

**Introduction**

In this lab, we will be learning about designing functions which consist of both combinational and sequential building blocks. The combinational part will consist of creating a 4 bit multiplier which will use 4 bit CLA adders in series to make an 8 bit added which will add the partial products from the multiplier. In a scenario where many items need to be multiplied, a normal 4 bit multiplier, which must wait for the multiplication process to finish before it starts a new one, can be slow. This is why we can use pipelining to make the process much more efficient. Pipelining will consist of placing registers in strategic locations within the multiplication process to separate one part of the process from the other.

By implementing a pipelining process, we will be able to start the process of multiplying two numbers before the first number finishes multiplying. Once the first set of numbers reaches the register, the value will be saved into the register to go into the second half, and then a new set of numbers can be placed into the first half of the process. To test that this works, we will be generating our own clock signal on the FPGA board to manually control the clock cycles to make sure that it works as it should.

**Design Methodology**

To start with, we first designed a 4 bit combinational unsigned integer multiplier. From our knowledge of multiplication, we know that by multiplying two numbers, we will receive a number of partial products dependent on the size of the numbers being multiplied (in this case it is 4 bits wide). The partial products must then be added together to receive the final product. Since we are multiplying two 4 bit numbers, the partial products will be up to 8 bits in length.

To add the 8 bit partial products, we must first make an 8 bit adder. To do so, we can use two 4 bit adders in series from the previous lab by connecting the carryout of the first one to the carry in of the second. Once this task is accomplished, we can add the numbers according to the following formulas.

Using A[3:0] and B[3:0]

AxB:

pp0 = 0, 0, 0, 0, (a3, a2, a1, a0)\*b0

pp1 = 0, 0, 0, (a3, a2, a1, a0)\*b1, 0

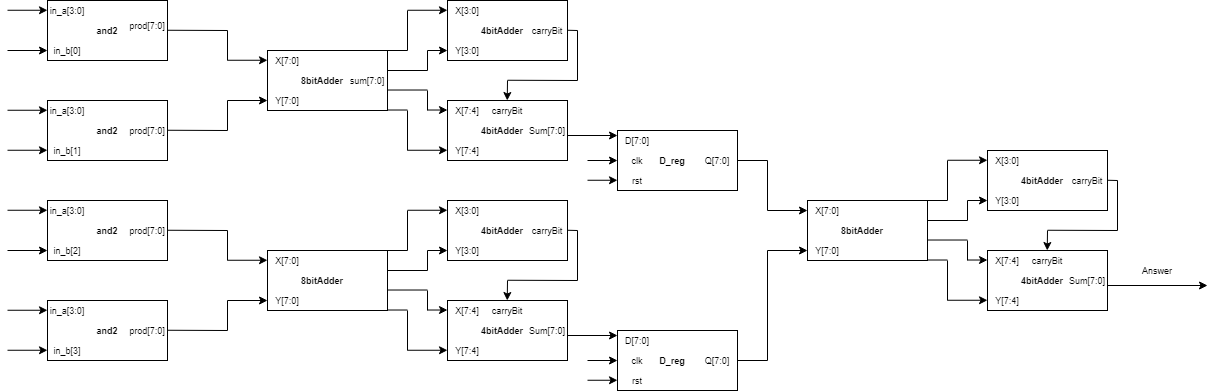
pp2 = 0, 0, (a3, a2, a1, a0)\*b1, 0, 0

pp3 = 0, (a3, a2, a1, a0)\*b1, 0, 0, 0

AxB = pp0 + pp1 + pp2 + pp3

Once the combinatorial method of multiplying two 4 bit numbers together was developed using the above formulas, we simply needed to add a register at an intermediate step to separate the two parts of the processes evenly. We placed the register in between the 8 bit adding steps because the addition takes the most time in the process, as compared to all of the other items. Once completed, we had developed the block diagram of the pipelined multiplier on figure 1 and modules listed in table 2.

Block Diagram

**

*Figure 1: Block diagram of pipeline multiplier*

D-register truth table

|  |  |  |  |
| --- | --- | --- | --- |
| d | clk | rst | Q |
| 0 | ↱ | 0 | 0 |
| 1 | ↱ | 0 | 1 |
| x | ↱ | 1 | 0 |

