**California State University, Fresno**

**Lyles College of Engineering**

**Electrical and Computer Engineering Department**

**TECHNICAL REPORT**

**Experiment Title:** 8x8 Multiplier Instantiations

**Course Title:** ECE 176 Computer Aided Design

**Date Submitted:** November 13, 2014

**Honor Code Statement:**

**“I have done my own work and have neither given nor received**

**unauthorized assistance on this work.”**

|  |  |
| --- | --- |
| **Prepared By:** |  |
| **Christopher Hays** |  |
|  |  |
| **Signature:** |  |

**INSTRUCTOR SECTION**

**Comments:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**TABLE OF CONTENTS**

**Section:** **Page:**

Title Page ……………………………………………………………………...…. 1.

Table of Contents ……………………………………………………………….... 2.

1. Statement of Objectives ……………………………………………………….. 3.

2. Theoretical Background …………………….…………………………………. 3.

3. Experimental Procedure ………………………………………………………. 4.

3.1 Equipment Used ……………………………………………………... 4.

3.2 Laboratory Procedure ……………………………………………...… 4.

4. Analysis ……………………………………………………………………...... 6.

4.1 Testbench File ………………………………………………………… 6.

4.2 Simulation ………………………………………………………… 6.

5. Conclusions ……………………………………………………………………. 7.

6. Appendix ………………………………………………………………………. 8.

6.1 Verilog Code …………………………………………………………. 8.

**1. STATEMENT OF OBJECTIVES**

The objective of this assignment was to design an 8x8 hardware multiplier using Verilog modeling and instantiations. The design was to use all of the individual modules that have been worked on up to this point to implement piecewise binary multiplication of two 8-bit numbers. The previous modules included were a 4-bit multiplexer, 4x4 multiplier, 2-bit counter, 8-bit shifter, 16-bit adder, 16-bit register, seven-segment LED controller and the multiplier controller state machine. The design was then synthesized with the Cyclone II as the target hardware and an RTL simulation was conducted in ModelSim software to verify the operation of the multiplier.

**2. THEORETICAL BACKGROUND**

Finite state machines are commonly used to control hardware processes that involve multiple steps or conditional instructions. They are typically custom designed to the specific hardware and design needs, starting with state diagrams and tables. This design uses a state machine to control the aspects of partial product multiplication of two 8-bit numbers.

Partial product multiplication multiplying sections of the multiplier and multiplicand, shifting the result by the correct number of bits, and then adding all of the intermediate results to obtain the final product. This multiplier takes two 8-bit numbers and multiplies them in four phases:

The lowest 4 bits of each number are multiplied and shifted by 0 bits (2^0).

The lowest 4 bits of input A are multiplied with the highest 4 bits of input B and shifted by 4 bits (2^4).

The highest 4 bits of input A are multiplied with the lowest 4 bits of input B and shifted by 4 bits (2^4).

The highest 4 bits of both inputs are multiplied and shifted by 8 bits (2^8).

result[15..0] = a[7..0]\*b[7..0]

= ((a[7..4]\*b[7..4])\*2^8)

+((a[7..4]\*b[3..0])\*2^4)

+((a[3..0]\*b[7..4])\*2^4)

+((a[3..0]\*b[3..0])\*2^0)

The controller is in the IDLE state when it is ready to begin multiplication, the LSB state for the lower bits, the MID state for the cross-multiplication, and the MSB state for the higher bits. A final CALC\_DONE state is used to signal that the product is complete and ready. The control unit organizes where the data goes as well as when and how it is processed. It takes inputs from the main module which are the start bit, the clock, and a reset bit. The various hardware components needed are instantiated within the Verilog file; their inputs and outputs connected with wires that belong to the main module.

**3. EXPERIMENTAL PROCEDURE**

**3.1 Equipment Used**

Altera ModelSim Software

Altera Quartus II

**3.2 Laboratory Procedure**

A new project was created in Quartus II with the Cyclone II as the target hardware and included the previously written Verilog files for the multiplexer, multiplier, adder, counter, shifter, register, control unit, and seven-segment control. The mult8x8 module was declared with two 8-bit data inputs (dataa and datab), and 1-bit start, reset, and clock inputs. Output registers were created for the done\_flag, the seven-segment control output and the final multiplication 16-bit output.

