Brodric Young
ECEN 340
Lab #9

Timing Evaluation and Improvement

Purpose:

1. To learn how to use Vivado's timing analysis tools
2. To learn how to make use of Vivado's "Device" view for timing analysis.
3. To learn how to predict the maximum operating frequency of the design.
4. To become familiar with the Artix 7 structure.
5. To learn how to make timing improvements.

Overview:

The first objective of this lab is to evaluate the timing of Lab 8's memory design to determine it's maximum operating frequency for:

A) An 8-bit adder using the "+" operator
B) An 8-bit multiplier using the "*" operator
C) An 8-bit multiplier using the instantiation of Lab 5, the continuous assignment (data-flow) multiplier.

In other words, for part B, you will replace the "+" operator in lab 8 with an instantiation of your Lab 5 multiplier in order to evaluate its maximum speed. For part C, you must write a new and improved multiplier module.

The final objective of this lab is to speed up the clock rate of the multiplier by pipelining the design for part C.

Part 1 (First lab day).

1. Make sure the "create_clock" line is uncommented in the constraints file. This line will set up the timing constraints for the analysis. The period for the analysis will be 10ns.
2. In Vivado, execute the Lab 8 memory project all the way to the "Implementation" stage, and open the implemented design.
3. Under the "Implementation" menu, select "Edit Timing Constraints" to make sure the timing will be evaluated at a 10ns period.

4. Under the same menu, select “Report Timing Summary”. Use the default settings and generate the summary.
5. Record the “Worst Negative Slack” number. A positive number will indicate the design meets timing requirements.
6. Click on the WNS number and you will be able to view the timing of each path. The default view shows the slowest path first.
7. Record the path starting point and the path stoping point for the 4 worst paths.
8. Activiate the “Schematic” view and observe what happens as you select different paths in the timing report.
9. Activate the “Device” view and make the same observation (you will need to zoom way in!).
10. Repeat this process for the “*” operation instead of the “+” operation.
11. Repeat the process again for the Lab 5 Data-flow Multiplier (instantiate the Lab 5 multiplier in the memory lab).

Part 2 (Second lab day).

Implement a pipelined multiplier and evaluate it’s speed by instatiating it into your lab 8 sequential memory.

Since pipelining takes multiple clock cylcles to complete a multiplication operation, the result won’t be ready until a few clock cycles after the input memory is accessed.

What must you do in order to force the product of the input memory data to be written to the coresponding address in output memory?

Pipelining Hint: If product = p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 (see lab 5), it is possible to pipeline the operation in using the technique in Figure 1.

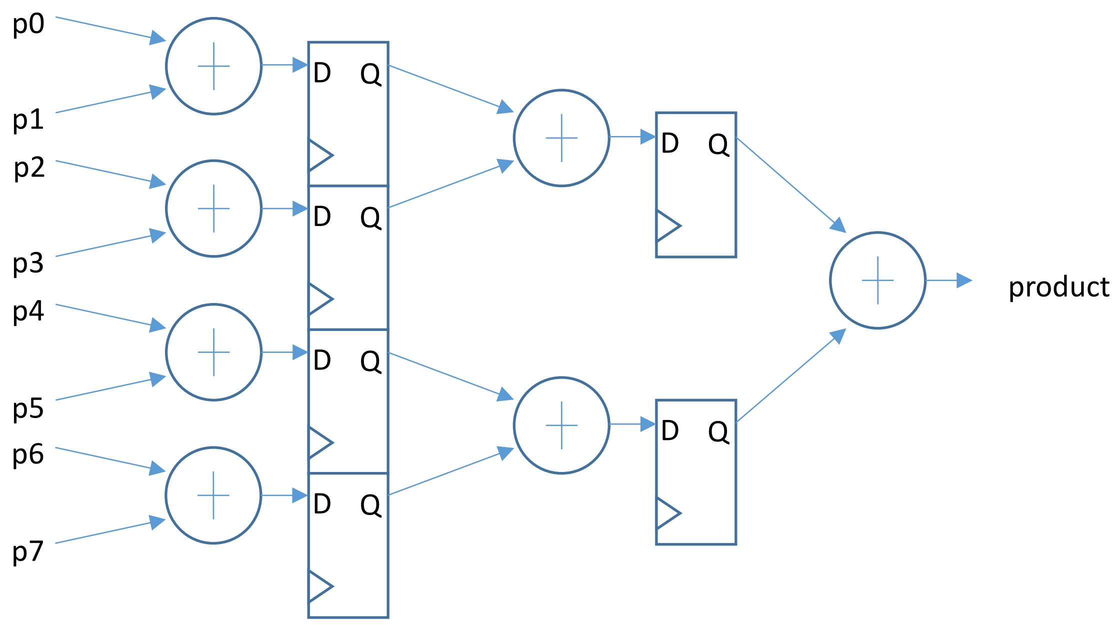


Figure 1: Pipelining the Partial-Product Sums

Timing results:

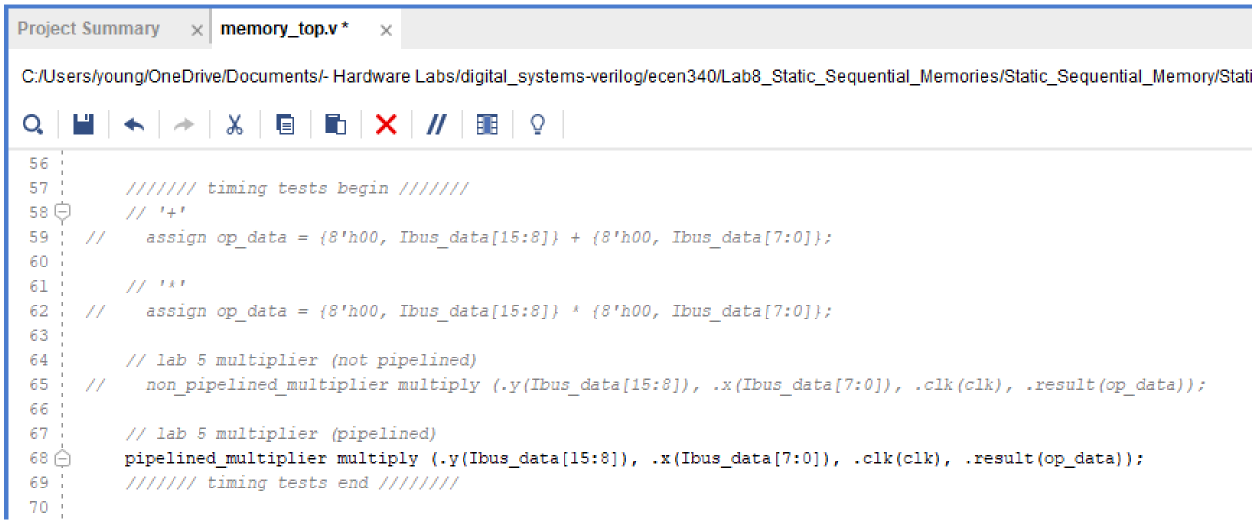
	WNS (ns)	Path Starting Point	Path Stopping Point	f_{max} (MHz)
Lab 8 "+"	5.071	IM/data_out_reg/i_/C	OM/mem_reg_0_15_5_5/SP/I	202
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_8_8/SP/I	
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_4_4/SP/I	
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_7_7/SP/I	
Lab 8 "*"	0.728	IM/data_out_reg/i_/C	OM/mem_reg_0_15_13_13/SP/I	106
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_15_15/SP/I	
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_11_11/SP/I	
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_12_12/SP/I	
Lab 5 "Non-pipelined"	1.025	IM/data_out_reg/i_/C	OM/mem_reg_0_15_14_14/SP/I	111
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_13_13/SP/I	
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_10_10/SP/I	
		IM/data_out_reg/i_/C	OM/mem_reg_0_15_15_15/SP/I	
Lab 5 "pipelined"	5.829	IM/data_out_reg/i_/C	multiplier/sum3_reg[12]/D	239
		IM/data_out_reg/i_/C	multiplier/sum3_reg[9]/D	
		IM/data_out_reg/i_/C	multiplier/sum3_reg[11]/D	
		IM/data_out_reg/i_/C	multiplier/sum3_reg[13]/D	