*Table 1, D-reg truth table*

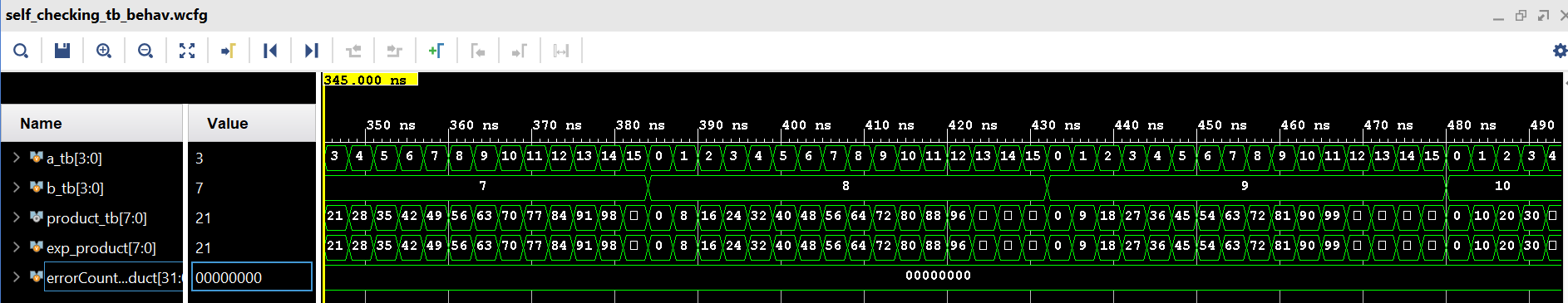
Table 1: Module Table

|  |  |
| --- | --- |
| **Module** | **Function** |
| 8bitCLA.v | Calculates 8 bit addition |
| 4bitCLA.v | Used two 4 bit addition to make 8 bit addition |
| addhalf.v | Calculates sum and carry over |
| carrybits.v | Calculates the carry bits in parallel |
| fourBitMult.v | Calculates 4 bit by 4 bit multiplication |
| pipeMult.v | Calculates 4 bit by 4 bit multiplication using pipeline method |
| button\_debouncer.v | Implements button press for clock cycle |
| dReg.v | D-register for saving values |
| clk\_gen.v | Generates a clock signal to synchronize with |
| 4bitmulti\_FPGA.v | Implements 4 bit multiplication on the FPGA board |
| bcd\_to\_7seg.v | Convert bcd for the 7 seg display |
| led\_mux.v | Chooses a 7seg display |

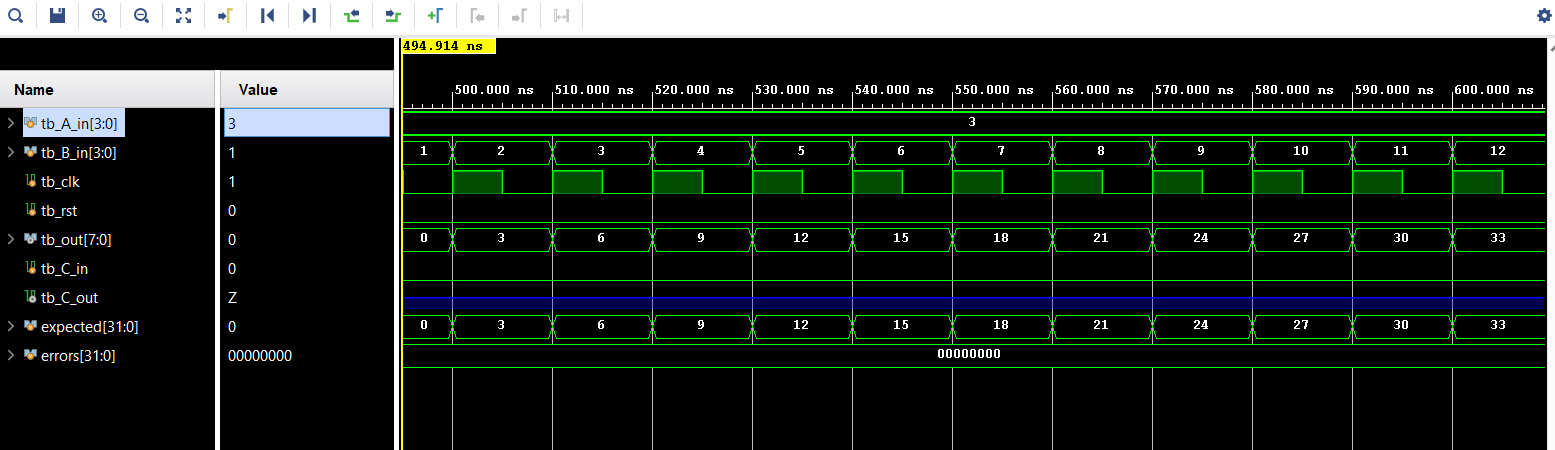
*Table 2, Module descriptions*

**Simulation Results**

The simulation below shows the 4 bit by 4 bit simple multiplication, with input A and input B. input A is constantly incrementing by 1, while input B increments when A goes from F to 0. The product is shown on the product\_tb and exp\_product shows what the expected product is. At the bottom, errorCount will increment by 1 if product\_tb and exp\_product does not equal to each other.



