README.md

Digital Design I - Project II - Sequential Signed Multiplier

Project members:

- Ahmed Ali
- Omar Elfouly
- Bavly Remon
- Omar Anwar

Contribution

Due to the complexity of this project no member ever worked alone. Every commit was done in pairs. A commit by OmarElfouly means it was done by Omar Elfouly and Bavly Remon, while any push commmited by iRustom was done by Ahmed Ali and Omar Anwar. A general outline of contribution is:

- Demo 1 - Omar and Bavly mainly worked on combinational binary to BCD and magnitude finder, while Ahmed Ali and Omar Anwar worked on unsigned multiplier and display.
- Demo 2 - Omar and Bavly mainly worked on multiplier, display, and circ. Ahmed Ali and Omar Anwar mainly worked on control unit and 7-segment display.
- Demo 3 - All members improved readability of code and applied changes according to comments made by professor Shalan. All members then worked on Report and README.

Demo 1

Dr. Shalan's unisgned multiplier, found in "Lectures 19-21: RTL & ASM Charts" slide 5, was used while designing our circuit. The bassic concept of a binary to bcd converter was taken from the double_dabbble article on wikipedia https://en.wikipedia.org/wiki/Double_dabble.

Control Signals:

- load_Inital => Loads the negative register and shift registers with their values simultaniously. It loads a zero to P register when it is High, else it gives the value of P + SHL_Register to register P.
- load_Initial(bar) => Enables shift registers (i.e. causes them to shift a single bit L/R).
- LP => a load signal that determines whether or not a value is loaded to register P.
- displaySelectControlSignal => display signal that determines which digits are displayed on the 7 seg. It choses between 4 states: Display only underscores, display rightmost digits, display center digits, display leftmost digits.

Black boxes:

- Double dabble => it's a function that converts a binary value into its BCD equivalent. In our case it converts 16 bits into 5x4bit BCD thats then passed on to the display function. It has been implmented using combinational logic.
- Display => function that chooses which BCD digits to display based on the control signal DS. In our case it chooses 3x4bit BCD from the total 5x4bit BCD provided from the double dabble. It will output 12'b1 if ds is in dash state else it will output the BCD values to be displayed.
- 7-segment function => takes the BCD digits provided by the Display and displays them on the three rightmost 7 segment displays on the FPGA, and takes the output of the Neg Reg, which contains information about whether or not

the product is negative or not, and displays a negative sign on the leftmost 7-segment display if it is negative, nothing if not.

Demo 2

Link to the signed multiplier Verilog code: https://cloudv.io/a/dd1_project (To be made public before we present)(Now outdated, please refer to the modules on this repo and in source files). Double dabbler's sequential implmentation was used to produce the binary to BCD verilog code since the function was not on Blackboard yet.

Demo 3

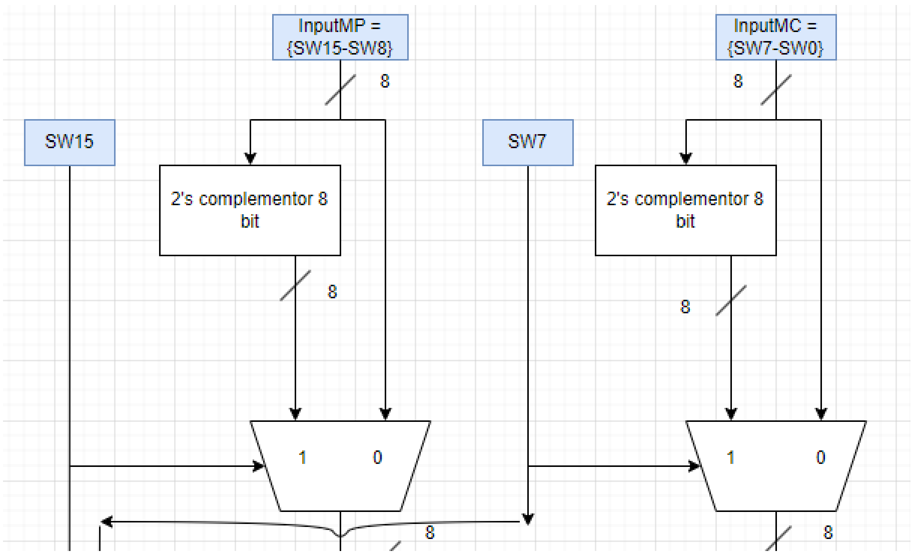
This inolved minor changes to the names and syntax of certain parts of our program. We also added license information.

The following is our Report on our project.

