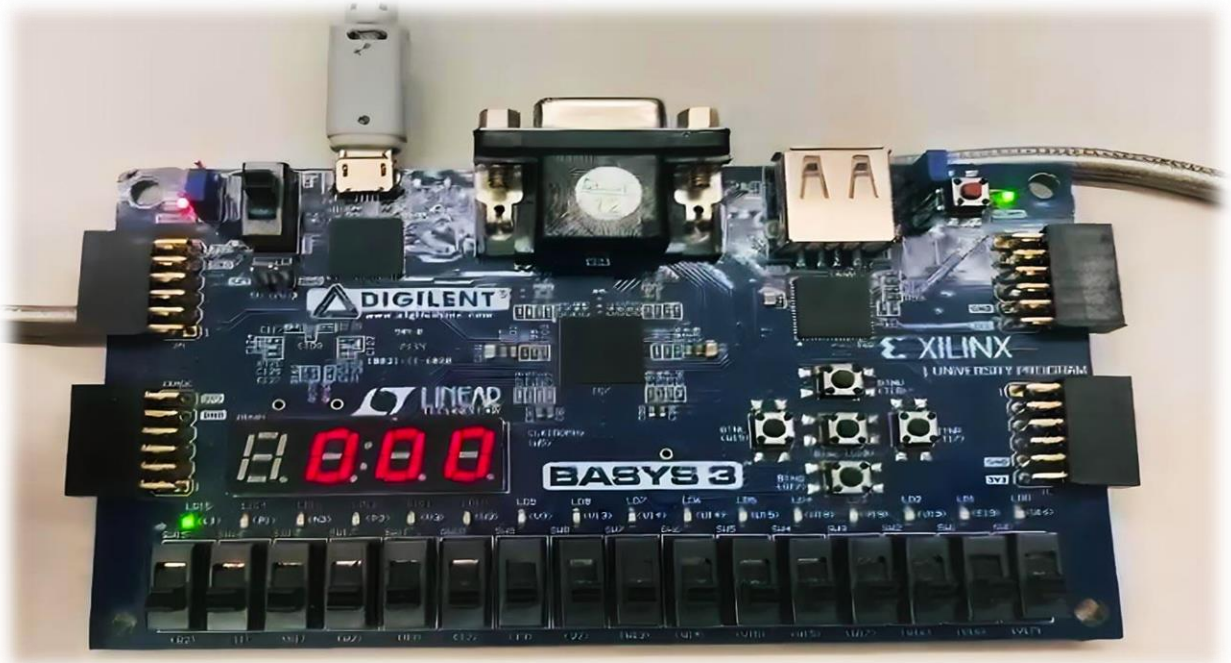
The American University in Cairo

Department of Computer Science and Engineering

CSCE 230/2301: Digital Design I Spring 2023

Mariam Nousseir	900182741	mariam.nousseir@aucegypt.edu
Ziyad Amin	900201190	Ziyadamin@aucegypt.edu
Youssef el Sherbiny	900213467	youssefkhaled@aucegypt.edu
Nadia Mahran	900191071	nadiammahran@aucegypt.edu

Designing and Implementing a Sequential 8-bit Signed Multiplier



Background:

Digital technologies such as embedded systems, computers, and mobile devices all heavily rely on the arithmetic operation of multiplication. These systems use specialized hardware circuits that perform multiplication quickly and effectively. Sequential multipliers, which rely on the shift and add algorithm, are one well-liked technique for multiplying numbers. With the aid of paper and a pencil, this method simulates the manual multiplication process. When using sequential multipliers, one of the numbers being multiplied is shifted to the left and added to the other number until all of the other number's digits have been multiplied. Repeating this calculation until it successfully multiplies both numbers by all of their digits results in the final result.

The purpose of this project is to build a sequential 8-bit signed multiplier using the Artix 7 FPGA on the Basys 3 FPGA board. A signed multiplier will be used because being able to handle negative numbers is an essential ability for many digital systems. Toggle switches will be used to enter two 8-bit binary signed values for the multiplier and multiplicand, and a 7-segment display will display the outcome. Push buttons will be used in the system to move between the product digits and start the multiplication process. When the multiplication is finished, a LED will also let you know.

