**California State University, Fullerton**

**Computer Engineering Program**

**EGCP 446 Advanced Digital Design using Verilog HDL Fall 2020**

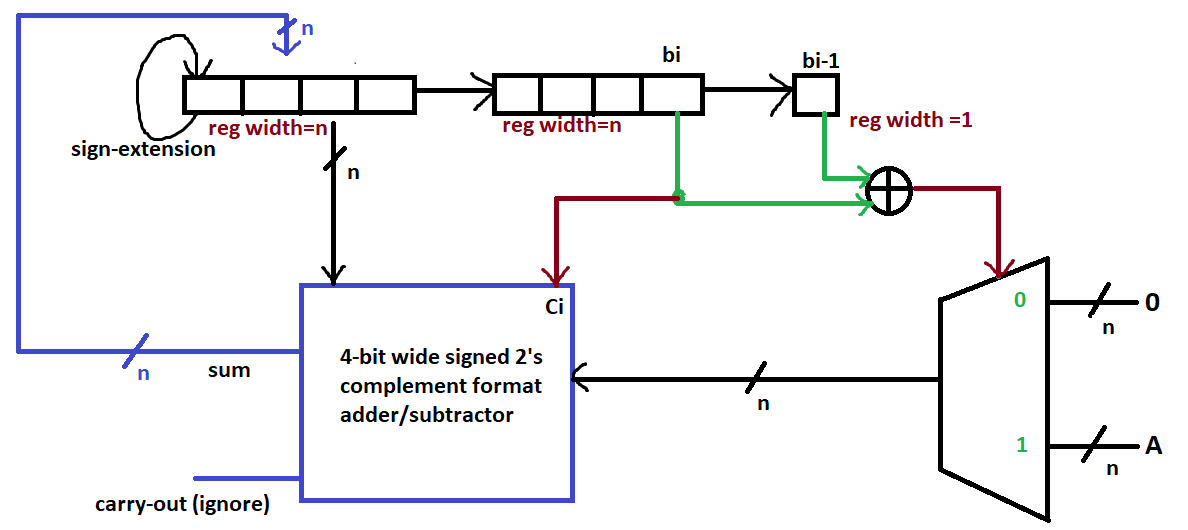
**Exercise #5**

**(Team-based)**

**By Levi Randall and James Samawi**

**(Due on December 4, 2020)**

In this exercise, you will develop the Verilog behavioral model for an 8x8 Booth’s multiplier. The block diagram for the datapath of a nxn Booth’s multiplier is shown below.



Follow the steps given below to complete the exercise:

1. Model a 2-input XOR gate at the behavioral level with a delay of 10 ns. Name the inputs as **U** and **V**. Name the output as **Y**. Place the code for this part here.

**`timescale 1ns / 1ps**

**module xorb(Y, U, V);**

**input U,V;**

**output reg Y;**

**always @ (U or V) begin**

**if (U == 1'b1 & V == 1'b1)**

**begin**

**#10 Y <= 1'b0;**

**end else if (U == 1'b0 & V == 1'b0)begin**

**#10 Y <= 1'b0;**

**end else begin**

**#10 Y <= 1'b1;**

**end**

**end**

**endmodule**

1. Develop the behavioral code for an 8-bit wide 2:1 MUX. Name the select input as **Sel**. Name the data inputs as **D1** and **D2**. Name the output as **Muxout**. The MUX has a delay of 10 ns. Place the code for this part here.

**`timescale 1ns / 1ps**

**module mux8(Muxout, D1, D2, sel);**

**input [7:0] D1;**

**input [7:0] D2;**

**input sel;**

**output reg [7:0] Muxout;**

**always @ (D1 or D2 or sel) begin**

**if (sel == 1'b0)**

**begin**

**#10 Muxout <= D1;**

**end else**

**begin**

**#10 Muxout <= D2;**

**end**

**end**

**endmodule**

1. Develop the behavioral code for an 8-bit wide 2’s complement adder subtractor. Use CLA (global) as the core adder. Name the inputs as **F**, **G** and **Ci**, where Ci is the carry-in. F is input associated with the shift-register in the above diagram. G is the input associated with the MUX. Name the outputs as Sum and Cout. The CLA has a delay of 25 ns. Place the code for this part here.

**`timescale 1ns / 1ps**

**module cla8(Sum, Cout, F,G,Ci,Operate);**

**input [7:0] F;**

**input [7:0] G;**

**input Operate;**

**input Ci;**

**output reg [7:0] Sum;**

**output reg Cout;**

**reg [8:0] t;**

**reg [7:0] inv;**

**always @ (posedge Operate) begin**

**if(Ci) begin**

**inv = -F;**

**t <= inv + G;**

**end**

**else begin**

**t <= F + G;**

**end**

**#25 Sum <= t[7:0];**

**#25 Cout <= t[8];**

**end**

**endmodule**

1. Develop the behavioral code for modeling a 17-bit wide arithmetic right shift register with parallel load capability to model the functionality of the arithmetic right shift register setup shown in the block diagram (total width = n+n+1 = 2n+1). Assume that load has no delay while arithmetic right shift has a delay of 5 ns. Place the code for this part here.