Report on Sequential Signed Multiplier

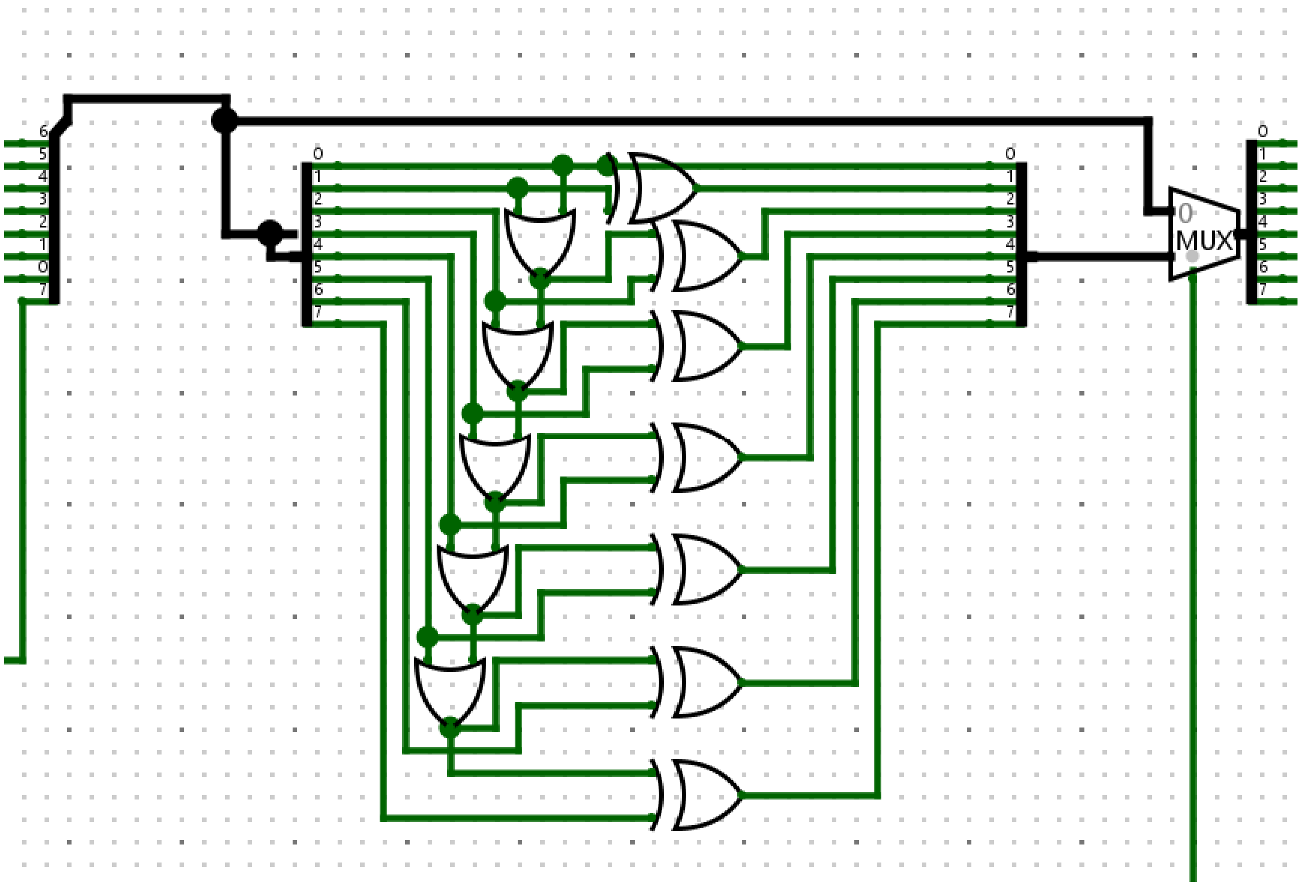
Magnitude Finder

Block Diagram



The 2’s complementors take the two binary inputs, and convert them into their magnitude. The MUX chooses whether we take the 8 bits as they are, which is in the case the sign bit is 0, indicating the input is positive, or whether we take the 8 bits 2’s complement, which is in the case the sign bit is 1, indicating that it is negative, so complementing the negative value gives us its magnitude. The selection line is, therefore, the sign bit, if 1, we complement, if 0, we take the input as is.

Logisim

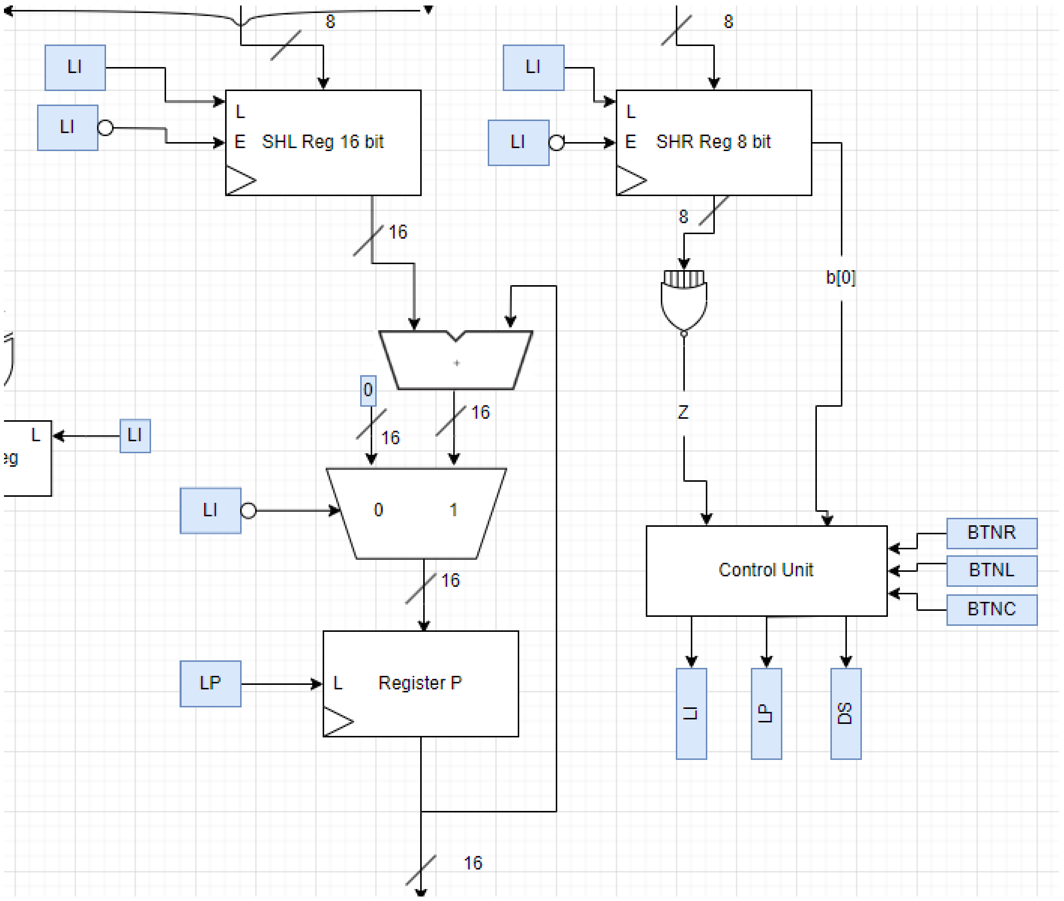


Verilog

```
module magnitudeFinder( circuitInput, magnitude);
        input wire [7:0] circuitInput;
        output wire [7:0] magnitude;

        wire [7:0] twosComp;
        assign twosComp = ~circuitInput +1'b1;
        assign magnitude = (circuitInput[7]) ? twosComp : circuitInput[7:0];


endmodule
```