*Figure 3: Simple 4 bit by 4 bit multiplication*

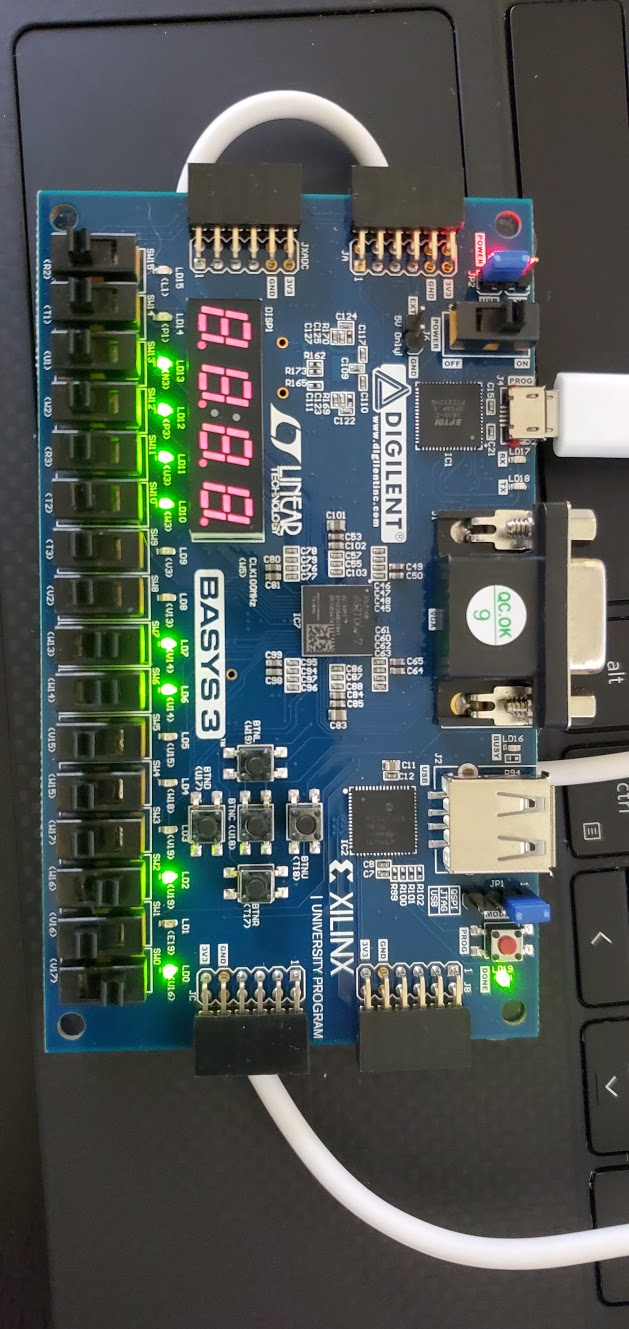
This simulation also follows the same incrementing rule for input A and B. However, now that we implemented a pipeline, the product will be delayed as shown on the waveform. We also have an error counter on the bottom to show if the product doesn’t equal the expected product.

*Figure 4: 4 bit by 4 bit multiplication using pipeline*

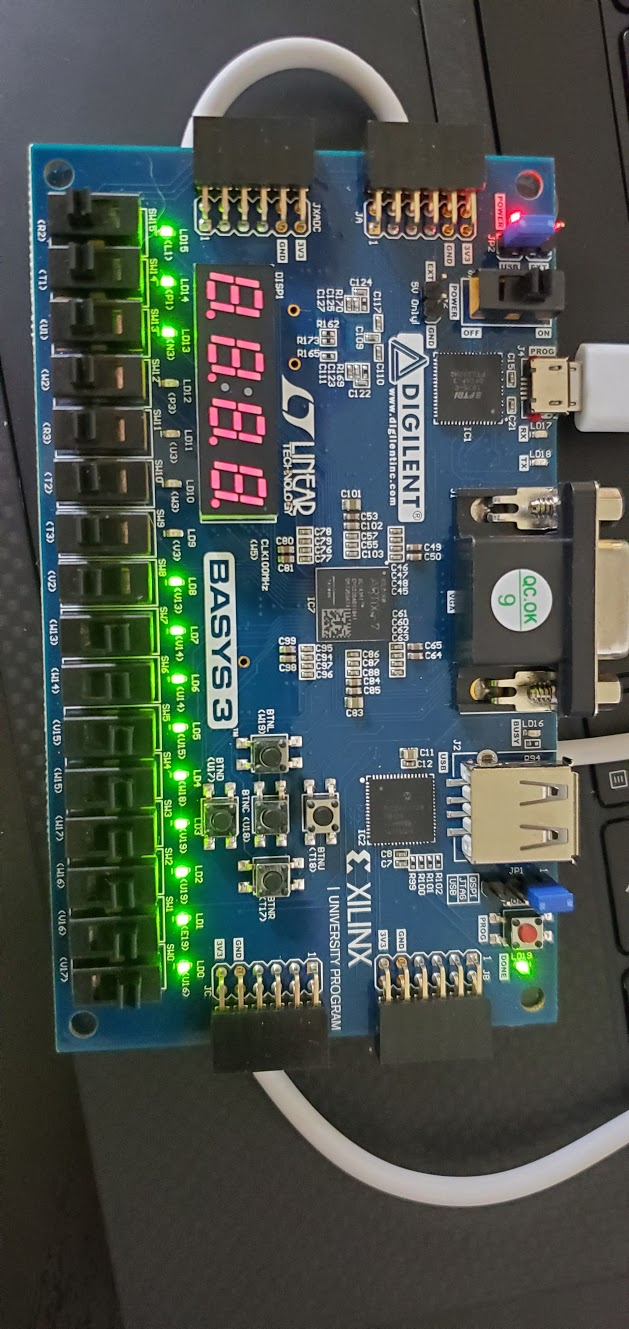
**FPGA Validation**

For our FPGA board we implemented the multiplication so that right 4 switches represent input A, while the next 4 switches represent input B. We made it so that when the switches are on the LED at the same location turns on for clarity. Then we made the output display on the left 8 LED with the LED on representing 1 and off representing 0. There will be few examples shown below in figures 5 and 6.