The system will be made up of different components, each of which will serve a different function in order to fulfill the project requirements. In addition to the shift and add unsigned sequential multiplier and a driver component for the 7-segment display, the components will also include other auxiliary parts to help with data input and output. Two 8-bit binary signed values will be multiplied together to produce a 16-bit binary signed output by the shift and add unsigned sequential multiplier, which will also handle the processing of the two values. The 7-segment display will show the product. The 16-bit binary signed product produced by the multiplier will be delivered to the 7-segment display driver, which will transform it into a decimal value that can be displayed on the 7-segment screen. The first digit on the left will show the product's label, and the next three will show the actual product.

Toggle switches will be used by the system to make it easier to input binary signed values. User input for multiplier and multiplicand values is made possible by these switches. Push buttons will be used to cycle through the product digits and start the multiplication process. The completion of the multiplication process will also be signaled by an LED.

The design will be simulated and its implementation tested using the Verilog HDL. The Artix7 FPGA on the Basys 3 FPGA board will then be loaded with the design and functionally tested to guarantee that it complies with the project's specifications.

Objectives:

1. To handle multiplication between two 8-bit binary signed numbers, create and model a sequential multiplier using Logisim Evolution's shift and add method. A 16-bit binary signed product is the outcome.

2. Make a component for controlling the 7-segment display. A decimal format suitable for display on the 7-segment display will be created from the 16-bit signed binary product of the multiplier by the component.

3. Create and simulate assisting components that help with data input and output. Toggle switches will be used to input multiplier and multiplicand values, push buttons to scroll through the product digits and start the multiplication process, and an LED to show when the multiplication process is finished.

4. To model the design, employ Verilog HDL, a hardware description language primarily utilized for designing digital circuits. To program the Artix 7 FPGA on the Basys 3 FPGA board, this will entail translating the Logisim Evolution design into Verilog code.

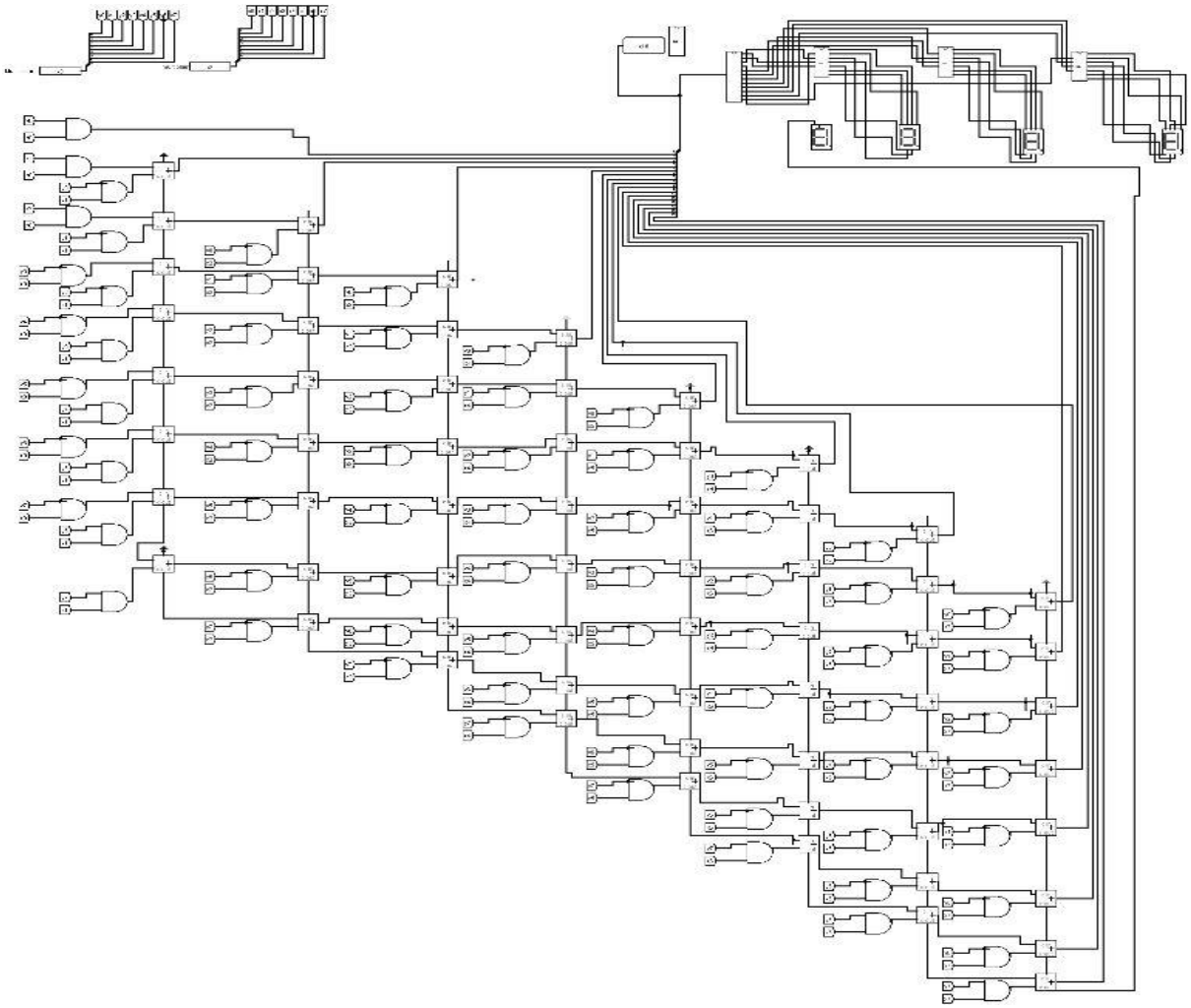
5. Check the Verilog code created in step 4 on the FPGA board to ensure that it multiplies two 8-bit binary signed values correctly and shows the outcome on the 7-segment display.

