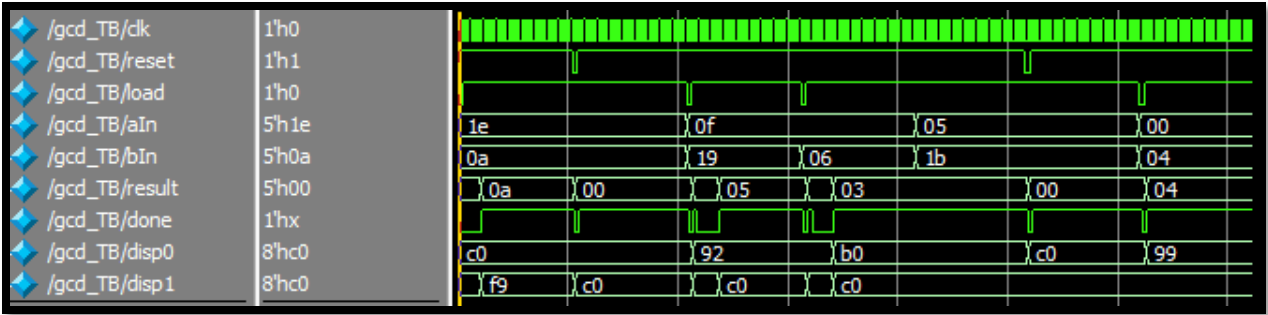


Figure 1: Waveforms shown in Hex format of the top-level test bench

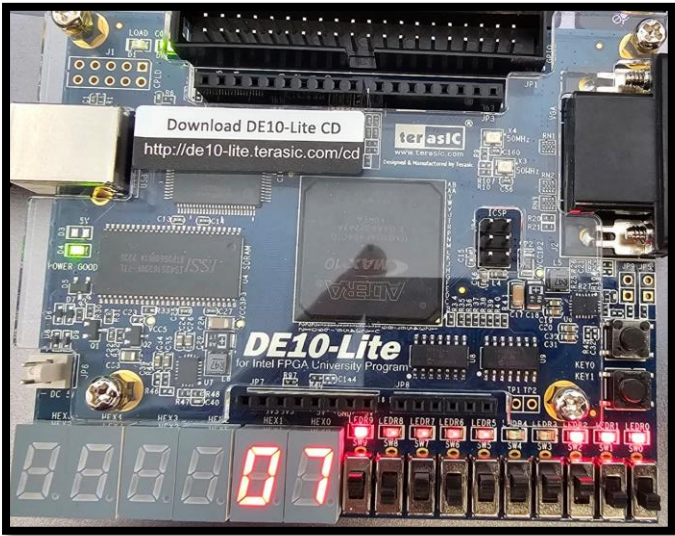


Figure 2: GCD(21,28) running on the MAX10 FPGA

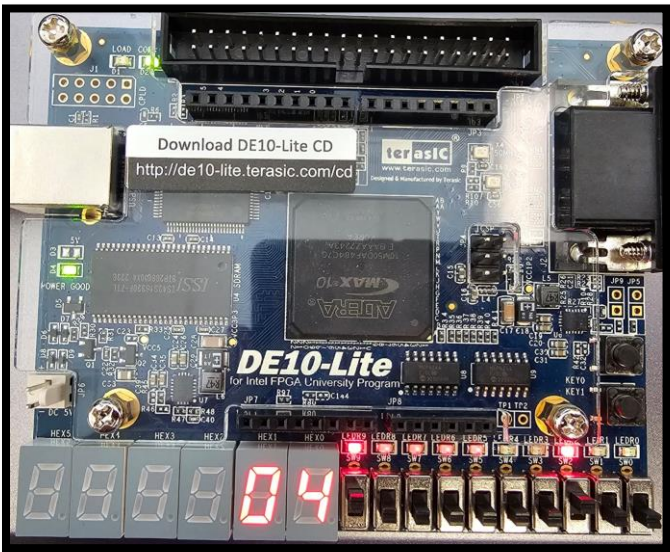


Figure 3: GCD(16,4) running on the MAX10 FPGA

**FPGA and Simulation Implementation and results:**

In conclusion, this activity introduced designing for storage. The important requirement is that inputs are to be ignored if the load button is not pressed. This introduced the requirement of previous states and latch generation in Quartus. If latches are not manually generated, unsafe latch behavior can be introduced which introduced random bugs to the result, see my simulation section for more information. This project thus was a good supplementary project to the last one for introducing sequential logic and state machines.