Unsigned Sequential Multiplier

Block Diagram



The shift left register is of size 16 bits and takes the multiplicand. If shifting is enabled then on each positive edge of the clock, it shifts 1-bit to the left. The register’s first 8 bits are loaded with the input from the magnitude fixer and the last 8 bits are grounded , such that when the load initial signal is high the register is loaded with its initial values. These control signals are provided by the control unit which guarantees that shifting only occurs after input has been loaded.

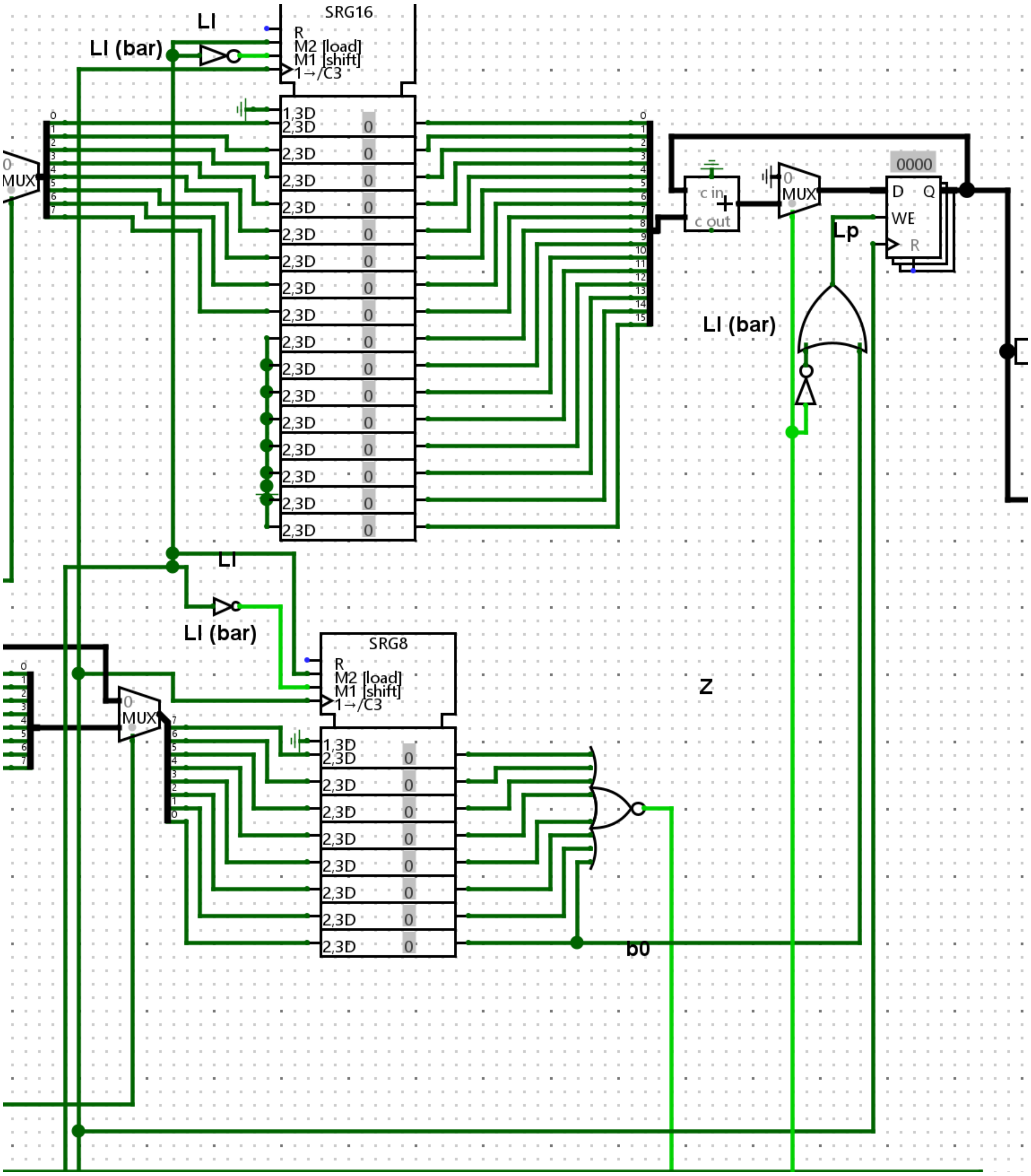
The shift right register is of size 8 bits and takes the multiplier. If shifting is enabled then on each positive edge of the clock, it shifts 1-bit to the right. The register’s 8 bits are loaded with the input from the magnitude fixer, such that when the load initial signal is high the register is loaded with its initial values. These control signals are provided by the control unit which guarantees that shifting only occurs after input has been loaded.

Register P can be considered an accumulator that either keeps its value or keeps its value plus the value of the shift left register. The mux chooses whether the input into the product register is 0, or is the accumulated adder which adds the left shift register along with the product. The decision to load is then controlled by b[0] (Which is the least significant bit of the shift right register), such that when b[0] is 1, we load the input into the product register, and when it is 0 we do not load anything. The control signals Li (bar) and LP are such that when the button is pressed, we load 0 into the product, if it is not pressed, then we load only when b[0] == 1.