Next, the previously mentioned modules were instantiated in order to be connected as shown in Figure 2. Wire variables were created to serve as the connections between the instantiations. The module objects were connected, specifying each connection by name using the “dot” method of instantiation. The data inputs were split into high/low 4-bit groups and connected to the multiplexers. The output of these were connected to the 4x4 multiplier to handle the intermediate multiplication. This output was then connected to the shifter, then through the adder, then to the register. The adder also had the current register output as its second input. In the block diagram generated by Quartus II, the adder is not a standalone unit; the register inputs are simply an addition of the output plus the shifter output.

The control unit was connected to the parent module’s clock, start, and reset lines. The count input was driven by the counter unit which also ran on the same clock. The input\_sel and shift\_sel outputs were connected to the multiplexers and the shifter so they would be controlled by the current state of the control unit. Clk\_ena and sclr\_n connected to the register inputs and the state\_out drove the input of the seven-segment controller. The seven-segment controller outputs were simply connected to the parent module main outputs corresponding to each line.

The parent module main outputs posed an interesting problem; they need to be declared as reg data type in order to hold their value, but reg variables cannot used in instantiation as was needed for the register, the seven-segment controller, as well as the done flag.



Figure 1: Instantiation Error

Temporary wire variables were made to use during the instantiations and a procedural block was created to update the main outputs on every positive clock edge. This ensured that the output would be current and work correctly. A testbench file to examine the output of the module was supplied by the instructor and code for all modules is listed in the Appendix. Successful analysis and synthesis is shown in Figure 3.