**``timescale 1ns / 1ps**

**module asr17(F,G, Multi, Cell, Product, I, StartI, Load, Shift, Reset);**

**input [7:0] I; // Input From Adder**

**input [7:0] StartI; // Used To Fill Multiplier Initially**

**input Load;**

**input Shift;**

**input Reset;**

**output reg F; // bi - 1**

**output reg G; // bi**

**output reg [7:0] Cell; // Output of Adder Goes Here**

**output reg [7:0] Multi; // Holds Multiplier**

**output reg [16:0] Product;**

**always @ (posedge Shift or posedge Load) begin**

**if(Reset == 1'b1) begin**

**Multi <= 8'b00000000; // Set Multiplier**

**Cell <= 8'b00000000;// Reset Cell**

**F <= 1'b0; // Reset F**

**G <= 1'b0;**

**Product[16] <= 1'b0;**

**Product[15:8] <= 8'b00000000;**

**Product[7:0] <= 8'b00000000;**

**#5;**

**end else if(Load == 1'b1 & Shift == 1'b0) begin**

**Multi <= StartI; // Set Multiplier**

**Cell <= 8'b00000000;// Reset Cell**

**F <= 1'b0; // Reset F**

**G <= StartI[0];**

**#5;**

**end else begin**

**Cell[6:0] <= I[7:1];**

**Cell[7] <= I[7];**

**F <= Multi[0];**

**G <= Multi[1];**

**Multi[6:0] <= Multi[7:1];**

**Multi[7] <= I[0];**

**Product[16] <= Cell[7];**

**Product[15:8] <= Cell;**

**Product[7:0] <= Multi;**

**#5;**

**end**

**end**

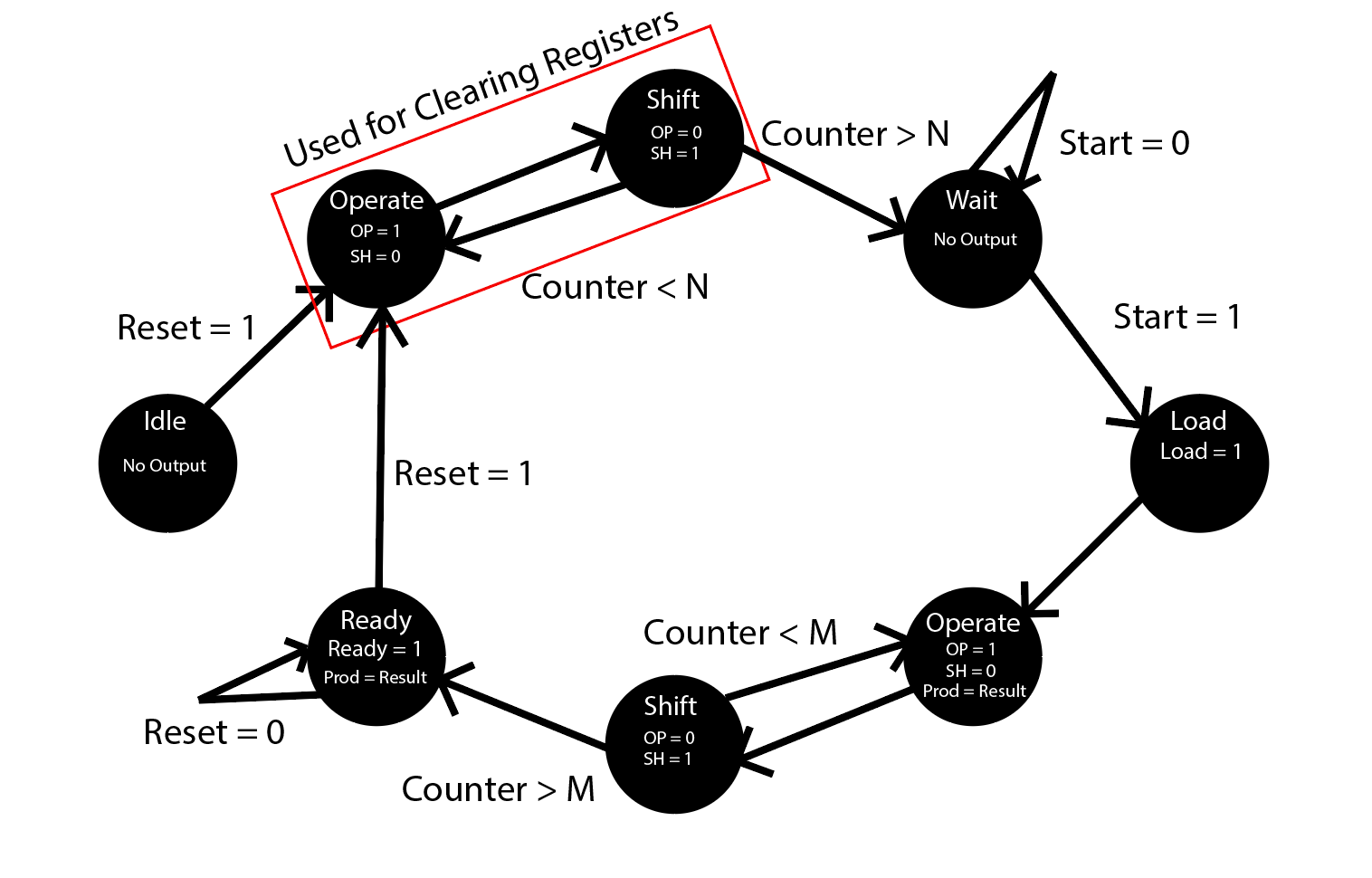
**endmodule**

1. Develop the Verilog code for the 8x8 Booth’s multiplier by instantiating the components developed in the steps above, along with its control logic. Name the multiplicand as **M**, and the multiplier as **N**. Name the product as **Product.**