Part 1



*Figure 5: Simple multiplication [1100]\*[0101]=[00111100]*



*Figure 6: Simple multiplication [1111]\*[1111]=[1110001]*

Part 2

For the second part of the lab, we implemented a pipeline multiplier and utilized the 7 segment display. The 4 rightmost switches are represent the 4 bit number A and the 4 leftmost switches represent the 4 bit number B. As seen in figure 7, we are multiplying 6 and 5 which returns 30. In figure 7, the clock has not been pressed. Because we are utilizing registers, the output will not be passed through until a rising edge clock signal gets passed through the dreg.

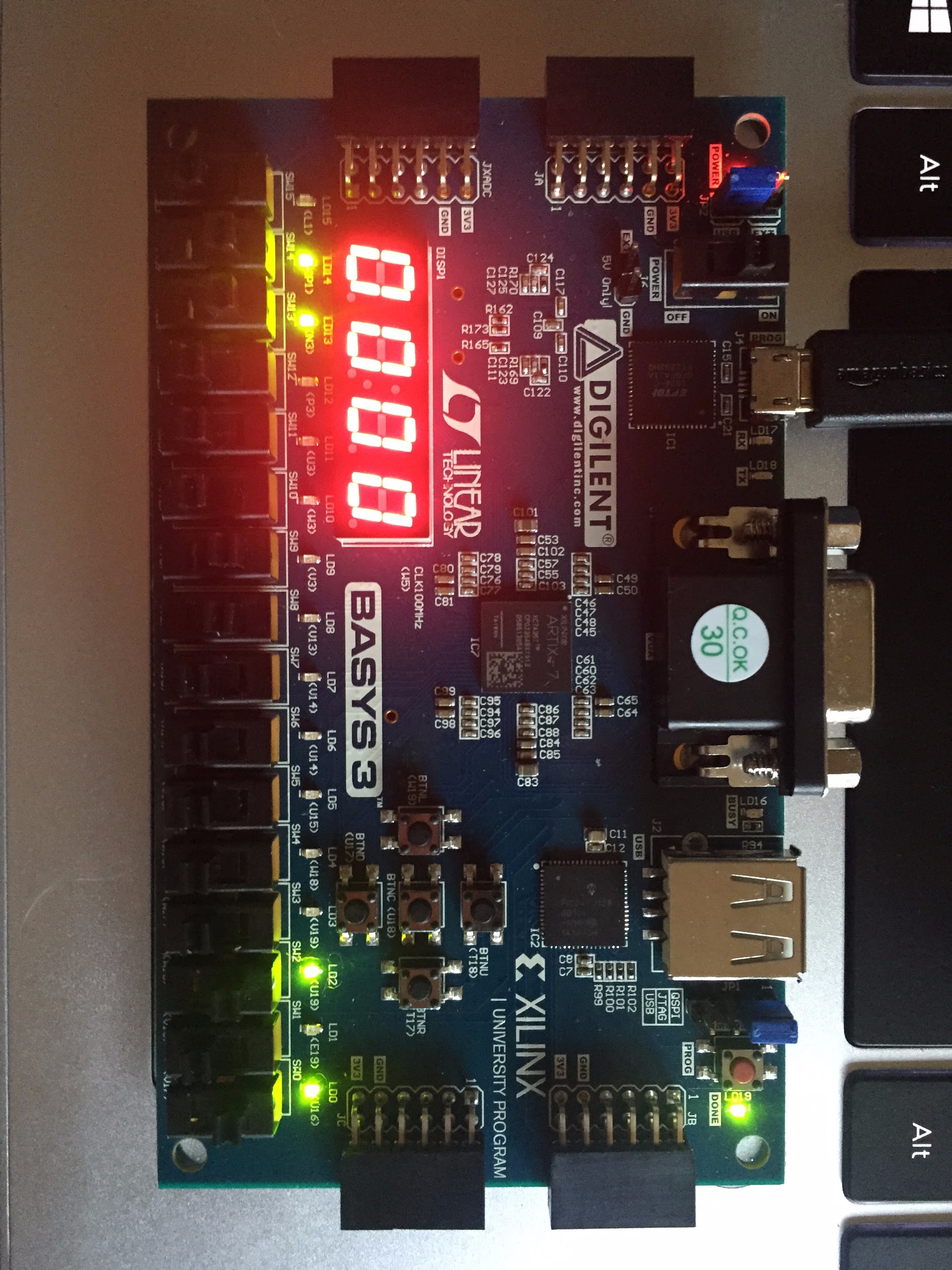


Figure 7: Multiplication of 6 x 5

In figure 8 below, the button has been pressed so the output changes to 30. When the switches are changed to 0000, 0000, the output stays at 30 because the clock button has not been pressed, as seen in figure 9.