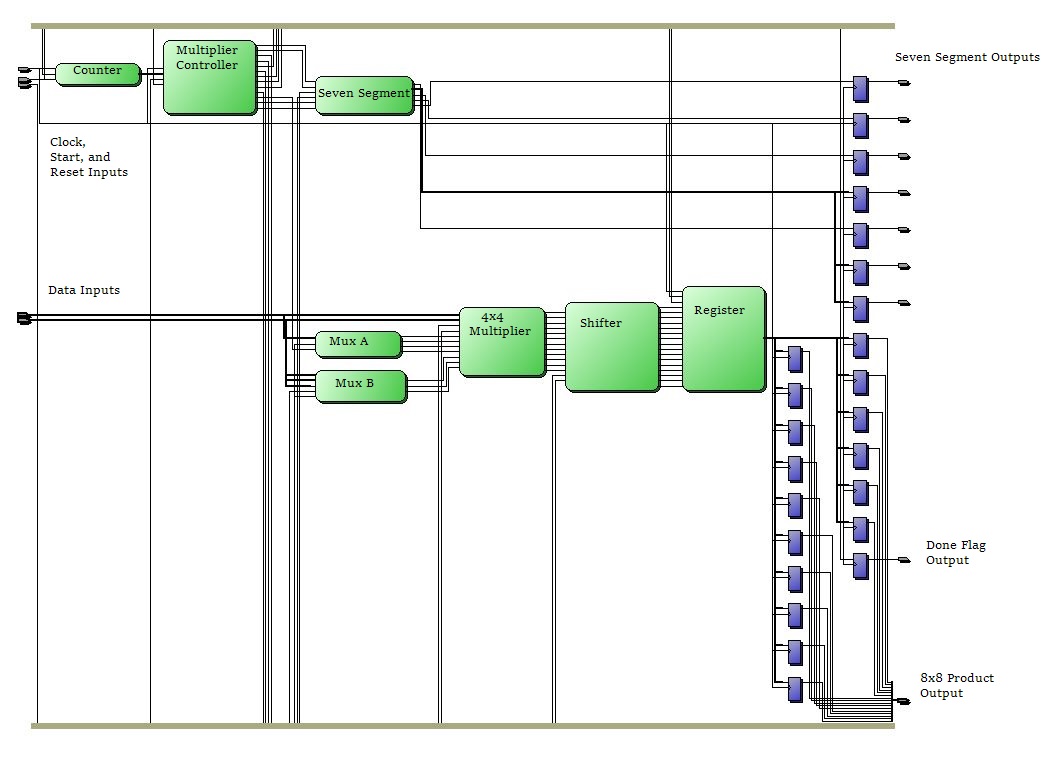


Figure 2: 8x8 Multiplier Block Diagram

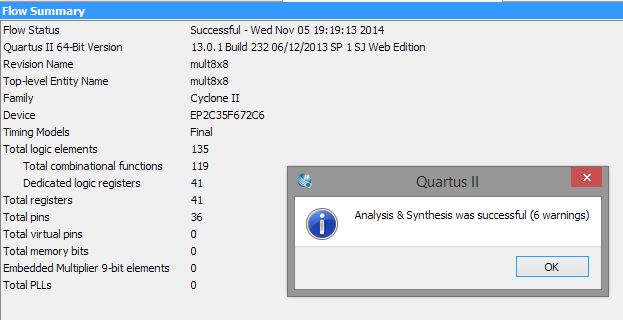


Figure 3: Successful Analysis and Synthesis

**4. ANALYSIS**

**4.1 Testbench File**

The mult8x8\_tb.v testbench file first created reg variables to hold the input values and wire variables to hold the output values of the mult8x8 module. The module was then instantiated and connected to these inputs and outputs. Five simultaneous procedural blocks control the testing of the module. The first initializes a clock signal at zero and sets it to alternate every 25 nanoseconds, for a period of 50 nanoseconds. The second block drives the reset signal high for the first 50 nanoseconds of the simulation, then drives it low, ensuring that the registers are clear and ready to process the multiplication.

The third procedural block drives the start line high for 50 nanoseconds then sets it to zero, starting the control unit and the multiplication. It then waits until the negative edge of the done\_flag and pulses the start line again to begin the cycle again. The fourth block controls what is on the data lines. Both data busses begin with a value of 0xFF. At the negative edge of the done\_flag, 0x24 is added to dataa and 0x51 is added to datab in time for the process to start over, multiplying the new values. The final block stops the simulation after 1200 nanoseconds, or roughly 3 multiplication cycles; without this the simulation was running forever.

**4.2 Simulation**

In Figure 4 it is shown that before the start line is pulsed, the data inputs both read 0xFF, the seven-segment output corresponds to a “0” shaped display, and the register reads 0x0000. When the start line drops the seven-segment controller now reads “1” indicating that it is in the LSB state and is multiplying the least significant 4 bits of the two inputs. On the next positive clock edge, the register updates to 0xE1 and the state output is now “2”. The state output stays as “2” for two clock cycles as it controls the cross-multiplication; the register output updates to 0x0EF1 then 0x1D01, verifying this. At this point the state output reads “3” and the state machine is in the MSB state, multiplying the highest 4 bits of the inputs. State output of “4” is reached, the register is updated with the final value of 0xFE01 and the done\_flag is pulsed to indicate completion of the process.

The testbench begins to reset everything on the negative edge of the done\_flag. The data inputs are updated, the register is cleared, and the start line is pulsed to begin the next round of multiplication. At this point the data lines read 0x17 and 0x32. The same cycles occur and the final result of this multiplication is 0x047E, which is correct. The third round of multiplication begins with the new values of 0x2F and 0x65. The final product of this third round is 0x128B which is also correct. Simulation ends in the middle of cycle four.