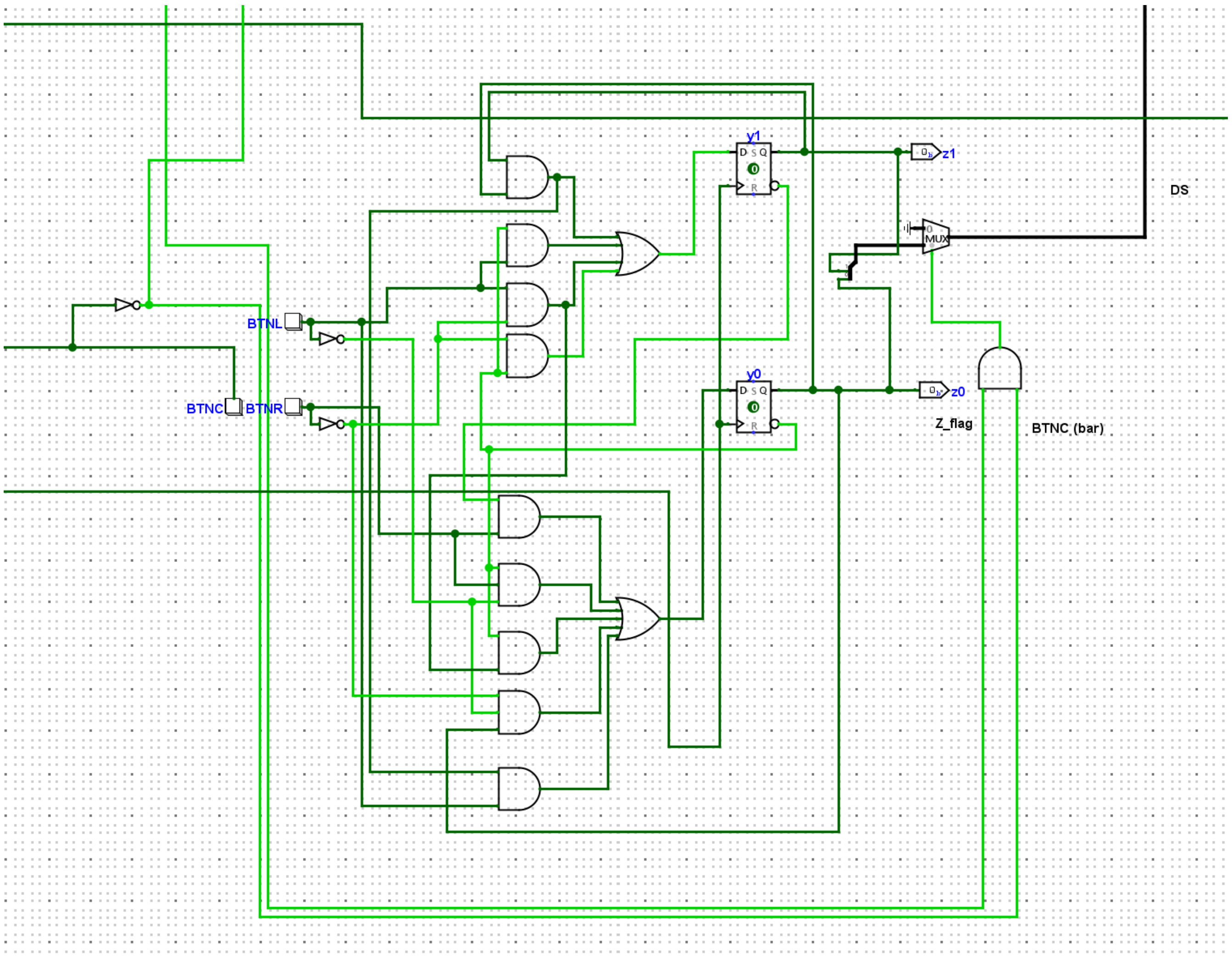
The control unit takes the buttons and b[0] and the z-flag of the left shift register as inputs. It produces the Li, which is dependent on the BTNC being clicked, and produces LP, which is dependent on BTNC and b[0] to determine whether we load the Register P or not. It also produces the display select, which uses a finite state machine that alternates between three states, and state changes are dependent on BTNR and BTNL, determining which digits are to be displayed (rightmost, middle, leftmost), hence, scrolling through the output product.

Logisim

Data path:



Control unit:



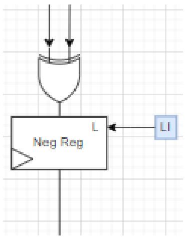
Verilog

```
module multiplier( clk, inMC, inMP, load_Initial, zeroFlag, LSB_SHRReg, product);
    input wire clk;
    input wire [7:0] inMC;
    input wire [7:0] inMP;
    input wire load_Initial;
    output wire zeroFlag;
    output wire LSB_SHRReg;
    output reg [15:0] product;

    reg [15:0] SHLReg;
    reg [7:0] SHRReg;
    reg [15:0] nextp;

    initial
    begin
        SHLReg = 16'b0;
        SHRReg = 8'b1;
        product = 16'b0;
    end

    always @(posedge clk)
    begin

        if(load_Initial)
        begin
        SHRReg <=inMP;
        SHLReg[15:0] <={8'b0,inMC};
        product <= 16'b0;

        end else
        begin

            SHLReg <= SHLReg <<1;
            SHRReg <= SHRReg >>1;

            if(LSB_SHRReg)
            begin
                product<= SHLReg+product;
            end

        end
    end

    assign LSB_SHRReg = SHRReg[0];
    assign zeroFlag = ~|SHRReg;

endmodule
```

```
module controlUnit(clk,zeroFlag, LSB_SHRReg, buttonRight, buttonCenter, buttonLeft, load_Initial, displayControlSigna
    input wire clk;
    input wire zeroFlag;
    input wire LSB_SHRReg;
    input wire buttonRight;
    input wire buttonCenter;
    input wire buttonLeft;
    output reg load_Initial;
    output wire [1:0] displayControlSignal;
    output reg calculatingFlag;


    reg [1:0] displayNextState;
    reg [1:0] displayState;
    //reg calculatingFlag;

    initial
    begin
        displayState = 2'b01;
        calculatingFlag =0;
    end

    localparam [1:0] right = 2'b01, middle = 2'b10, left = 2'b11;

    always @(*)
    begin
        case(displayState)
            right: if(buttonLeft) displayNextState = middle;
                   else displayNextState = right;

            middle: if(buttonLeft) displayNextState = left;
                    else if(buttonRight) displayNextState = right;
                    else displayNextState = middle;

            left: if(buttonRight) displayNextState = middle;
                  else displayNextState = left;

            default: displayNextState = right;
        endcase
    end
    always @(posedge clk)
    begin
        if(load_Initial)
            displayState <= right;
        else
            displayState <= displayNextState;
    end

    always @(posedge clk)
    begin
        load_Initial <= buttonCenter;
    end

    always @(posedge clk)
    begin
            if(buttonCenter)
            begin
                calculatingFlag <= 1;
            end
    end

    assign displayControlSignal = (calculatingFlag&zeroFlag & ~buttonCenter) ? displayState : 2'b00;

endmodule
```