Table 1: Slack times and paths for differing operations

Maximum frequency predicted using this formula:

$$f_{max} = \frac{1}{T_{clk} - T_{slack}} = \frac{1}{10ns - T_{WNS}}$$

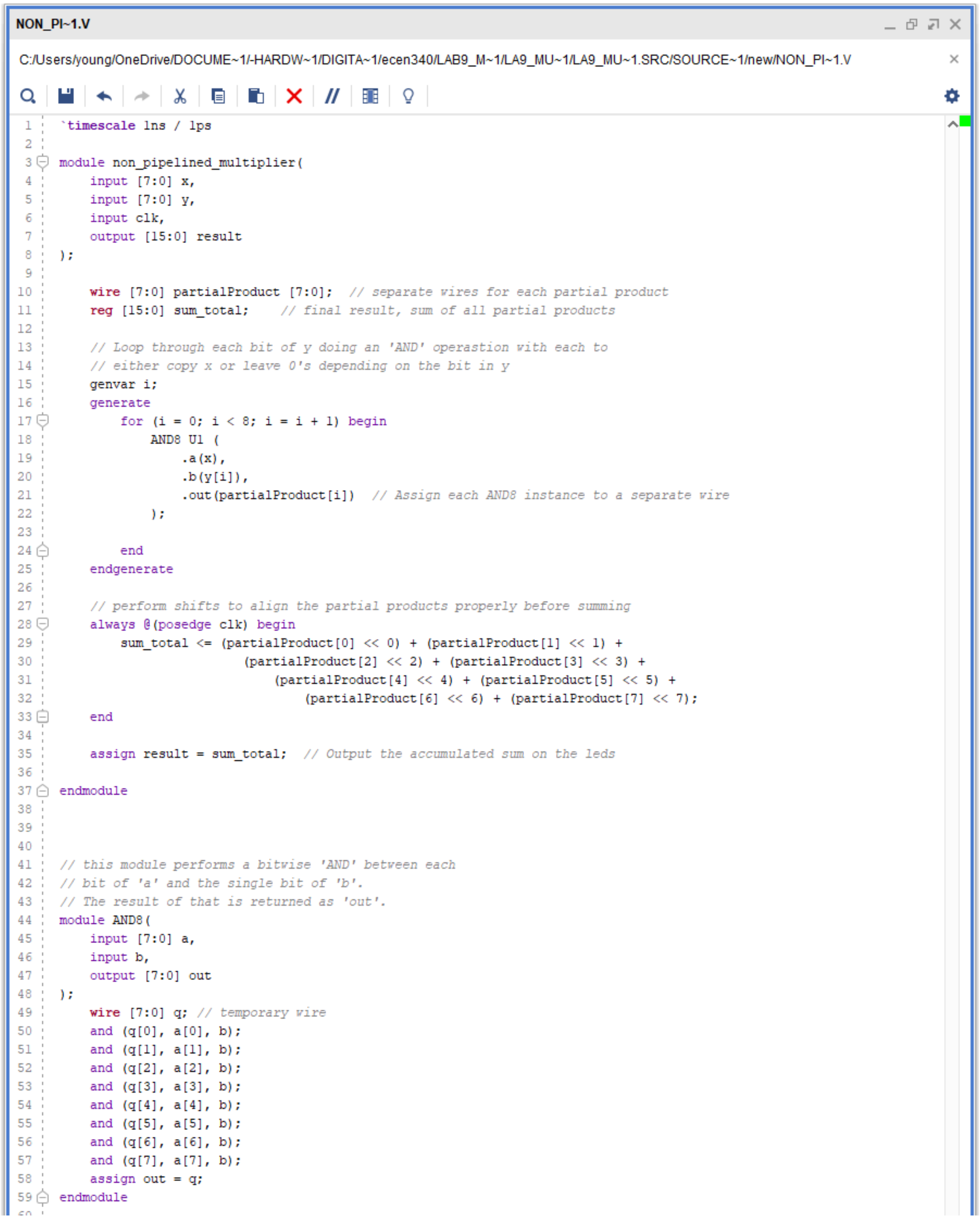
Code:


Project Summary | memory_top.v *

C:/Users/young/OneDrive/Documents/- Hardware Labs/digital_systems-verilog/ecen340/Lab8_Static_Sequential_Memories/Static_Sequential_Memory/Stati

```

    /////// timing tests begin ///////
    // '+'
//    assign op_data = {8'h00, Ibus_data[15:8]} + {8'h00, Ibus_data[7:0]};

    // '*'
//    assign op_data = {8'h00, Ibus_data[15:8]} * {8'h00, Ibus_data[7:0]};

    // lab 5 multiplier (not pipelined)
//    non_pipelined_multiplier multiply (.y(Ibus_data[15:8]), .x(Ibus_data[7:0]), .clk(clk), .result(op_data));

    // lab 5 multiplier (pipelined)
    pipelined_multiplier multiply (.y(Ibus_data[15:8]), .x(Ibus_data[7:0]), .clk(clk), .result(op_data));
    /////// timing tests end ////////