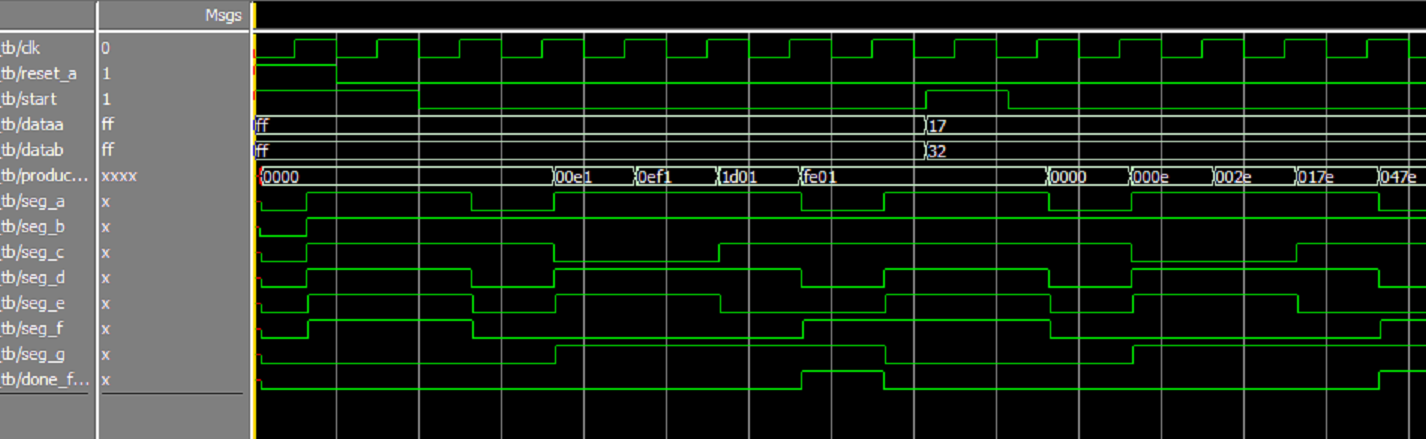


Figure 4: Input and Output Waveforms

**5. CONCLUSIONS**

Module instantiation and linking is a key skill in top level Verilog design projects. Completed modules can be used repeatedly, saving time and even providing an abstraction layer above the low-level functions of specific modules if desired. Connecting pre-designed modules is very easy and allows the designer to concentrate on the higher level aspects of the problem. This assignment demonstrated how complicated hardware multiplication actually is compared to addition, which explains why multiplication instructions take much longer to execute. Partial product multiplication is a useful way to use shifters and adders to aid in the calculations.

Analysis of the testbench file and output waveforms confirmed that the design was working as intended. Partial products were created via the multiplexers, multiplier, and shifter; intermediate and final results are stored in the register and all values were determined to be correct. The seven-segment display controller always contained the output pattern that corresponded with the current state. Actually assigning pins and programming the Cyclone II hardware would have given a constant visual representation of the current state. Upon completion the done flag was raised which could be used as a possible control signal for more hardware further down the line. This assignment demonstrated multiple hardware modules working together and controlled by a state machine, which is the basis of many practical applications.

**6. APPENDIX**

**6.1 Verilog Code**

**Multiplexer:**

// Christopher Hays

// ECE 176

// 4-bit, 2:1 multiplexer

// mux4.v

// Declare module

module mux4 (mux\_in\_a, mux\_in\_b, mux\_sel, mux\_out);

// Declare inputs and outputs

input [3:0] mux\_in\_a, mux\_in\_b;

input mux\_sel;

output reg [3:0] mux\_out;

// Update the multiplexer output when any of the inputs change

always@(mux\_sel, mux\_in\_a, mux\_in\_b)begin

if (mux\_sel)

mux\_out <= mux\_in\_b;

else

mux\_out <= mux\_in\_a;

end

endmodule // End module

**Multiplier:**

// Christopher Hays

// ECE 176 Assignment 4

// 4x4 Multiplier

// mult4x4.v

// Declare module

module mult4x4 (dataa, datab, product);

//Declare inputs and outputs

input [3:0] dataa, datab;

output reg [7:0] product;

// Update the product output when any of the inputs change

always @(dataa, datab)begin

product <= dataa \* datab;

end

endmodule // End module

**Counter:**

// Christopher Hays

// ECE 176

// 2-bit counter with asynchronous reset

// counter.v

// Declare module, inputs, and outputs

module counter(input clk, aclr\_n, output reg [1:0] count\_out);