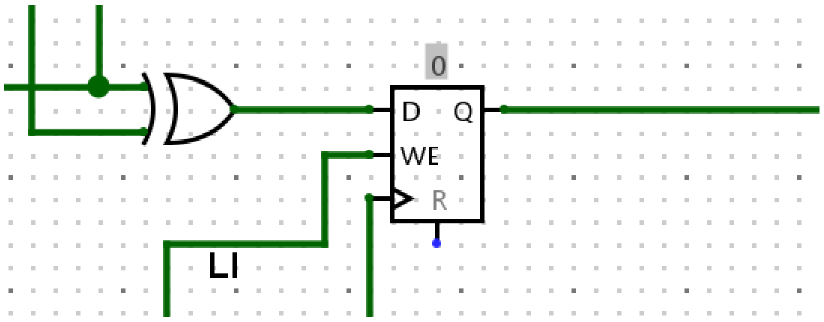
Negative Register

Block Diagram



The Neg Reg register loads the value of the sign, positive (0) or negative (1), into it, by XORing the sign bit of both inputs. This is necessary because we only want a 1 when exactly one of the inputs is negative and the other is positive, and a 0 if the signs are both the same, which implies an XOR gate. We used a register to store the value of the sign so that when the user is changing their inputs, it does not affect the sign bit on the 7-segment display, and this is done by disabling the load once the initial load of the inputs is complete, and only enabling it on a center button press.

Logisim

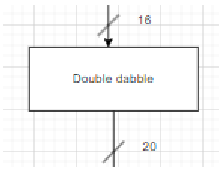


Verilog

```
module negativeBoolModule(clk,signBit0,signBit1, load_Initial, negativeProductFlag);
    input wire clk;
    input wire signBit0;
    input wire signBit1;
    input wire load_Initial;
    output reg negativeProductFlag;


    always @(posedge clk) begin
      if(load_Initial)
        negativeProductFlag <= signBit0  ^ signBit1 ;
    end

endmodule
```