The inputs associated with the control logic are **Start** and **Reset** (synchronous), which are given by the user.. The control logic produces signals **Load**, **Shift** (*Optional*), **Operate**, and **Ready**.

* When Load=1, parallel loading of the register takes place in that clock cycle.
* (*Use of the Shift signal is optional*) When Shift=1, only the right-arithmetic shift takes place in that clock cycle.
* When Operate=1, addition takes place, followed by right-arithmetic shift in the same clock cycle.
* Start=1 when the user wants to start a multiplication operation. The value of Start is used only in the initial clock cycle of the multiplication process. It is ignored in the subsequent clock cycles until the multiplication process is complete. When Start = 1 in the first clock cycle, and Reset=0: Load the 17-bit wide shift register with the initial values in the first clock cycle (*Load=1* in the first clock cycle). In the subsequent clock cycles make *Load=0,* to continue with Shift/Add-Shift (as the case may be) until the multiplication process is complete. When the multiplication process is complete, Ready=1, and **Product** will contain the final result.
* When Reset=1 (use synchronous reset): Load the 17-bit wide shift register with all zeros, and make Load=Shift=Operate=Ready=0.

Draw a neat, clear and labeled Moore FSM for this part. Place this and the complete Verilog code for this part here.



**`timescale 1ns / 1ps**

**module multiplier(Product, M, N, Reset, Load, Operate, Shift);**

**input [7:0] M; // Multiplicand**

**input [7:0] N; // Multiplier**

**input Reset; // From User**

**input Load; // From CU**

**input Operate;**

**input Shift;**

**output [16:0] Product; // Output**

**wire [7:0] muxout;**

**wire [7:0] sumout;**

**wire [7:0] multiout;**

**wire [7:0] cellout;**

**wire [16:0] productout;**

**wire xorout;**

**wire cout; // Deadwire**

**wire F; // bi-1**

**wire G; // bi**

**wire [7:0] z; // Zeros**

**assign z = 8'b00000000;**

**asr17 a(F, G, multiout, cellout, productout, sumout, N, Load, Shift, Reset); // Produces F, G, Cellout and Takes in Sumout, The Multiplier, Start Signal**

**xorb x(xorout,F,G); // Chooses for Multiplexer**

**mux8 m(muxout,z, M, xorout); // Uses Multiplicand, 0000-0000, and xorout**

**cla8 c(sumout, cout, muxout, cellout, G, Operate); // Produces the Sum, Cout, and Takes in Muxout, Cellout, G as add or Sub**