// Counter increments synchronously, clears asynchronously, counts 0 to 3

always @(posedge clk, negedge aclr\_n)begin

if (~aclr\_n)

count\_out <= 0;

else

count\_out <= count\_out + 1;

end

endmodule // End module

**Shifter:**

// Christopher Hays

// ECE 176

// Shifter using concatenation, essentially multiplying by 2^0, 2^4, or 2^8

// shifter.v

// Declare module

module shifter(inp, shift\_cntrl, shift\_out);

// Declare inputs and outputs

input [7:0] inp;

input [1:0] shift\_cntrl;

output reg [15:0] shift\_out;

// Shift on any change in input

always @(inp, shift\_cntrl) begin

if (shift\_cntrl == 0 || shift\_cntrl == 3)

shift\_out = {8'H00, inp};

else if (shift\_cntrl == 1)

shift\_out = {4'H0, inp, 4'H0};

else

shift\_out = {inp, 8'H00};

end

endmodule // End module

**Adder:**

// Christopher Hays

// 16-bit adder

// adder.v

// Declare module

module adder (dataa, datab, sum);

// Declare inputs and outputs

input [15:0] dataa;

input [15:0] datab;

output [15:0] sum;

// Continuous assignment statement

assign sum = (dataa + datab);

endmodule // End module

**Register:**

// Christopher Hays

// 16-bit register with synchronous control

// reg16.v

// Declare module, inputs, and output

module reg16 (input clk, clk\_ena, sclr\_n, input [15:0] datain,

output reg [15:0] reg\_out);

// Synchronously update or clear the register

always@(posedge clk)begin

if (clk\_ena)begin

if (~sclr\_n)

reg\_out <= 16'H0000;

else

reg\_out <= datain;

end

end

endmodule // End module

**Control Unit:**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

// ECE 176 Fall 2014

// Module Name: mult\_control

// File Name: mult\_control.v

// Module Function: This file contains the state machine control logic for the

// multiplier

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Begin module declaration for "mult\_control"

module mult\_control (

input clk, reset\_a, start, // Declare control inputs

input [1:0] count,

output reg [1:0] input\_sel, shift\_sel, // Declare output control signals

output reg [2:0] state\_out,

output reg done, clk\_ena, sclr\_n

);

// Declare "current\_state" and "next\_state" to be state variables

reg [2:0] current\_state, next\_state;

// Declare parameters for states consisting of 6 values: "idle", "lsb", // "mid", "msb", "calc\_done" and "err"

parameter idle=0, lsb=1, mid=2, msb=3, calc\_done=4, err=5;

// Create sequential process to control state transitions by making

// current\_state equal to next state on rising edge transitions;

// Use asynchronous clear control

always @(posedge clk, posedge reset\_a) begin

if (reset\_a)

current\_state <= idle;

else

current\_state <= next\_state;

end

// Create combinational process & case statement to determine next\_state based // on current state and inputs

always @ (\*) begin

case (current\_state)

idle :

if (start)

next\_state = lsb;

else

next\_state = idle;

lsb :

if (count == 0 && start == 0)

next\_state = mid;

else

next\_state = err;

mid :

if (start == 0 && count == 2)

next\_state = msb;

else if (start == 0 && count == 1)

next\_state = mid;

else

next\_state = err;

msb :

if (count == 3 && start == 0)

next\_state = calc\_done;

else

next\_state = err;

calc\_done :

if (!start)

next\_state = idle;

else

next\_state = err;

err :

if (start)

next\_state = lsb;

else

next\_state = err;

endcase

end

// Create process for Mealy output logic for input\_sel, shift\_sel, done, // clk\_ena and sclr\_n (outputs function of inputs and current\_state)

always @ (\*) begin

// Initialize outputs to default values so case only covers when they // change

input\_sel = 2'bxx;

shift\_sel = 2'bxx;

done = 1'b0;

clk\_ena = 1'b0;

sclr\_n = 1'b1;

case (current\_state)

idle :

if (start) begin

clk\_ena = 1'b1;

sclr\_n = 1'b0;

end

lsb :

if (count == 0 && start == 0) begin

input\_sel = 2'd0;

shift\_sel = 2'd0;

clk\_ena = 1'b1;

end

mid :