Figure 8: Debouncer button has been pressed

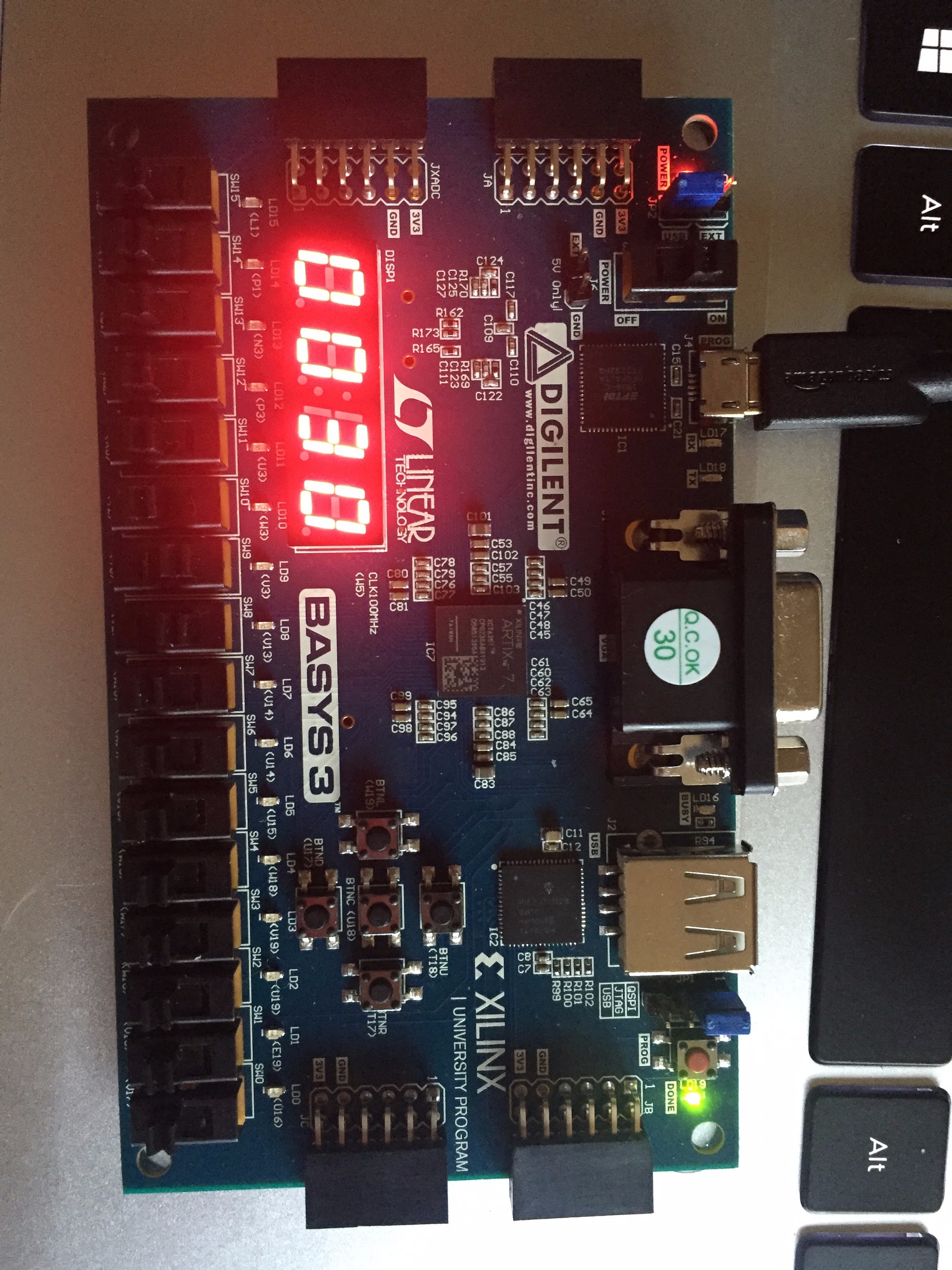


Figure 9: Switches are changed, but the output does not change because the button has not been pressed.

**Conclusion**

The purpose of this lab was to design functions which consist of both combinational and sequential building blocks. First we implemented a simple multiplication, where we multiplied 4-bit input (input A) to the first bit of the other input (input B). Then we repeat this process but next we multiply input A with the 2nd bit of input B, and so on. Later we add the 4 products all together using an 8 bit adder, made from two 4bit adders made in the previous lab. However, we can make this more efficient by implementing a pipeline. A combinational must wait until an output is received before it can start on the next set of numbers. Pipelining will consist of placing dregisters in strategic locations within the multiplication process to separate one part of the process from the other.