**assign Product = productout;**

**endmodule**

**//USED AS CONTROL LOGIC**

**`timescale 1ns / 1ps**

**module controlunit(Ready, Load, Operate, Shift, Start, clk, Reset);**

**input Start;**

**input clk;**

**input Reset;**

**reg [4:0] done; // Counter Used For Indexing and Knowing When Done**

**reg latch;**

**reg go;**

**output reg Ready;**

**output reg Load;**

**output reg Operate;**

**output reg Shift;**

**always @ (posedge clk) begin**

**if(latch !== 1'b1) begin**

**done <= 5'b00000;**

**end**

**if(clk) begin**

**latch <= "1";**

**end**

**if(Reset == 1'b1 & go == 1'b1)begin**

**Shift <= "0";**

**Operate <= "0";**

**Load <= "0";**

**Ready <= "0";**

**go <= "0";**

**done <= 5'b00000;**

**end else if(done >= 5'b01000)begin // Signals that Values have been Flushed**

**go <= "1";**

**end else if (Ready !== 1'b1) begin // Flushes Values Of High Impedance Out Of Registers**

**if(done[0] == 1'b1) begin**

**Shift <= "0";**

**Operate <= "0";**

**Load <= "1";**

**end else begin**

**Load <= "0";**

**Shift <= "0";**

**Operate <= "1";**

**end**

**done = done + clk; // Counts How Many Times Loop has Been Done**

**go <= "0";**

**end**

**if(Reset == 1'b1 & go == 1'b1)begin**

**Shift <= "0";**

**Operate <= "0";**

**Load <= "0";**

**Ready <= "0";**

**go <= "0";**

**done <= 5'b00000;**

**end else if (Start == 1'b1 & go == 1'b1 & Ready !== 1'b1) begin // Multiplication Can Begin**

**if(done[0] == 1'b1) begin**

**Load <= "0";**

**Operate <= "0";**

**Shift <= "1";**

**end else begin**

**Load <= "0";**

**Shift <= "0";**

**Operate <= "1";**

**end**

**done = done + clk; // Counts How Many Times Loop has Been Done**

**end**

**if(Reset == 1'b1 & go == 1'b1)begin**

**Shift <= "0";**

**Operate <= "0";**

**Load <= "0";**

**Ready <= "0";**

**go <= "0";**

**done <= 5'b00000;**

**end else if(done >= 5'b11011)begin // Signals that Multiplication is Done**

**Ready <= "1";**

**end**

**end**

**endmodule**

**// Top Level Module That Contains Control Unit and Multiplier**

**`timescale 1ns / 1ps**

**module top\_multiplier(Product, Ready, N, M, Start, Reset, clk);**

**input [7:0]N; // Multiplicand**

**input [7:0]M; // Multiplier**

**input Start; // User Controlled Values**

**input Reset;**

**input clk; // System Clock**

**output [16:0]Product; // Output**

**output Ready; // Tells When Output Should be Observed**

**wire Ready, Load, Operate, Shift; // Internal Signals**

**controlunit cu(Ready, Load, Operate, Shift, Start,clk, Reset);**

**multiplier mlti(Product, M, N, Reset, Load, Operate, Shift);**

**endmodule**

1. Write a Verilog testbench for your design. Test for the following cases:

* A = 1; B = 0
* A reset case in which Reset=1 in the third clock cycle
* A = 100; B = 99
* A = **˗**100; B = 99
* A = 100; B = **˗** 99

Place the code below. Also, include **clear, legible and well-sized screenshots** for all of these test cases. **Highlight and markup the inputs and output in each test case**.