```

Code 1: Top module changes for timing tests

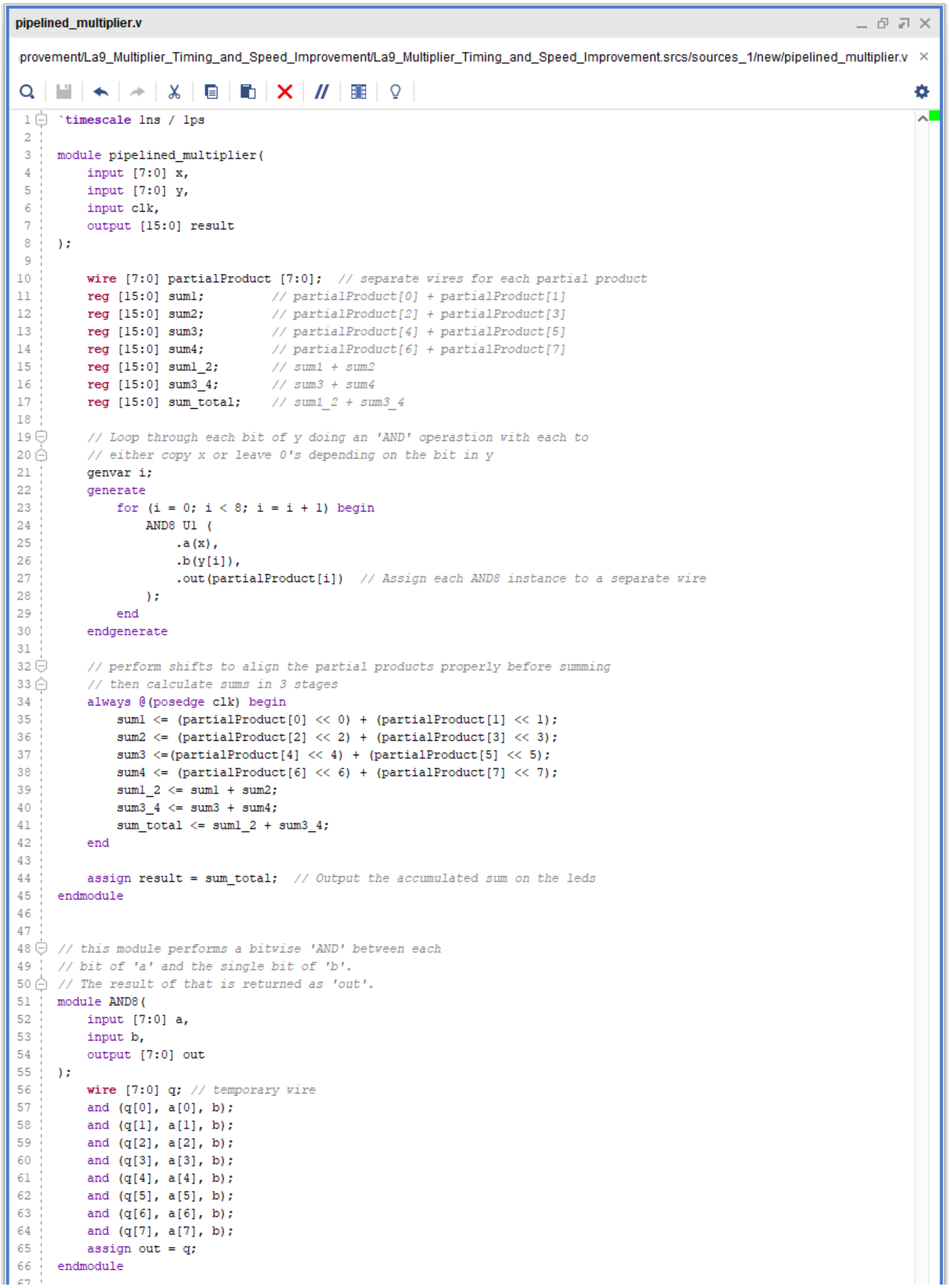


NON_PI~1.V

C:/Users/young/OneDrive/DOCUME~1/-HARDW~1/DIGITA~1/ecen340/LAB9_M~1/LA9_MU~1/LA9_MU~1.SRC/SOURCE~1/new/NON_PI~1.V

```verilog
`timescale 1ns / 1ps

module non_pipelined_multiplier(
    input [7:0] x,
    input [7:0] y,
    input clk,
    output [15:0] result
);

    wire [7:0] partialProduct [7:0];  // separate wires for each partial product
    reg [15:0] sum_total;    // final result, sum of all partial products

    // Loop through each bit of y doing an 'AND' operastion with each to
    // either copy x or leave 0's depending on the bit in y
    genvar i;
    generate
        for (i = 0; i < 8; i = i + 1) begin
            AND8 U1 (
                .a(x),
                .b(y[i]),
                .out(partialProduct[i])  // Assign each AND8 instance to a separate wire
            );

        end
    endgenerate

    // perform shifts to align the partial products properly before summing
    always @(posedge clk) begin
        sum_total <= (partialProduct[0] << 0) + (partialProduct[1] << 1) +
                     (partialProduct[2] << 2) + (partialProduct[3] << 3) +
                         (partialProduct[4] << 4) + (partialProduct[5] << 5) +
                             (partialProduct[6] << 6) + (partialProduct[7] << 7);
    end

    assign result = sum_total;  // Output the accumulated sum on the leds

endmodule



// this module performs a bitwise 'AND' between each
// bit of 'a' and the single bit of 'b'.
// The result of that is returned as 'out'.
module AND8(
    input [7:0] a,
    input b,
    output [7:0] out
);
    wire [7:0] q; // temporary wire
    and (q[0], a[0], b);
    and (q[1], a[1], b);
    and (q[2], a[2], b);
    and (q[3], a[3], b);
    and (q[4], a[4], b);
    and (q[5], a[5], b);
    and (q[6], a[6], b);
    and (q[7], a[7], b);
    assign out = q;
endmodule
```