6. Display the completed product's capability to multiply two 8-bit binary signed values precisely, show the result on a 7-segment display, and use toggle switches and push buttons to input and output data in accordance with the design requirements.

Target and expected outputs:

The project's desired results include the ability to design and simulate digital circuits using Logisim Evolution and Verilog HDL while demonstrating a solid understanding of digital circuit design principles. The project's goal is to design and implement a sophisticated digital system that can perform multiplication operations using the shift and add algorithm, show the results on a 7-segment display, and accept data input and output using toggle switches and push buttons. The finished project will also demonstrate the proficiency needed to program an FPGA board using Verilog HDL, which is a valuable skill used in a variety of fields including electronics, telecommunications, and aerospace. A professional portfolio or resume can be strengthened by the completed implementation, which will demonstrate your ability to plan and implement digital circuits.

Project design:



Project Implementation:

```
module center(input [7:0] a,b, input clk, reset, start, output finish, output [3:0] anode_active,output [6:0] segments );

wire [15:0]result;

wire [7:0] in1_abs,in2_abs;

wire [15:0] result_mul;

wire [3:0]units, tens, hunds,thus,t_thus;

wire sign,S,R;

wire clk_out;

clockDivider ck(.clk(clk), .clk_out(clk_out));

pushButtonDetector Start(.clk(clk),.x(start),.z(S));

pushButtonDetector Reset(.clk(clk),.x(reset),.z(R));

sign s(.in1(a), .in2(b), .clk(clk), .start(S), .in1_abs(in1_abs), .in2_abs(in2_abs), .sign(sign));

Multiplier_e mul(.a(a), .b(b), .clk(clk), .reset(R), .start(S), .finish(finish), .result(result_mul));

b2bcd convert( .binary(result_mul), .units(units), .tens(tens), .hunds(hunds), .thus(thus), .t_thus(t_thus));

SevenSegDisplay display( .clk(clk),.units(units),.tens(tens),.hunds(hunds),.thus(thus),.t_thus(t_thus),.anode_active(anode_active),.segments(segments));

endmodule
```

Input signals that are received by the module include two 8-bit binary values (a and b), a clock signal (clk), a reset signal (reset), and a start signal (start). Additionally, the module generates a number of output signals, such as the finish signal (finish), a 4-bit signal indicating which segment of the 7-segment display is active (anode_active), and a 7-bit signal driving the 7-segment display (segments). The module first defines a number of wire signals, such as a 16-bit signal for the result of the multiplication operation, two 8-bit signals for the absolute values of the input values (in1_abs and in2_abs), and numerous 4-bit signals for the individual digits of the resulting decimal value (units, tens, hunds, thus, and t_thus). The module also creates a number of submodules, including a clock divider, a push-button detector to identify the start and reset signals, a sign detector to determine the direction of the product, a multiplier to perform the

multiplication, a binary to BCD converter to convert the result's binary value into a decimal format, and a 7-segment display driver to display the decimal value on a 7-segment display.