**A = 1; B = 0**

**`timescale 1ns / 1ps**

**module top\_multiplier\_tb;**

**reg [7:0] M; // Multiplicand**

**reg [7:0] N; // Multiplier**

**reg Start;**

**reg Reset; // From User**

**reg clk;**

**wire [16:0] Product; // Output**

**wire Ready; // Tells When Output Should be Observed**

**top\_multiplier dut(.Product(Product), .Ready(Ready), .M(M), .N(N), .Start(Start), .Reset(Reset), .clk(clk));**

**initial begin**

**#0 Reset = 1'b1;**

**#0 clk = 1'b0;**

**#1 N = 8'b00000100; // B = 0**

**#1 M = 8'b00000101; // A = 1**

**repeat(8) #50 clk = ~clk;**

**#10 Reset = 1'b0;**

**#0 Start = 1'b1;**

**repeat(60) #50 clk = ~clk;#50 clk = 1'b1;**

**$finish;**

**end**

**endmodule**



**Reset = 1; A = 13; B = 8**

**`timescale 1ns / 1ps**

**module top\_multiplier\_tb;**

**reg [7:0] M; // Multiplicand**

**reg [7:0] N; // Multiplier**

**reg Start;**

**reg Reset; // From User**

**reg clk;**

**wire [16:0] Product; // Output**

**wire Ready; // Tells When Output Should be Observed**

**top\_multiplier dut(.Product(Product), .Ready(Ready), .M(M), .N(N), .Start(Start), .Reset(Reset), .clk(clk));**

**initial begin**

**#0 Reset = 1'b1;**

**#0 clk = 1'b0;**

**#1 N = 8'b00001000; // B = 8**

**#1 M = 8'b00001101; // A = 13**

**repeat(8) #50 clk = ~clk;**

**#10 Reset = 1'b0;**

**#0 Start = 1'b1;**

**repeat(6) #50 clk = ~clk;#50 clk = 1'b1;**

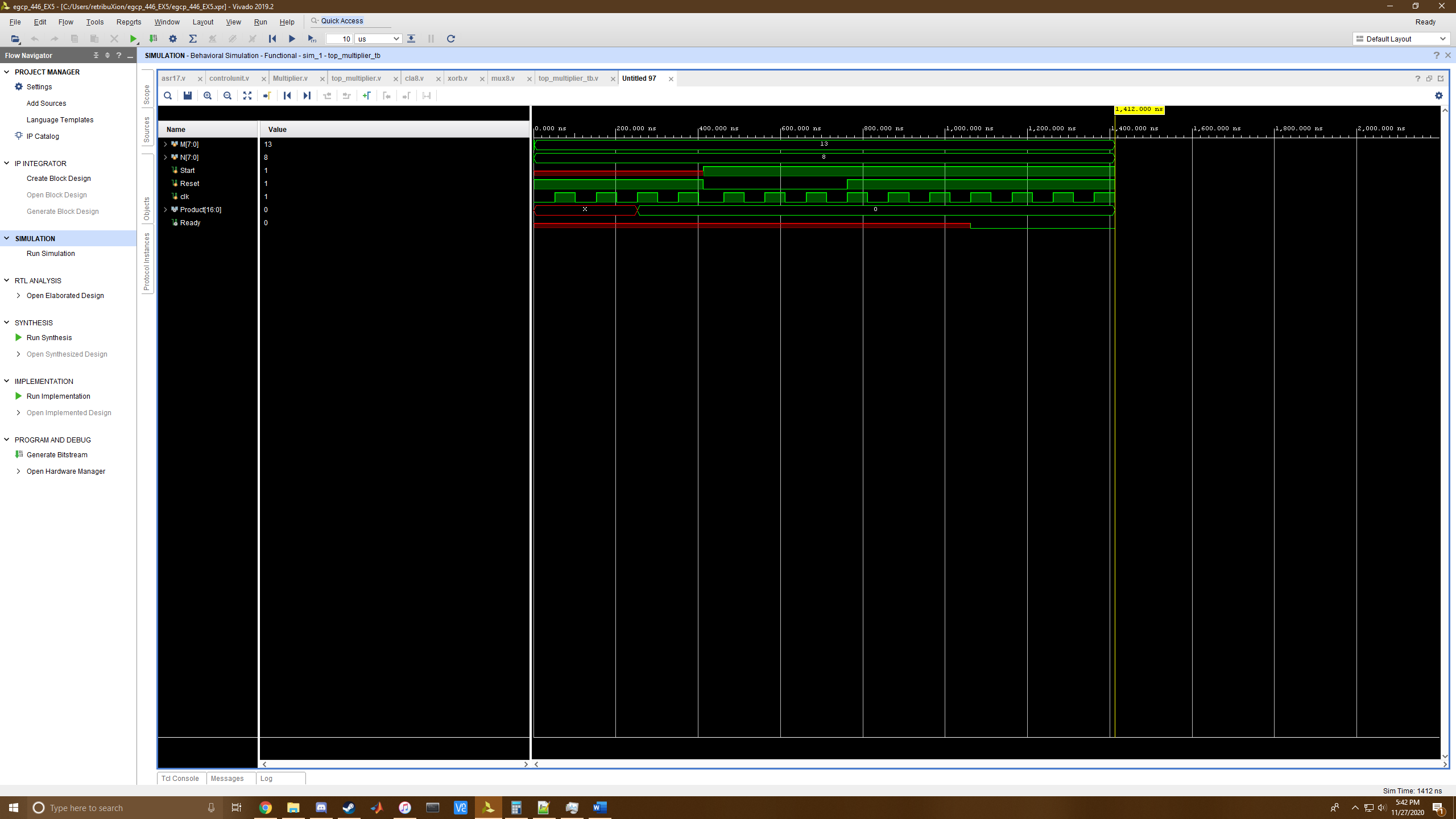
**#0 Reset = 1'b1;**

**repeat(12) #50 clk = ~clk;#50 clk = 1'b1;**

**$finish;**

**end**

**endmodule**



**A = 100; B = 99**

**`timescale 1ns / 1ps**

**module top\_multiplier\_tb;**

**reg [7:0] M; // Multiplicand**

**reg [7:0] N; // Multiplier**

**reg Start;**

**reg Reset; // From User**

**reg clk;**

**wire [16:0] Product; // Output**

**wire Ready; // Tells When Output Should be Observed**

**top\_multiplier dut(.Product(Product), .Ready(Ready), .M(M), .N(N), .Start(Start), .Reset(Reset), .clk(clk));**

**initial begin**

**#0 Reset = 1'b1;**

**#0 clk = 1'b0;**

**#1 N = 8'b01100011; // B = 99**

**#1 M = 8'b01100100; // A = 100**

**repeat(8) #50 clk = ~clk;**

**#10 Reset = 1'b0;**

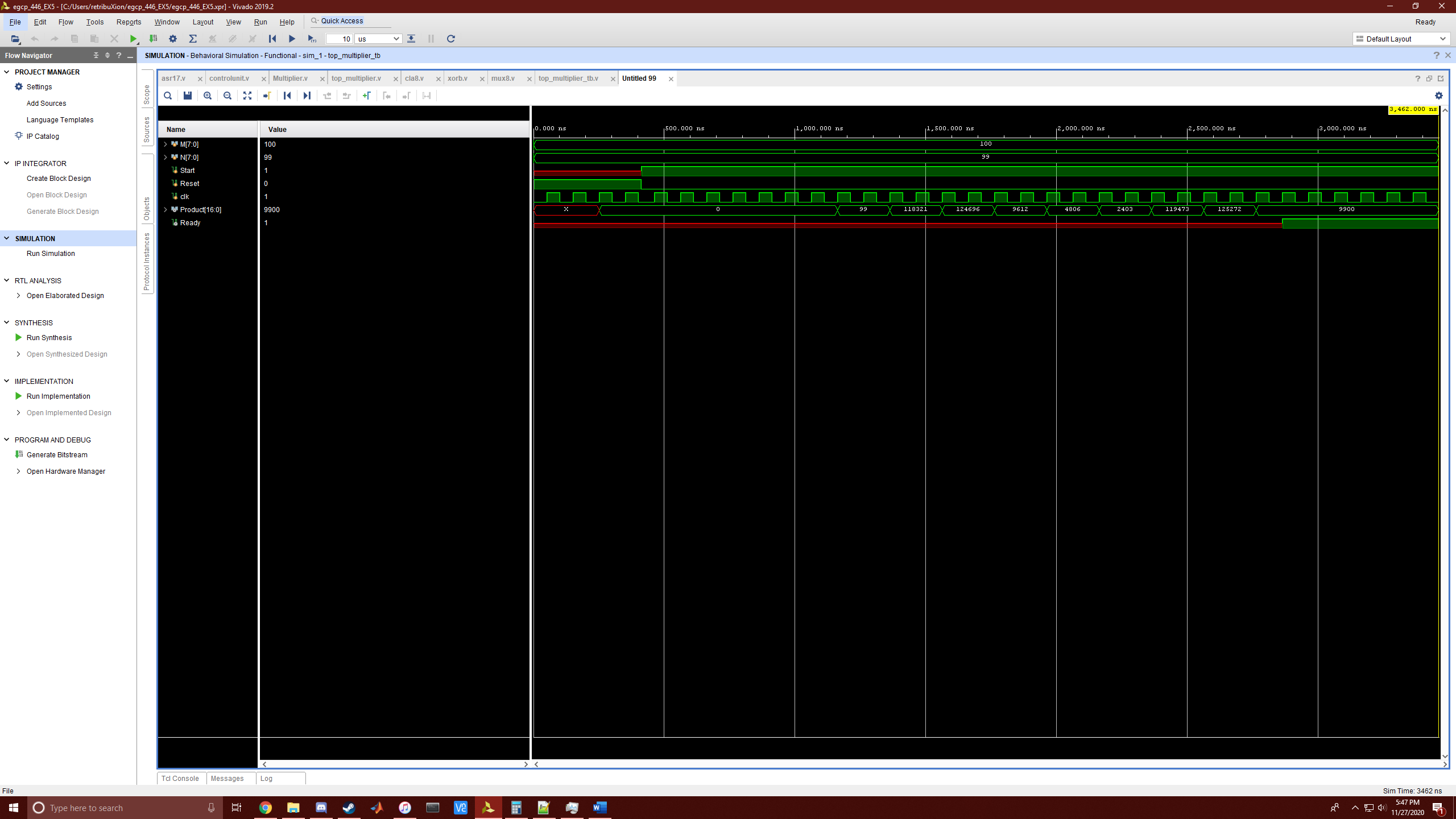