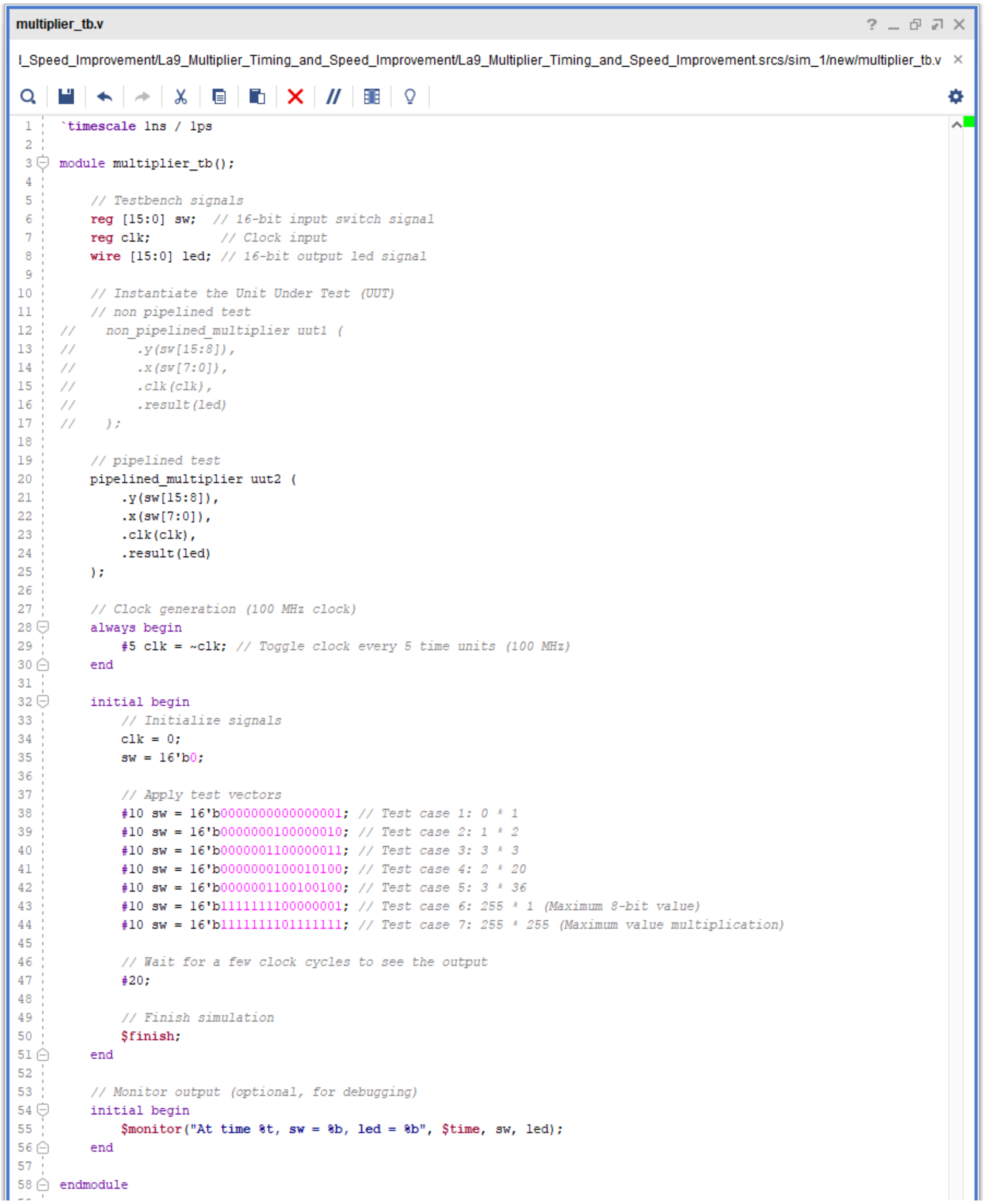
Code 2: Non-pipelined multiplier module


pipelined_multiplier.v

provement/La9_Multiplier_Timing_and_Speed_Improvement/La9_Multiplier_Timing_and_Speed_Improvement.srcs/sources_1/new/pipelined_multiplier.v

```verilog
`timescale 1ns / 1ps

module pipelined_multiplier(
    input [7:0] x,
    input [7:0] y,
    input clk,
    output [15:0] result
);

    wire [7:0] partialProduct [7:0];  // separate wires for each partial product
    reg [15:0] sum1;          // partialProduct[0] + partialProduct[1]
    reg [15:0] sum2;          // partialProduct[2] + partialProduct[3]
    reg [15:0] sum3;          // partialProduct[4] + partialProduct[5]
    reg [15:0] sum4;          // partialProduct[6] + partialProduct[7]
    reg [15:0] sum1_2;        // sum1 + sum2
    reg [15:0] sum3_4;        // sum3 + sum4
    reg [15:0] sum_total;     // sum1_2 + sum3_4

    // Loop through each bit of y doing an 'AND' operastion with each to
    // either copy x or leave 0's depending on the bit in y
    genvar i;
    generate
        for (i = 0; i < 8; i = i + 1) begin
            AND8 U1 (
                .a(x),
                .b(y[i]),
                .out(partialProduct[i])  // Assign each AND8 instance to a separate wire
            );
        end
    endgenerate

    // perform shifts to align the partial products properly before summing
    // then calculate sums in 3 stages
    always @(posedge clk) begin
        sum1 <= (partialProduct[0] << 0) + (partialProduct[1] << 1);
        sum2 <= (partialProduct[2] << 2) + (partialProduct[3] << 3);
        sum3 <=(partialProduct[4] << 4) + (partialProduct[5] << 5);
        sum4 <= (partialProduct[6] << 6) + (partialProduct[7] << 7);
        sum1_2 <= sum1 + sum2;
        sum3_4 <= sum3 + sum4;
        sum_total <= sum1_2 + sum3_4;
    end

    assign result = sum_total;  // Output the accumulated sum on the leds
endmodule


// this module performs a bitwise 'AND' between each
// bit of 'a' and the single bit of 'b'.
// The result of that is returned as 'out'.
module AND8(
    input [7:0] a,
    input b,
    output [7:0] out
);
    wire [7:0] q; // temporary wire
    and (q[0], a[0], b);
    and (q[1], a[1], b);
    and (q[2], a[2], b);
    and (q[3], a[3], b);
    and (q[4], a[4], b);
    and (q[5], a[5], b);
    and (q[6], a[6], b);
    and (q[7], a[7], b);
    assign out = q;
endmodule
```