if (count == 2 && start == 0) begin

input\_sel = 2'd2;

shift\_sel = 2'd1;

clk\_ena = 1'b1;

end

else if (count == 1 && start == 0) begin

input\_sel = 1'd1;

shift\_sel = 1'd1;

clk\_ena = 1'b1;

end

msb :

if (count == 3 && start == 0) begin

input\_sel = 2'd3;

shift\_sel = 2'd2;

clk\_ena = 1'b1;

end

calc\_done :

if (!start)

done = 1'b1;

err :

if (start) begin

clk\_ena = 1'b1;

sclr\_n = 1'b0;

end

endcase

end

// Create process for Moore output logic for state\_out (outputs function of // current\_state only)

always @(current\_state) begin

// Initialize state\_out to default values so case only covers when they // change

state\_out = 3'd0;

case (current\_state)

idle : ;

lsb : state\_out = 3'd1;

mid : state\_out = 3'd2;

msb : state\_out = 3'd3;

calc\_done : state\_out = 3'd4;

err : state\_out = 3'd5;

endcase

end

endmodule // End module

**Seven-segment Control:**

// Christopher Hays

// ECE 176

// Seven segment display controller, 3-bit input

// seven\_segment\_cntrl.v

// Declare module

module seven\_segment\_cntrl(in, seg\_a, seg\_b, seg\_c, seg\_d, seg\_e, seg\_f, seg\_g);

// Declare input and outputs

input [2:0] in;

output seg\_a, seg\_b, seg\_c, seg\_d, seg\_e, seg\_f, seg\_g;

// Create a register to temporarily hold the output

reg [6:0] out;

// Update the output on any change in input

always @(\*)begin

case (in)

0 : out = 7'b1111110;

1 : out = 7'b0110000;

2 : out = 7'b1101101;

3 : out = 7'b1111001;

4 : out = 7'b0110011;

5 : out = 7'b1011011;

6 : out = 7'b1011111;

7 : out = 7'b1110000;

default : out = 7'b1001111;

endcase

end

// Split the temporary register to the appropriate data lines

assign {seg\_a, seg\_b, seg\_c, seg\_d, seg\_e, seg\_f, seg\_g} = out;

endmodule // End module

**8x8 Multiplier (main module):**

// Christopher Hays

// ECE 176

// Lab 10

// 8x8 Multiplier Instantiations

// lab10.v

// Declare the module, inputs, and outputs

module mult8x8 (input [7:0] dataa,

input [7:0] datab,

input start,

input reset\_a,

input clk,

output reg done\_flag,

output reg [15:0] product8x8\_out,

output reg seg\_a, seg\_b, seg\_c, seg\_d, seg\_e, seg\_f, seg\_g

);

// Declare wires to connect the instantiated modules

// Wires have the same name as the port they represent

wire clk\_ena, sclr\_n, done;

wire a, b, c, d, e, f, g;

wire [1:0] sel, shift, count;

wire [2:0] state\_out;

wire [3:0] aout, bout;

wire [7:0] product;

wire [15:0] shift\_out, sum, product8x8;

// Mux A switches between the low and high bits of dataa and outputs

// to the multiplier, contolled by the low mux\_sel bit

mux4 m1 (.mux\_in\_a(dataa[3:0]),

.mux\_in\_b(dataa[7:4]),

.mux\_sel(sel[0]),

.mux\_out(aout[3:0])

);

// Mux B switches between the low and high bits of datab and outputs

// to the multiplier, controlled by the high mux\_sel bit

mux4 m2 (.mux\_in\_a(datab[3:0]),

.mux\_in\_b(datab[7:4]),

.mux\_sel(sel[1]),

.mux\_out(bout[3:0])

);

// 4x4 multiplier sends the product of the two 4-bit mux outputs to the shifter

mult4x4 mult1 (.dataa(aout[3:0]),

.datab(bout[3:0]),

.product(product[7:0])

);

// 2-bit counter synchronized with the clock that counts the clock cycles

// and outputs to the control unit. The asynchronous clear is enabled

// when the pulse on the start line is done

counter c1 (.clk(clk),

.aclr\_n(~start),

.count\_out(count[1:0])