Overall, we were able to complete both parts of the lab. Although it was not required, we also implemented the combinational circuit onto the FPGA board. After this, we added our debounced button and dregisters to store values in intermediate steps. We also implemented the 7seg display to make it easier to see our outputs.

**Appendix**

|  |
| --- |
| 8bitCLA.v |
| `timescale 1ns / 1ps  module eightBitCLA( //created using two 4 bit adders  input [7:0] a,  input [7:0] b,  input cin,  output cout,  output [7:0] eightSum  );  wire temp;    fourBitCLA fb0(.A(a[3:0]), //adding the first 4 bits  .B(b[3:0]),  .C\_in(cin),  .C\_out(temp),  .Sum(eightSum[3:0]));      fourBitCLA fb1(.A(a[7:4]),.B(b[7:4]), //adding second four bits. Take carry out of the previous 4bit adder  .C\_in(temp), .C\_out(cout),.Sum(eightSum[7:4]));    endmodule |

|  |
| --- |
| 4bitCLA.v |
| module fourBitCLA(  input [3:0]A,  input [3:0]B,  input C\_in,  output C\_out,  output [3:0] Sum  );  wire [3:0] P;  wire [3:0] G;  wire [4:0] C;  carryBits cBits(.P(P), .G(G), .C\_in(C\_in), .C\_out(C));  addHalf AH0(.a(A[0]), .b(B[0]), .sum(P[0]), .c\_out(G[0]));  addHalf AH1(.a(A[1]), .b(B[1]), .sum(P[1]), .c\_out(G[1]));  addHalf AH2(.a(A[2]), .b(B[2]), .sum(P[2]), .c\_out(G[2]));  addHalf AH3(.a(A[3]), .b(B[3]), .sum(P[3]), .c\_out(G[3]));  assign Sum[0] = P[0] ^ C[0];  assign Sum[1] = P[1] ^ C[1];  assign Sum[2] = P[2] ^ C[2];  assign Sum[3] = P[3] ^ C[3];  assign C\_out = C[4];  endmodule |

|  |
| --- |
| addhalf.v |
| `timescale 1ns / 1ps  module Add\_half( //half adder  input a,b;  output c\_out, sum;  );  assign sum = a ^ b; //xor operation  assign c\_out = a & b;  endmodule |

|  |
| --- |
| carrybits.v |
| module carryBits(  input [3:0]P,  input [3:0]G,  input C\_in,  output[4:0]C\_out  );  assign C\_out[0] = C\_in; //the following carry bits follow rules to assign bits in parrallel  assign C\_out[1] = G[0] | P[0] & C\_in;  assign C\_out[2] = G[1] | P[1] & G[0] | P[1] & P[0] & C\_in;  assign C\_out[3] = G[2] | P[2] & G[1] | P[2] & P[1] & G[0] | P[2] & P[1] & P[0] & C\_in;  assign C\_out[4] = G[3] | P[3] & G[2] | P[3] & P[2] & G[1] | P[3] & P[2] & P[1] & G[0] | P[3] & P[2] & P[1] & P[0] & C\_in;  endmodule |

|  |
| --- |
| fourBitMult.v |
| `timescale 1ns / 1ps  module fourBitMult(  input [3:0] a,  input [3:0] b,  output [7:0] product  );    wire [7:0] pp0; //partial products  wire [7:0] pp1;  wire [7:0] pp2;  wire [7:0] pp3;    wire [7:0] sum1; //temporary sums to store 8 bit added partial products  wire [7:0] sum2;  wire [7:0] sum3;    wire cout1; //temporary holder for carry out  wire cout2;  wire cout3;      assign pp0 = {4'b0000, b[0] & a[3], b[0] & a[2], b[0] & a[1], b[0] & a[0]};  assign pp1 = {3'b000,b[1] & a[3], b[1] & a[2], b[1] & a[1], b[1] & a[0], 1'b0};  assign pp2 = {2'b00,b[2] & a[3], b[2] & a[2], b[2] & a[1], b[2] & a[0], 2'b00};  assign pp3 = {1'b0,b[3] & a[3], b[3] & a[2], b[3] & a[1], b[3] & a[0], 3'b000};  eightBitCLA cla1( // adding ppo with pp1  .a(pp0),  .b(pp1),  .cin(1'b0),  .cout(cout1),  .eightSum(sum1)  );  eightBitCLA cla2( //adding pp2 with pp3  .a(pp2),  .b(pp3),  .cin(1'b0),  .cout(cout2),  .eightSum(sum2)  );  eightBitCLA cla3( adding sums of previous two partial products  .a(sum1),  .b(sum2),  .cin(1'b0),  .cout(cout3),  .eightSum(sum3) //total product  );  assign product = sum3;  endmodule |