**#0 Start = 1'b1;**

**repeat(60) #50 clk = ~clk;#50 clk = 1'b1;**

**$finish;**

**end**

**endmodule**



**A = -100; B = 99**

`**timescale 1ns / 1ps**

**module top\_multiplier\_tb;**

**reg [7:0] M; // Multiplicand**

**reg [7:0] N; // Multiplier**

**reg Start;**

**reg Reset; // From User**

**reg clk;**

**wire [16:0] Product; // Output**

**wire Ready; // Tells When Output Should be Observed**

**top\_multiplier dut(.Product(Product), .Ready(Ready), .M(M), .N(N), .Start(Start), .Reset(Reset), .clk(clk));**

**initial begin**

**#0 Reset = 1'b1;**

**#0 clk = 1'b0;**

**#1 N = 8'b01100011; // B = 99**

**#1 M = 8'b10011100; // A = -100**

**repeat(8) #50 clk = ~clk;**

**#10 Reset = 1'b0;**

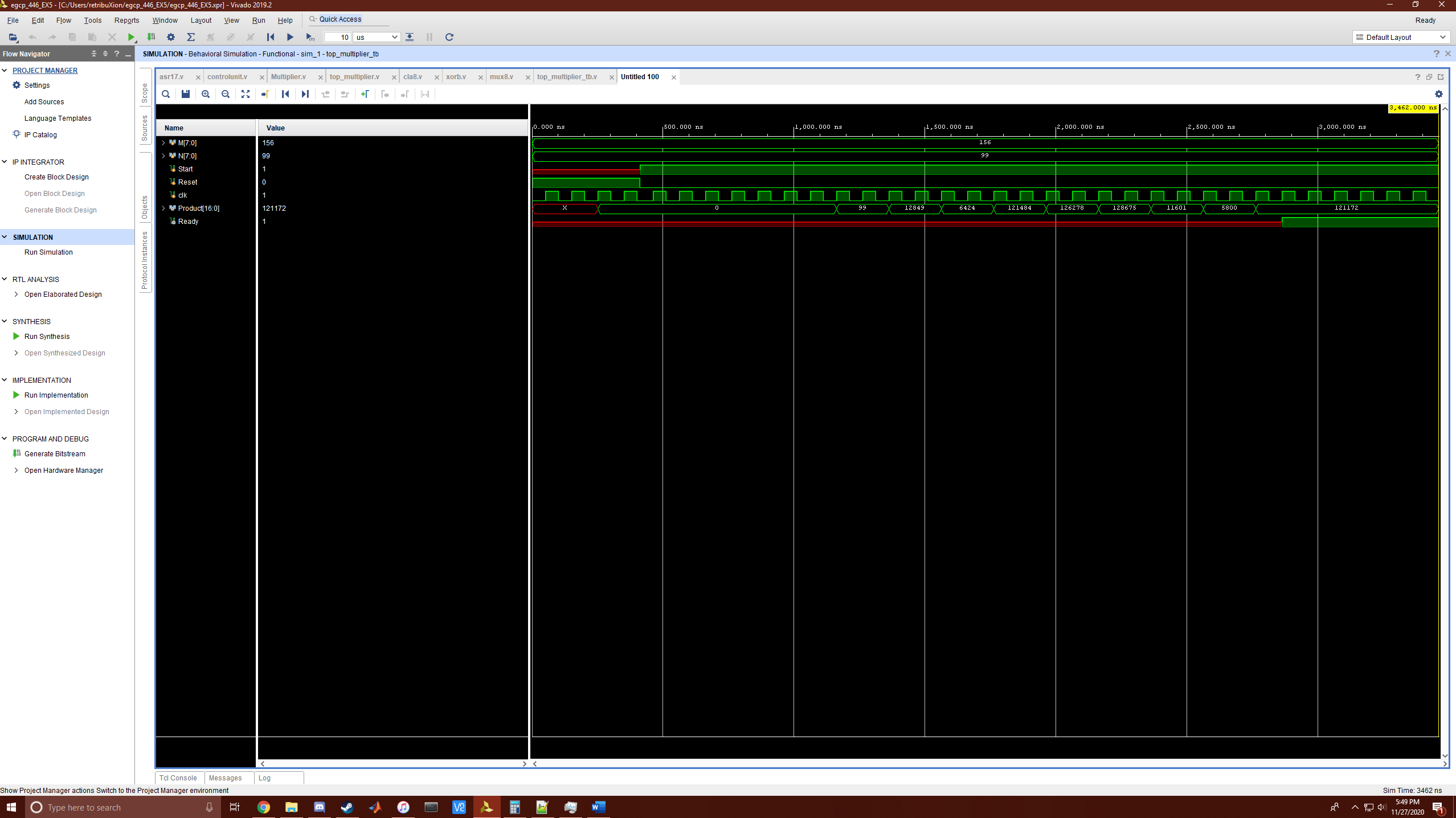
**#0 Start = 1'b1;**

**repeat(60) #50 clk = ~clk;#50 clk = 1'b1;**

**$finish;**

**end**

**endmodule**



**A = 100; B = -99**

**`timescale 1ns / 1ps**

**module top\_multiplier\_tb;**

**reg [7:0] M; // Multiplicand**

**reg [7:0] N; // Multiplier**

**reg Start;**

**reg Reset; // From User**

**reg clk;**

**wire [16:0] Product; // Output**

**wire Ready; // Tells When Output Should be Observed**

**top\_multiplier dut(.Product(Product), .Ready(Ready), .M(M), .N(N), .Start(Start), .Reset(Reset), .clk(clk));**

**initial begin**

**#0 Reset = 1'b1;**

**#0 clk = 1'b0;**

**#1 N = 8'b10011101; // B = -99**

**#1 M = 8'b01100100; // A = 100**

**repeat(8) #50 clk = ~clk;**

**#10 Reset = 1'b0;**

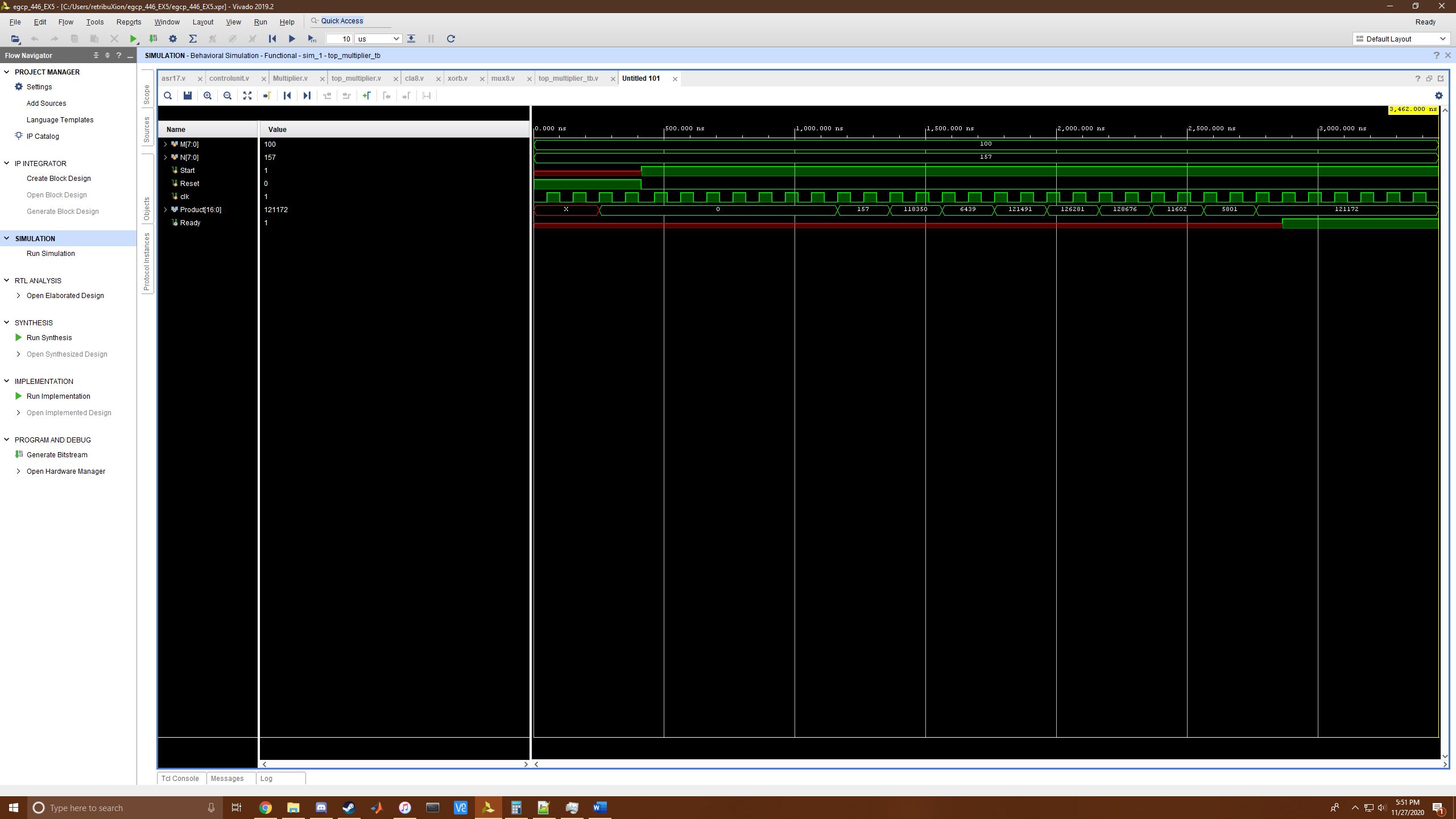
**#0 Start = 1'b1;**

**repeat(60) #50 clk = ~clk;#50 clk = 1'b1;**

**$finish;**

**end**

**endmodule**



Submit this document including all the code and screenshotson Canvas by 11:45 p.m. on the deadline. **The file name must contain the names of both team members**. Include the following also:

* A statement identifying the problem that you are trying to solve
* Methods and procedures used to complete the assignment
* Discussion of the results
* Technical challenges/difficulties you faced and how you overcame them
* Your conclusions