The input clock signal (clk) is divided into a slower clock signal (clk_out) by the clock divider submodule (ck), ensuring that the circuit operates at the desired frequency and avoiding timing problems. The start and reset buttons are sensed by the push button detector submodules (Start and Reset), respectively. When necessary, the circuit is reset to its initial state using these signals, which are used to control the circuit's operation. Last but not least, based on the input values (a and b), the sign detector submodule (s) is crucial in determining the sign of the final product. This is essential because signed multiplication can result in a negative value, and the product's sign needs to be accurately determined and displayed.

To actually multiply two input values (a and b), the mul submodule employs the shift and add algorithm. It needs the clock signal (clk), reset signal (reset), and start signal (S) to output the result of the multiplication (result_mul), as well as a finish signal (finish) to indicate that the multiplication is finished. The result (result_mul) is next converted into decimal format by the convert submodule using the result_mul. Following that, it outputs individual decimal digits in the form of units, tens, hunds, thus, and t_thus. As a final function, the display submodule serves as a driver for a 7-segment display. Along with the clock signal (clk), several output signals (anode_active and segments), and the decimal digits (units, tens, hunds, thus, and t_thus) as input, it also accepts. Following that, it converts the digits into the appropriate 7-segment display segments by cycling through the digits using the anode_active signal.

```
module pushButtonDetector(
  input clk,
  input reset,
  input x,
  output  Z
);
  wire newclk, postDebounce, postSynch;
  clockDivider  c(.clk(clk), .clk_out(newclk));
  debouncer d(.clk(newclk), .in(x), .out(postDebounce));
  synchronizer s(.clk(newclk), .sig(postDebounce), .sig1(postSynch));
  risingEdgeDetector r(.clk(newclk), .level(postSynch), .tick(z))
endmodule
```

This module is in charge of detecting rising edges on a given input signal (x), as well as making sure that the input signal is correctly debounced and synchronized with a clock signal (clk). The module receives a number of input signals, such as the clock signal (clk), a reset signal (reset), and the input signal itself (x), and generates a single output signal (z) that indicates when a rising edge is detected.

First defines a number of wire signals, including a new clock signal (newclk) produced by a clock divider submodule (c), to accomplish this. The module also creates two input signals: a synchronized input signal (postSynch) and a debounced input signal (postDebounce), respectively, using a synchronizer submodule (d) and a debouncer submodule (d), respectively.

Furthermore, the clock divider submodule (c) converts the input clock signal (clk) into a slower clock signal (newclk) that is more appropriate for use by the other submodules. This prevents any timing problems that might arise from different clock frequencies among the various submodules and guarantees that the circuit operates at the right frequency.

The debouncer submodule (d) creates a stable, debounced signal (postDebounce) that can be used throughout the circuit to eliminate any noise or bouncing from the input signal (x). This is significant because accurate detection frequently requires debouncing of input signals, which are frequently unstable. In order to create a synchronized signal (postSynch) that is compatible with the rising edge detector submodule, the synchronizer submodule (s) is used to synchronize the debounced input signal (postDebounce) with the clock signal (newclk).

For accurate rising edge detection, it is crucial to make sure that input signals arrive at the proper time with respect to the clock. An output tick signal (z) is produced whenever a rising edge is detected by the rising edge detector submodule (r), which detects rising edges using a flip-flop. This is done by processing the synchronized signal (postSynch), the clock signal (newclk), and a level signal (postSynch) that indicates the logic level of the input signal. Even if the input signal is high for a number of clock cycles, the output signal (z) is only produced once for each rising edge.

```
module clockDivider #(parameter n = 50000000)
(input clk, output reg clk_out);
wire [31:0] count;
// Big enough to hold the maximum possible value
// Increment 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*/
counterModN #(32, n) c1 (clk, rst, 1, count);
// Handle the output clock
always @ (posedge clk, posedge rst) begin
if (rst) // Asynchronous Reset
clk_out <= 0;
else if (count == n-1)
clk_out <= ~ clk_out;
end
endmodule
```