|  |
| --- |
| pipeMult.v |
| `timescale 1ns / 1ps  //////////////////////////////////////////////////////////////////////////////////  // Company:  // Engineer:  //  // Create Date: 03/09/2020 03:29:08 PM  // Design Name:  // Module Name: pipeMult  // Project Name:  // Target Devices:  // Tool Versions:  // Description:  //  // Dependencies:  //  // Revision:  // Revision 0.01 - File Created  // Additional Comments:  //  //////////////////////////////////////////////////////////////////////////////////  module pipeMult(  input [3:0] a,  input [3:0] b,  input clk, reset,  output [7:0] product  );  wire [7:0] pp0;  wire [7:0] pp1;  wire [7:0] pp2;  wire [7:0] pp3;    wire [7:0] sum1;  wire [7:0] sum2;  wire [7:0] sum1out;  wire [7:0] sum2out;    wire cout1;  wire cout2;  wire cout3;  //assigning partial products based on rules of multiplication  assign pp0 = {4'b0000, b[0] & a[3], b[0] & a[2], b[0] & a[1], b[0] & a[0]};  assign pp1 = {3'b000,b[1] & a[3], b[1] & a[2], b[1] & a[1], b[1] & a[0], 1'b0};  assign pp2 = {2'b00,b[2] & a[3], b[2] & a[2], b[2] & a[1], b[2] & a[0], 2'b00};  assign pp3 = {1'b0,b[3] & a[3], b[3] & a[2], b[3] & a[1], b[3] & a[0], 3'b000};    eightBitCLA cla1( //add pp0 with pp1  .a(pp0),  .b(pp1),  .cin(1'b0),  .cout(cout1),  .eightSum(sum1)  );  eightBitCLA cla2( //add pp2 with pp3  .a(pp2),  .b(pp3),  .cin(1'b0),  .cout(cout2),  .eightSum(sum2)  );    dReg reg1( //store sum1 in a dreg  .clk(clk),  .reset(reset),  .en(1'b1),  .d(sum1),  .q(sum1out)  );    dReg reg2( //store sum2 in a dreg  .clk(clk),  .reset(reset),  .en(1'b1),  .d(sum2),  .q(sum2out)  );    eightBitCLA cla3( //add dreg outputs together to get the total sum aka product  .a(sum1out),  .b(sum2out),  .cin(1'b0),  .cout(cout3),  .eightSum(product)  );      endmodule |

|  |
| --- |
| button\_debouncer.v |
| `timescale 1ns / 1ps  module button\_debouncer #(parameter depth = 16) (  input wire clk, /\* 5 KHz clock \*/  input wire button, /\* Input button from constraints \*/  output reg debounced\_button  );    localparam history\_max = (2\*\*depth)-1;  /\* History of sampled input button \*/  reg [depth-1:0] history;  always @ (posedge clk) begin  /\* Move history back one sample and insert new sample \*/  history <= { button, history[depth-1:1] };    /\* Assert debounced button if it has been in a consistent state throughout history \*/  debounced\_button <= (history == history\_max) ? 1'b1 : 1'b0;  end    endmodule |

|  |
| --- |
| dReg.v |
| `timescale 1ns / 1ps  module dReg #(parameter WIDTH = 8)  (input clk, reset,  input en,  input [WIDTH - 1:0]d,  output reg[WIDTH - 1:0]q);  always @(posedge clk, posedge reset) begin  if (reset) q <=0;  else if(en) q <= d;  else q<=q;  end    endmodule |

|  |
| --- |
| clk\_gen.v |
| module clk\_gen (  input wire clk100MHz,  input wire rst,  output reg clk\_4sec,  output reg clk\_5KHz  );  integer count1, count2;  always @ (posedge clk100MHz) begin  if (rst) begin  count1 = 0; clk\_4sec = 0;  count2 = 0; clk\_5KHz = 0;  end else begin  if (count1 == 200000000) begin  clk\_4sec = ~clk\_4sec;  count1 = 0;  end  if (count2 == 10000) begin  clk\_5KHz = ~clk\_5KHz;  count2 = 0; end  count1 = count1 + 1;  count2 = count2 + 1;  end |

|  |
| --- |
| 4bitmulti\_FPGA.v |
| `timescale 1ns / 1ps  //////////////////////////////////////////////////////////////////////////////////  // Company:  // Engineer:  //  // Create Date: 03/16/2020 04:11:50 PM  // Design Name:  // Module Name: pipe\_mult\_fpga  // Project Name:  // Target Devices:  // Tool Versions:  // Description:  //  // Dependencies:  //  // Revision:  // Revision 0.01 - File Created  // Additional Comments:  //  //////////////////////////////////////////////////////////////////////////////////  module pipe\_mult\_fpga( //fpga file  input [3:0]Ain, //inputs A and B to be multiplied  input [3:0]Bin,  input clk\_FPGA, //clk from fpga board  input button\_clk,    output [7:0]LEDOUT,  output [3:0]LEDSEL,  output reg [3:0]Aout,  output reg [3:0]Bout  );    always @ (\*) begin  Aout = Ain;  Bout = Bin;  end    wire[7:0] pipe\_output;  wire fiveKhz\_clk;  wire button\_w;  wire button\_FPGA;    clk\_gen clk ( //gen clk for button debouncer  .clk100MHz(clk\_FPGA),  .rst(1'b0),  .clk\_5KHz(fiveKhz\_clk)  );  button\_debouncer db( //debounced button  .clk(fiveKhz\_clk),  .button(button\_clk),  .debounced\_button(button\_FPGA)  );  pipeMult pm( //initiate pipeline multiplier  .a(Ain),  .b(Bin),  .clk(button\_FPGA),  .reset(1'b0),  .product(pipe\_output) //product  );    wire [3:0] led3 = pipe\_output % 10;  wire [3:0] led2 = ((pipe\_output % 100) - led3)/10;  wire [3:0] led1 = ((pipe\_output % 1000) - led2)/100;  wire [3:0] led0 = ((pipe\_output % 10000) - led1)/1000;  bcd\_to\_7seg BCD (  .BCD (led),  .s (LEDOUT)  );  led\_mux LED (  .clk (clk\_5KHz),  .rst (rst),  .LED3 (led3),  .LED2 (led2),  .LED1 (led1),  .LED0 (led0),  .LEDSEL (LEDSEL),  .LEDOUT (LEDOUT)  );  endmodule |