**Problem Statement:**

In this exercise, we attempted to create a Boothe’s multiplier in Verilog. This multiplier has a very specific compared to other multipliers and due to its conceptual nature, it is being coded on the behavioral level within Verilog. In total, all components were create on the behavioral level which included a 2n+1 bit wide right shifting register, a single bit XOR gate, an 8bit wide 2to1 Multiplexer, an 8 bit CLA Adder/Subtractor, and an external control unit that creates necessary signals for the multiplier to operate.

**Methodology:**

To begin, we started with the smallest components first and gradually made bigger components. The XOR was first to be coded and here we learned that registers were going to be used during almost everywhere in the project. We then created the MUX and then created the CLA Adder. Due to previous projects and our Term Project, both James and I, have already had experience both with Carry Look Ahead Adders, Adder/Subtractor Circuits, and Multiple Bit Wide 2to1 Multiplexers and thus, finished the beginning without any major issues.

Here we moved wrote test benches for each component to verify their use. We then created a 17bit wide Shift Register and at first had few problems but later ran into many. A new component was made to then house all the components that we labeled as the multiplier. We discovered many issues with the register and realized that the whole circuit almost had a pipeline effective if the enable signals weren’t cautiously conducted.

After many problems that will be discussed later, we got that circuit working and created another component of equal level with our multiplier to supply the necessary Operate, Shift, Load, and Ready signals called the control unit. We setup a complicated module that used if statements and a counter to index called “done” to know when our circuit was done at certain intervals. The control unit also supplied these other signals in an alternating fashion to effectively pipeline the signals and would shut off the multiplier when the correct result was achieved. A final component was made to house these two level circuits and took in user inputs for test bench purposes. The code functions effectively and works successfully as a Boothe’s Multiplier.