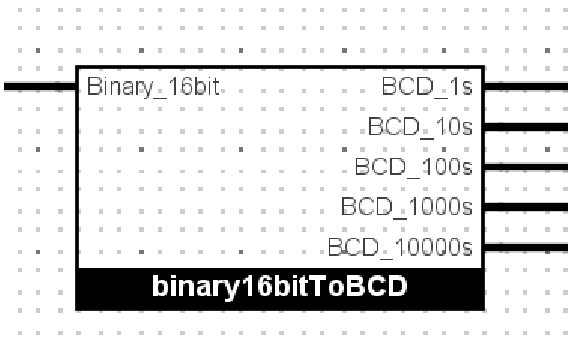
Binary to BCD

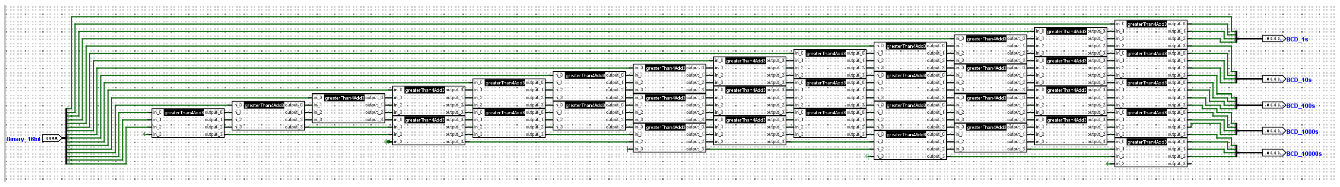
Block Diagram

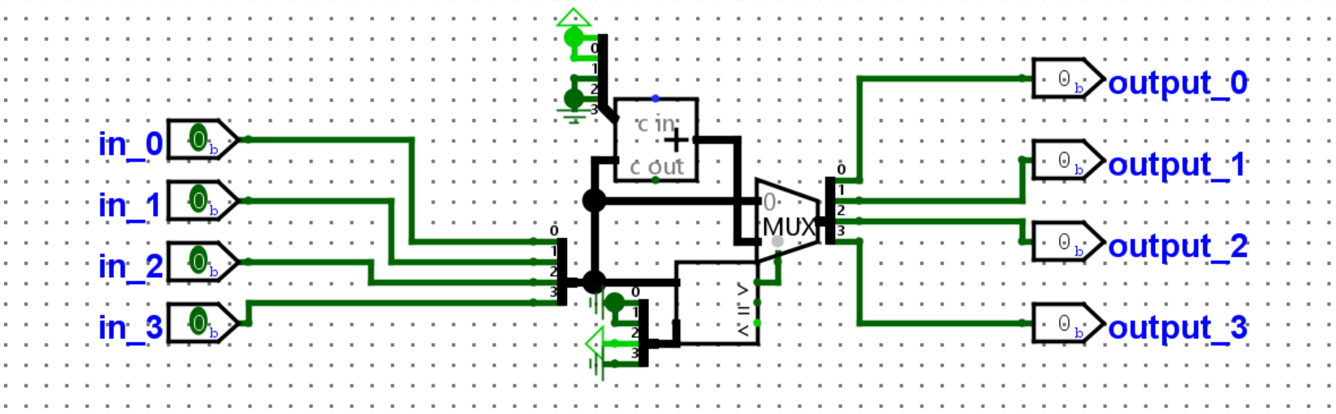


The double dabble function takes in the 16-bit binary product produced by the multiplier and uses combinational logic to convert that input into its 5x4-bit BCD equivalent, which is 20 bits, as each digit occupies 4-bits, and we have a total of 5 digits.

Logisim





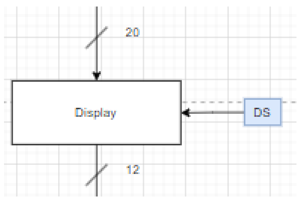


Verilog

```
module binaryToBCD (binary, BCD);
  input wire [15:0] binary;
  output reg [20:0] BCD;

  integer i,j;
  always @(binary) begin
      BCD = 21'b0;
      BCD[15:0] = binary;
      for(i = 0; i <= 12; i = i+1)
        for(j = 0; j <= i/3; j = j+1)
          if (BCD[16-i+4*j -: 4] > 4)
            BCD[16-i+4*j -: 4] = BCD[16-i+4*j -: 4] + 4'd3;
  end

endmodule
```

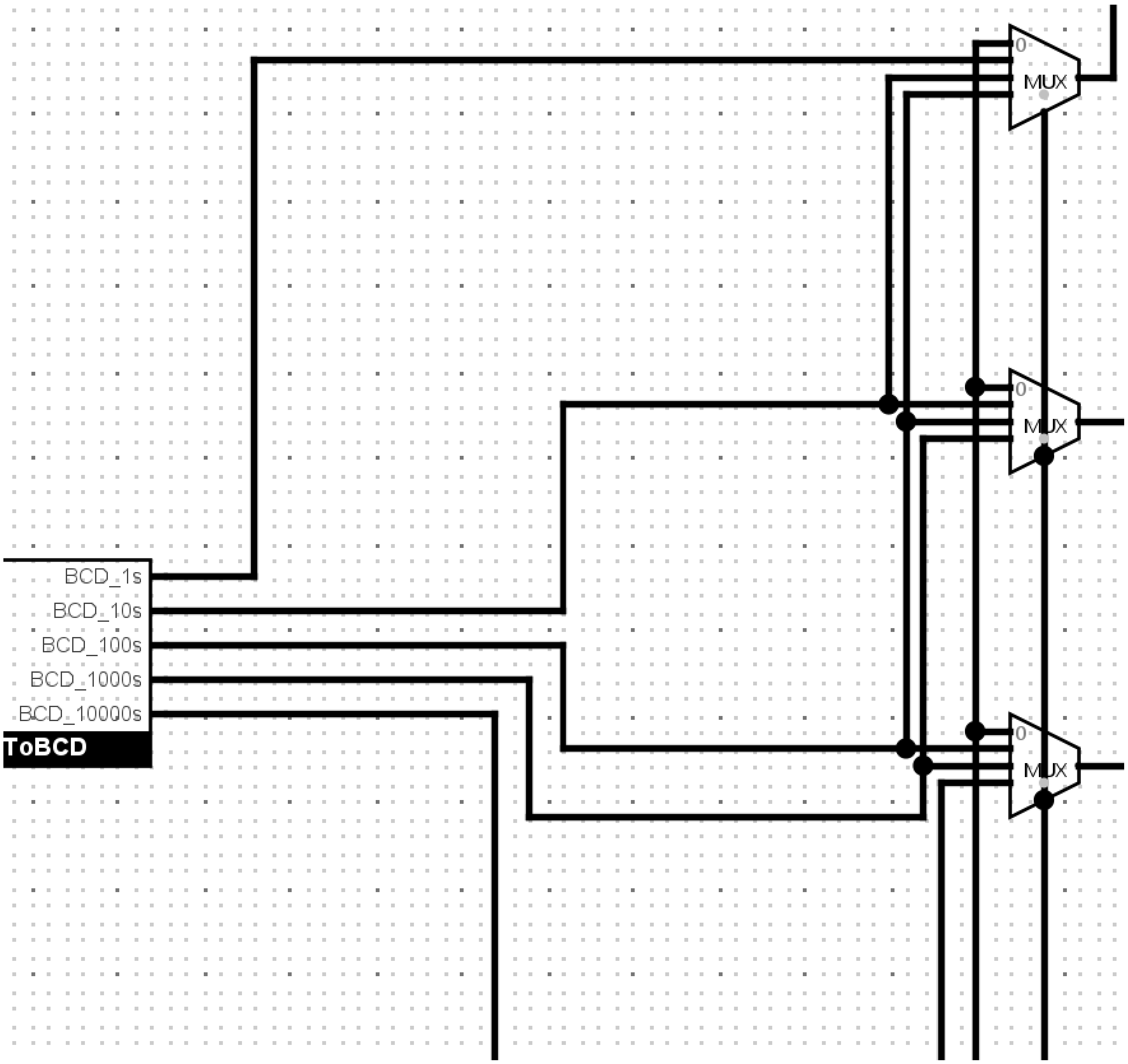
Display

Block Diagram



The display function takes in the 20 BCD bits, and takes in the display select provided by the control unit, which decides which 3 digits to display on the 7 segment display.