Code 3: Pipelined multiplier


multiplier_tb.v
l_Speed_Improvement/La9_Multiplier_Timing_and_Speed_Improvement/La9_Multiplier_Timing_and_Speed_Improvement.srcs/sim_1/new/multiplier_tb.v

```verilog
`timescale 1ns / 1ps

module multiplier_tb();

    // Testbench signals
    reg [15:0] sw;  // 16-bit input switch signal
    reg clk;         // Clock input
    wire [15:0] led; // 16-bit output led signal

    // Instantiate the Unit Under Test (UUT)
    // non pipelined test
//    non_pipelined_multiplier uut1 (
//        .y(sw[15:8]),
//        .x(sw[7:0]),
//        .clk(clk),
//        .result(led)
//    );

    // pipelined test
    pipelined_multiplier uut2 (
        .y(sw[15:8]),
        .x(sw[7:0]),
        .clk(clk),
        .result(led)
    );

    // Clock generation (100 MHz clock)
    always begin
        #5 clk = ~clk; // Toggle clock every 5 time units (100 MHz)
    end

    initial begin
        // Initialize signals
        clk = 0;
        sw = 16'b0;

        // Apply test vectors
        #10 sw = 16'b0000000000000001; // Test case 1: 0 * 1
        #10 sw = 16'b0000000100000010; // Test case 2: 1 * 2
        #10 sw = 16'b0000001100000011; // Test case 3: 3 * 3
        #10 sw = 16'b0000000100010100; // Test case 4: 2 * 20
        #10 sw = 16'b0000001100100100; // Test case 5: 3 * 36
        #10 sw = 16'b1111111100000001; // Test case 6: 255 * 1 (Maximum 8-bit value)
        #10 sw = 16'b1111111101111111; // Test case 7: 255 * 255 (Maximum value multiplication)

        // Wait for a few clock cycles to see the output
        #20;

        // Finish simulation
        $finish;
    end

    // Monitor output (optional, for debugging)
    initial begin
        $monitor("At time %t, sw = %b, led = %b", $time, sw, led);
    end

endmodule
```

Code 4: Multiplier (both pipelined and non-pipelined) test bench

Pipelined simulation and synthesis:

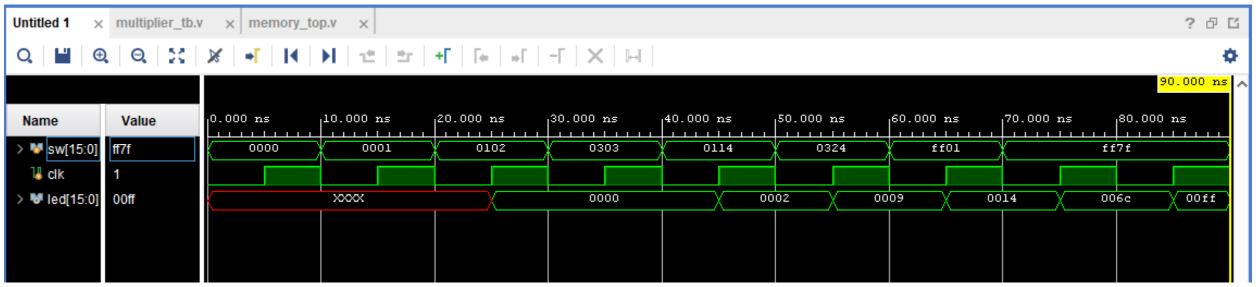


Figure 2: Pipelined multiplier simulation

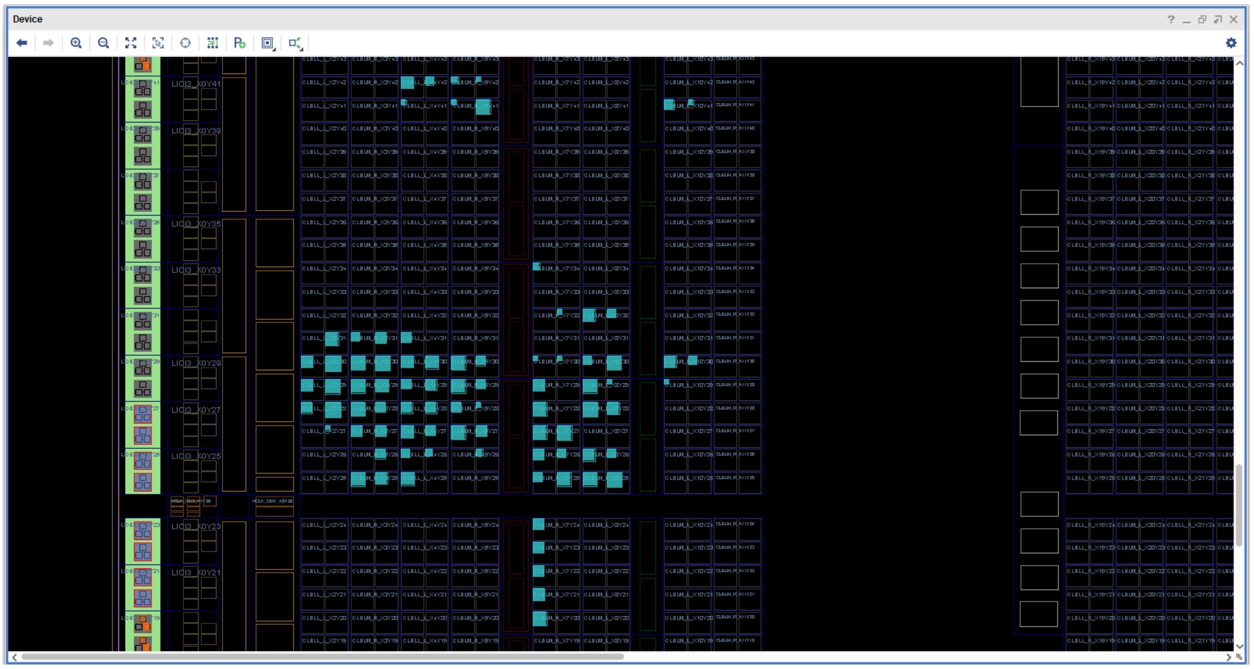


Figure 3: Pipelined multiplier device synthesis

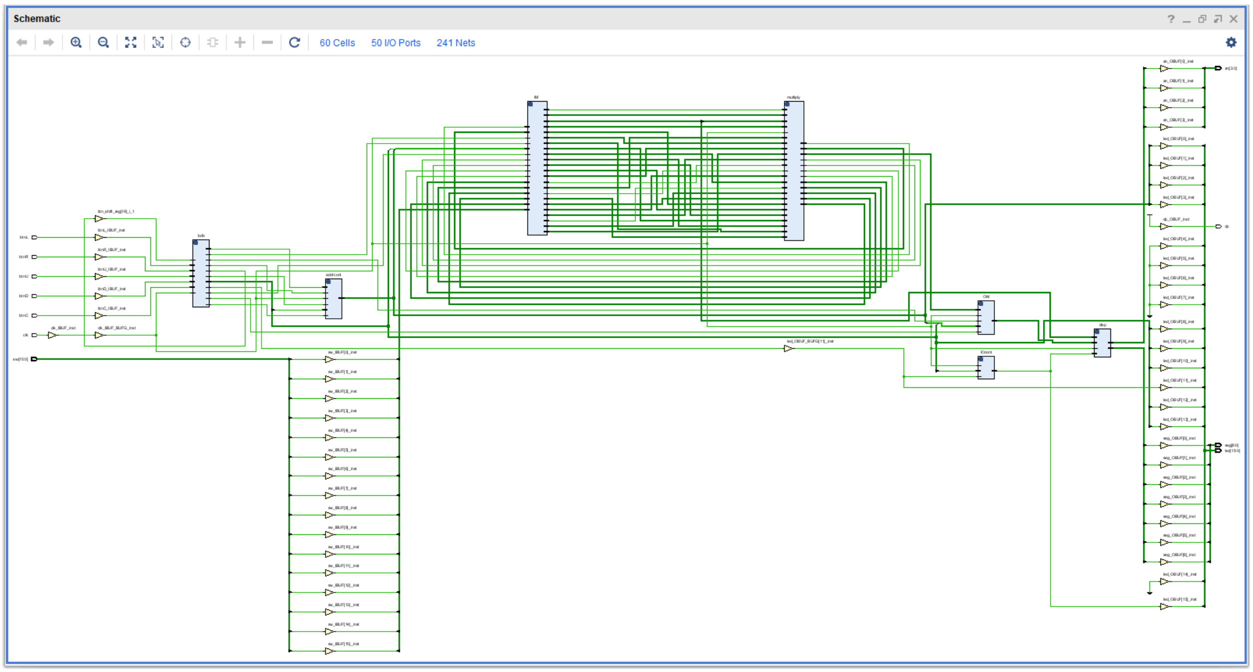


Figure 4: Pipelined multiplier schematic

Conclusion:

In this lab, we used a new tool in Vivado called the timing summary which allowed us to view how much time our implementation takes and what slack we have. Using this new tool we saw how a new process we learned about called pipelining changed the speed at which our implementation could run. Our maximum frequency should be higher when our slack time is greater and that's what helped me predict the maximum frequencies which can be seen in "Table 1: Slack times and paths for differing operations" along with the other timing information. I learned some advantages of pipelining are that through it you can have an increased throughput if you have lots of data to pass through and also that you can have a higher clock speed and less idle time waiting for things to be processed. Some disadvantages are that if you only have a few pieces of data it doesn't help much because it takes a couple more clock cycles to get them through and also its more complex. Pipelining is mainly helpful when you have a lot of things needing to go through since after you get past the first couple things you get the rest done at every single clock cycle.