|  |
| --- |
| mult\_fpga.xdc |
| # Clock input  set\_property -dict {PACKAGE\_PIN W5 IOSTANDARD LVCMOS33} [get\_ports {clk\_FPGA}];  create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports {clk\_FPGA}];  # Button (debouncer)  set\_property -dict {PACKAGE\_PIN U18 IOSTANDARD LVCMOS33} [get\_ports {button\_clk}]; # Center Button  # Input Switches  set\_property -dict {PACKAGE\_PIN V17 IOSTANDARD LVCMOS33} [get\_ports {Bin[0]}]; # Bin0  set\_property -dict {PACKAGE\_PIN V16 IOSTANDARD LVCMOS33} [get\_ports {Bin[1]}]; # Bin1  set\_property -dict {PACKAGE\_PIN W16 IOSTANDARD LVCMOS33} [get\_ports {Bin[2]}]; # Bin2  set\_property -dict {PACKAGE\_PIN W17 IOSTANDARD LVCMOS33} [get\_ports {Bin[3]}]; # Bin3  set\_property -dict {PACKAGE\_PIN W2 IOSTANDARD LVCMOS33} [get\_ports {Ain[0]}]; # Ain1  set\_property -dict {PACKAGE\_PIN U1 IOSTANDARD LVCMOS33} [get\_ports {Ain[1]}]; # Ain2  set\_property -dict {PACKAGE\_PIN T1 IOSTANDARD LVCMOS33} [get\_ports {Ain[2]}]; # Ain3  set\_property -dict {PACKAGE\_PIN R2 IOSTANDARD LVCMOS33} [get\_ports {Ain[3]}]; # Ain4  # LED Output  set\_property -dict {PACKAGE\_PIN U16 IOSTANDARD LVCMOS33} [get\_ports {Bout[0]}]; # Bout0  set\_property -dict {PACKAGE\_PIN E19 IOSTANDARD LVCMOS33} [get\_ports {Bout[1]}]; # Bout1  set\_property -dict {PACKAGE\_PIN U19 IOSTANDARD LVCMOS33} [get\_ports {Bout[2]}]; # Bout2  set\_property -dict {PACKAGE\_PIN V19 IOSTANDARD LVCMOS33} [get\_ports {Bout[3]}]; # Bout3  set\_property -dict {PACKAGE\_PIN P3 IOSTANDARD LVCMOS33} [get\_ports {Aout[0]}]; # Aout0  set\_property -dict {PACKAGE\_PIN N3 IOSTANDARD LVCMOS33} [get\_ports {Aout[1]}]; # Aout1  set\_property -dict {PACKAGE\_PIN P1 IOSTANDARD LVCMOS33} [get\_ports {Aout[2]}]; # Aout2  set\_property -dict {PACKAGE\_PIN L1 IOSTANDARD LVCMOS33} [get\_ports {Aout[3]}]; # Aout3  # 7 Segment Display  set\_property -dict {PACKAGE\_PIN W7 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[0]}]; # CA  set\_property -dict {PACKAGE\_PIN W6 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[1]}]; # CB  set\_property -dict {PACKAGE\_PIN U8 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[2]}]; # CC  set\_property -dict {PACKAGE\_PIN V8 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[3]}]; # CD  set\_property -dict {PACKAGE\_PIN U5 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[4]}]; # CE  set\_property -dict {PACKAGE\_PIN V5 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[5]}]; # CF  set\_property -dict {PACKAGE\_PIN U7 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[6]}]; # CG  set\_property -dict {PACKAGE\_PIN V7 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[7]}]; # DP  set\_property -dict {PACKAGE\_PIN U2 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[0]}]; # AN0  set\_property -dict {PACKAGE\_PIN U4 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[1]}]; # AN1  set\_property -dict {PACKAGE\_PIN V4 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[2]}]; # AN2  set\_property -dict {PACKAGE\_PIN W4 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[3]}]; # AN3 |