Logisim



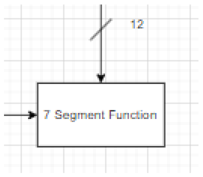
Verilog

```
module display( displayControlSignal, bcdProduct, segBCD3, segBCD2, segBCD1);
  input wire [1:0]displayControlSignal;
  input wire [19:0]bcdProduct;
  output reg [3:0] segBCD3;
  output reg [3:0] segBCD2;
  output reg [3:0] segBCD1;

  localparam [1:0] start=2'b00, right=2'b01,middle=2'b10,left=2'b11;
  localparam [3:0] underScore = 4'b1111;

  always @(*)
  begin
    case(displayControlSignal)
      start: begin
        segBCD1 = underScore;
        segBCD2 = underScore;
        segBCD3 = underScore;
      end
      right: begin
```

```
          segBCD1 = bcdProduct[3:0];
          segBCD2 = bcdProduct[7:4];
          segBCD3 = bcdProduct[11:8];
        end
        middle: begin
          segBCD1 = bcdProduct[7:4];
          segBCD2 = bcdProduct[11:8];
          segBCD3 = bcdProduct[15:12];
        end
        left: begin
          segBCD1 = bcdProduct[11:8];
          segBCD2 = bcdProduct[15:12];
          segBCD3 = bcdProduct[19:16];
        end
      endcase
    end
  endmodule
```

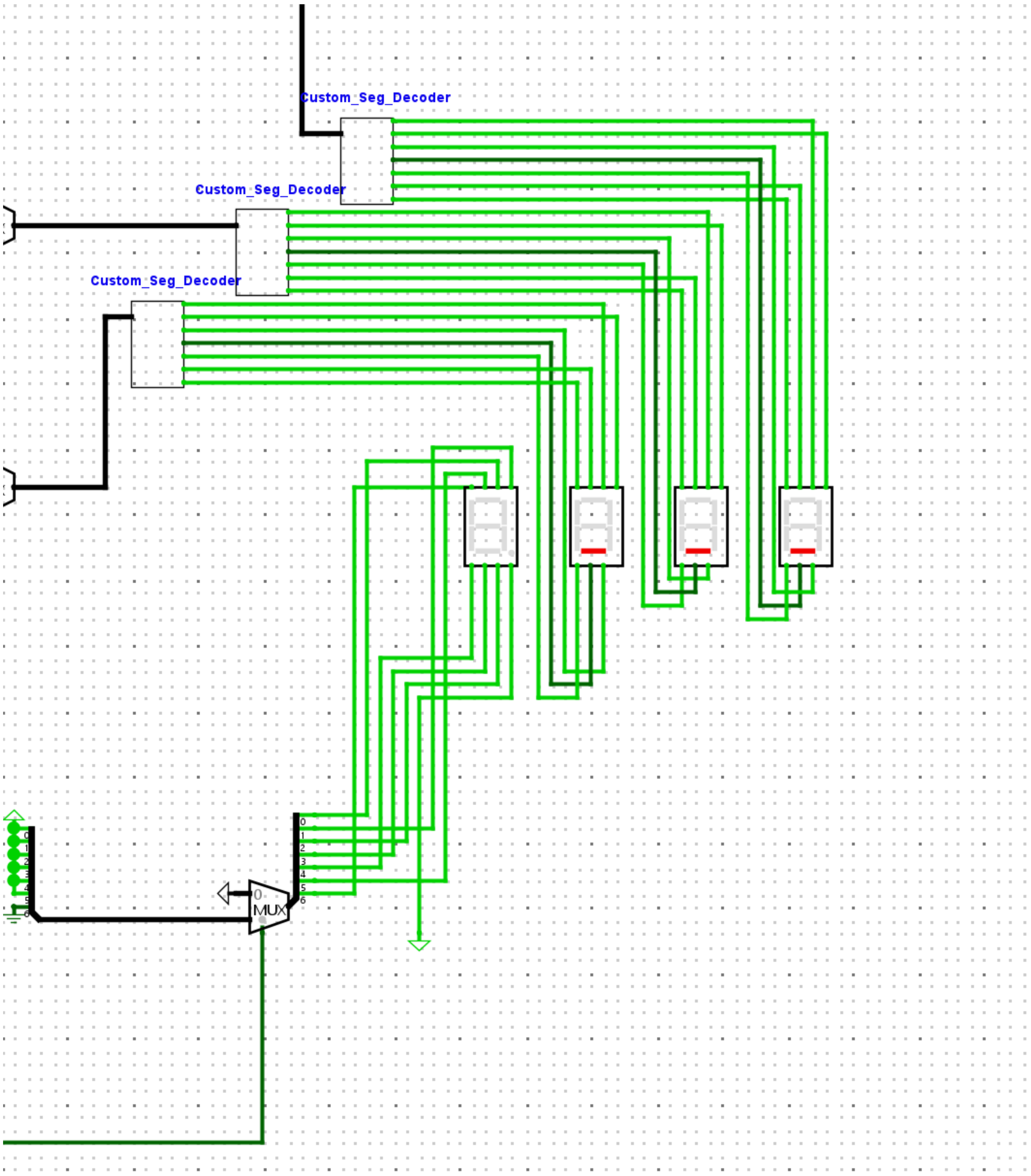
7 Segment function

Block Diagram



The 7-segment function takes the 3 BCD digits, and negative bool then decodes them into their corresponding 7-segment binary bits, which then drive the display on the board, and also produces the negative or positive sign on the board. The function also alternates between the displays on the board at a high enough frequency such that all 4 segments are apparent at the same time to the naked eye. The function also displays “_” in place of the digits before a multiplication takes place.