**Discussion:**

Our results were exactly as expected. The Multiplier would iterate slowly over the numbers we had given and after enough cycles we could watch as the number slowly turned into the answer we were expecting. It also functioned effectively when given a negative number. The negative numbers did originally have a problem where the Most Significant Bit was not being sign extended but this was due to a small variable assignment mistake.

**Technical Challenges:**

A large majority of this lab’s difficulty lies within managing signals as if the component was a multistage computer. The registers gave us a large amount of trouble trying to figure out how to initialize them to have all 0s so that our conditional statements would work appropriately and ended up using a reset followed by a value load in order to flood the system with 0s initially so the circuit could work, otherwise the system would only reflect XXXX values. Additionally, we attempted to not use the shift variable for shifting and found that it was too difficult without it. Our answers without it would be almost perfect but would always be shifted one time off the number it should have been. Other unforeseen problems were the values in each always @ block not updating immediately, meaning multiple if else statements had to be used in conjunction with other if statements instead of simply updating one variable that could be reflected in pre-existing if statements.

**Conclusion:**

In all, this project was very challenging and was unlike previous exercises. The heavy reliance on registers made the flow of variables react different and made everything react at the pace of the clock. The overall structure of the circuit was also very challenging, learning how the circuit functioned was a good exercise all on its own. Another interesting task was how almost all components were done with behavioral logic which made even simple XOR gates more of an adversary. We both learned much more about Verilog behavioral coding as well as the registers variable type.