);

// Shifter takes the product of the 4x4 multiplier and shifts it the left by 0,

// 4, or 8 bits, multiplying by 2^0, 2^4, or 2^8. Output is sent to the adder

shifter s1 (.inp(product[7:0]),

.shift\_cntrl(shift[1:0]),

.shift\_out(shift\_out[15:0])

);

// Adder adds the output of the shifter with the current contents of the register

// and sends the result back to the same register

adder a1 (.dataa(shift\_out[15:0]),

.datab(product8x8[15:0]),

.sum(sum[15:0])

);

// Register acts as an accumulator, storing the intermediate and final results of

// the multiplication. Synchronous clear and clock enable inputs are

// controlled by the control unit state machine.

reg16 r1 (.clk(clk),

.sclr\_n(sclr\_n),

.clk\_ena(clk\_ena),

.datain(sum[15:0]),

.reg\_out(product8x8[15:0])

);

// Multiplier control unit. Start, reset, and clock inputs are supplied by the

// parent module. Input\_sel controls the multiplexers, shift\_sel controls

// the shifter, and state\_out is sent to the seven segment display controller.

// The done output connects to the parent module output while the clk\_ena

// and sclr\_n control the register.

mult\_control u6 (.clk(clk),

.reset\_a(reset\_a),

.start(start),

.count(count[1:0]),

.input\_sel(sel[1:0]),

.shift\_sel(shift[1:0]),

.state\_out(state\_out[2:0]),

.done(done),

.clk\_ena(clk\_ena),

.sclr\_n(sclr\_n)

);

// Seven segment controller takes the state output from the controller and drives

// the LEDs in order to display it.

seven\_segment\_cntrl ss1 (.in(state\_out[2:0]),

.seg\_a(a),

.seg\_b(b),

.seg\_c(c),

.seg\_d(d),

.seg\_e(e),

.seg\_f(f),

.seg\_g(g)

);

// Synchronously update the main output registers. These outputs can only be

// connected to a module via a wire variable

always @(posedge clk)begin

done\_flag <= done;

product8x8\_out <= product8x8;

seg\_a <= a;

seg\_b <= b;

seg\_c <= c;

seg\_d <= d;

seg\_e <= e;

seg\_f <= f;

seg\_g <= g;

end

endmodule // End module

**8x8 Multiplier Testbench:**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Module Name: mult8x8\_tb

// File Name: mult8x8\_tb.v

// Module Function: Test bench for ECE 176 Assignment 10

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

`timescale 1 ns/1 ns

module mult8x8\_tb();

// Wires to connect to DUT

reg clk, reset\_a, start;

reg [7:0] dataa, datab;

wire [15:0] product8x8\_out;

wire seg\_a, seg\_b, seg\_c, seg\_d, seg\_e, seg\_f, seg\_g;

// Instantiate unit under test (mult\_control)

mult8x8 mult8x8\_1 (.clk(clk),

.reset\_a(reset\_a),

.dataa(dataa),

.datab(datab),

.product8x8\_out(product8x8\_out),

.done\_flag(done\_flag),

.start(start),

.seg\_a(seg\_a),

.seg\_b(seg\_b),

.seg\_c(seg\_c),

.seg\_d(seg\_d),

.seg\_e(seg\_e),

.seg\_f(seg\_f),

.seg\_g(seg\_g)

);

// Process to create clock signal

initial begin

clk = 0;

forever clk = #25 ~clk;

end

// Set the reset control

initial begin

reset\_a = 1'b1;

#50 reset\_a = 1'b0;

end

// Set input values to control start signal

initial begin

start = 1'b1;

#50 ;

forever begin

start = 1'b1;

#50 start = 1'b0;

@(negedge done\_flag) ;

#25 ;

end

end

// Process to control data inputs

initial begin

dataa = 8'hFF;

datab = 8'hFF;

#50 ;

forever begin

@(negedge done\_flag)

#25 dataa = dataa + 24;

datab = datab + 51;

end

end

initial begin

#1200 $stop;

end

endmodule