Logisim



Verilog

```
module SevenSegDisplay(inclk,segBCD3,segBCD2,segBCD1,product,negativeProductFlag,anode_active,segments);
    input wire inclk;
    input wire [3:0] segBCD3;
    input wire [3:0] segBCD2;
    input wire [3:0] segBCD1;
    input wire [13:0] product;
    input wire negativeProductFlag;
    output reg [3:0] anode_active;
    output reg [6:0] segments;
    wire [1:0] toggle;
    wire TOGClk;
    clockDivider #(50000) TOGClkDiv(.clk(inclk),.rst(reset) ,.clk_out(TOGClk));

    wire enOn = 1'b1;
    counterModN #(2,4) binCounter2 (.clk(TOGClk),.reset(rst),.en(enOn), .count(toggle));

    reg [3:0] numToDisplay;

    localparam [3:0]  nothing = 4'b1101, negative =4'b1110, underscore = 4'b1111;

    always @(*)
    begin
      case(toggle)
        0: numToDisplay <= segBCD1;
        1: numToDisplay <= segBCD2;
        2: numToDisplay <= segBCD3;
        3:begin
            if(product==0)numToDisplay <=nothing;// to prevent -0
            else numToDisplay <= negativeProductFlag? negative : nothing;
        end
      endcase
    end
    always @(*) begin
        case(toggle)
            2'b00: anode_active = 4'b1110;
            2'b01: anode_active = 4'b1101;
            2'b10: anode_active = 4'b1011;
            2'b11: anode_active = 4'b0111;
        endcase
    end
    always @(*) begin

      case(numToDisplay )
        0: segments = 7'b0000001;
        1: segments = 7'b1001111;
        2: segments = 7'b0010010;
        3: segments = 7'b0000110;
        4: segments = 7'b1001100;
        5: segments = 7'b0100100;
        6: segments = 7'b0100000;
        7: segments = 7'b0001111;
        8: segments = 7'b0000000;
        9: segments = 7'b0000100;
        nothing: segments =7'b1111111;
        negative: segments =7'b1111110;
        underscore: segments =7'b1110111;
        default: segments=7'b1110111;
      endcase

    end
endmodule
```

FPGA specific code and other modules

```
module clockDivider #(parameter n = 5000000)(clk,rst,clk_out);
  input wire clk;
  input wire rst;
  output reg clk_out;

  reg [31:0] count;
  always @ (posedge clk, posedge rst) begin
    if (rst == 1'b1) // Asynchronous Reset
```

```
      count <= 32'b0;
    else if (count == n-1)
      count <= 32'b0;
    else
      count <= count + 1;
  end
  always @ (posedge clk, posedge rst) begin
    if (rst) // Asynchronous Reset
      clk_out <= 0;
    else if (count == n-1)
      clk_out <= ~ clk_out;
    end
endmodule
```

```
module pushButtonDetector( clk, rst, uncleanInput, cleanOutput);
    input wire clk;
    input wire rst;
    input wire uncleanInput;
    output wire cleanOutput;

    wire newclk;
    clockDivider #(5000) newclkDiv(.clk(clk),.rst(rst) ,.clk_out(newclk));

    wire postBounce;
    debouncer d(.clk(newclk),.rst(rst),.in(uncleanInput),.out(postBounce));

    wire postSynch;
    synchronizer s(.clk(newclk),.sig(postBounce),.sig1(postSynch));

    risingEdgeDetector r(.clk(newclk), .level(postSynch),.tick(cleanOutput));

endmodule
```

```
module debouncer(clk,rst,in,out);
    input wire clk;
    input wire rst;
    input wire in;
    output wire out;

    reg q1,q2,q3;
    always@(posedge clk, posedge rst) begin
        if(rst == 1'b1) begin
            q1 <= 0;
            q2 <= 0;
            q3 <= 0;
        end
        else begin
            q1 <= in;
            q2 <= q1;
            q3 <= q2;
        end
    end
    assign out = (rst) ? 0 : q1&q2&q3;

endmodule
```

```
module synchronizer( clk, sig, sig1);
    input wire clk;
    input wire sig;
    output reg sig1;
    reg meta;
    always @(posedge clk)begin
        meta <= sig;
        sig1 <= meta;
    end
endmodule
```

```
module risingEdgeDetector(clk, level, tick);
    input wire clk;
    input wire level;
    output wire tick;

    reg [1:0] state, nextState;
    reg nextOut;
    localparam [1:0] zero=2'b00, positiveEdge=2'b01, one=2'b10;//localparam is not supported by vivado

    always @ (level or state)
    case (state)
        zero: if (level==0) nextState = zero;
            else nextState = positiveEdge;
        positiveEdge: if (level==0) nextState = zero;
            else nextState = one;
        one: if (level==0) nextState = zero;
            else nextState = one;
        default: nextState = zero;
    endcase

    always @ (posedge clk ) begin
        state <= nextState;
    end
    assign tick = (state==positiveEdge);
endmodule
```

```
module counterModN (clk,reset,en,count);
    input clk;
    input reset;
    input en;
    output reg [x-1:0] count;

    //input clk, reset, en;
    //output reg [x-1:0] count;
    parameter x=4, n=3;

    always @(posedge clk or posedge reset)
    begin
        if (reset)
        begin
            count <= 0;
        end else if(en)
        begin
            if(count == n-1)
            begin
                count <= 0;
            end else
            begin
                count <= count + 1;
            end
        end
    end

endmodule
```

Implmentation issues

Sequential multiplier could be optimised for speed by using an array of adders. Furthermore, its also possible to optimise for size by using an optimised signed baugh wooley multiplier. There currently exist no other known issues with our implmentation other than its below optimum speed and size.

Validation Activities

A video demo showcasing all our test cases has been recorded and uploaded to the repo. The video demo does the following operations:

- 0 x 0
- 0 x 4
- 0 x -3
- -3 x -5
- 127 x 127
- -128 x -128
- -128 x 4
- -128 x 127

Our program has been flashed onto our FPGA which we will bring with us to present in class.