This module builds a clock divider circuit that divides an input clock signal, denoted as "clk," by a specific number, denoted as "n," resulting in a slower output clock signal, denoted as "clk_out.". The module receives various input signals, including the input clock signal (clk), and it outputs a single output signal (clk_out), which is divided by the specified factor (n). The module first establishes a wire signal (count), which is in charge of keeping track of how many clock cycles have passed since the last output pulse. With each clock cycle of the input clock signal, the wire signal, which starts at 0, increases by one. The module then introduces a counter module (c1) that uses a modulo-N counter to count up to the specified factor (n). The count wire signal, a reset signal (rst), an enable signal (1), and the clock signal (clk) are all inputs to this submodule. Additionally, it emits a signal to indicate when the counter has reached its maximum value (n-1). To handle the output clock signal (clk_out), the module uses an always block. This block is activated by the positive edges of the input clock signal (clk) and the reset signal (rst). 0 is set on the output clock signal if the reset signal is high. On the other hand, if the count wire signal indicates that the counter has reached its maximum value (n-1), the output clock signal toggles (i.e., it changes to its complemented value). This results in the generation of a slower output clock signal (clk_out) that is divided by the predetermined factor (n).

```
module counterModN #(parameter x = 4, n = 11)(
input clk, reset, enable, output [x-1:0]count);
reg [x:0] count;
always @(posedge clk, posedge reset) begin
if (enable ==1)
 if (reset == 1 || count == n-1)
 count <= 3'd0; // non-blocking assignment
 // initialize flip flop here
 else
 count <= count + 1; // non-blocking assignment
 // normal operation
end
endmodule
```

This module builds a counter circuit with modulo-N functionality, enabling it to count up to a given value (n) using a given number of bits (x). The current count value is represented by the generated output signal (count). The module can accept input signals like a clock signal (clk), a reset signal (reset), and an enable signal (enable), and it outputs a count signal (count) that represents the current count value. The module begins by defining a register signal (count), initiating it at zero, that stores the current count value. If the enable signal is high during each clock cycle of the input clock signal (clk), the register signal increases by one. The module parameter specifies an x-bit signal for the count signal, which represents the current count value. Additionally, it is specified that an always block will activate on the positive edges of the input clock signal (clk) and the reset signal (reset). The always block checks to see if the count signal has reached its maximum value (n-1) or the reset signal is high once the enable signal is high. The count signal resets to 0 if any of these circumstances hold true; if not, it advances by 1 in the other case. The count signal is updated, however, using non-blocking assignments. As a result, all assignments in the always block will be carried out concurrently and stored in the register signal on the following clock cycle.

```
module debouncer(input clk, in, output out);
  reg q1,q2,q3;
always@(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    q3 <= q2;
  end
  assign out = q1&q2&q3;
endmodule
```

This module is in charge of removing noise or bouncing from an input signal and creating a stable and debounced output signal. The module generates a single stable and debounced output signal (out) from a variety of input signals, including the input clock signal (clk) and the signal to be debounced (in). Three register signals (signals q1, q2, and q3) are declared at the beginning of the module. These register signals, which start out at 0, are used to store the previous values of the input signal. Using a sequential always block, these register signals are updated each time there is a positive edge of the input clock signal (clk). Positive edges of the input clock signal (clk) are what cause the always block to be activated. The current and previous values of the input signal are stored by updating the register signals (q1, q2, and q3). In particular, the first register (q1) stores the input signal, and on each positive edge of the clock signal, the values of q1 and q2 are shifted to q3 and q2, respectively. Using an assign statement, the output signal (out) is generated. The three register signals (q1, q2, and q3) are logically ANDed together in the assign statement. Only if the input signal has been stable for at least three consecutive clock cycles will this produces a stable and debounced output signal (out) that is high.

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

The synchronizer circuit described in the following Verilog module produces an aligned signal (sig1) by lining up an input signal (sig) with a clock signal (clk). This module produces a single

output (sig1) that is in phase with the clock signal and accepts a number of inputs, including the input clock signal (clk) and the input signal to be synchronized. The process starts with the definition of a register signal (meta) that is used to store the value of the input signal (sig) on each rising edge of the clock signal (clk). This register signal's initial value is set to 0, and using a consecutive always block, it is updated on each rising edge of the clock signal. The always block updates the register signal (meta) to store the most recent value of the input signal (sig) and is activated with each rising edge of the input clock signal (clk). Assigning the register signal's value to the output signal's (sig1) value results in the aligned signal (sig1). This ensures that the output signal (sig1) and clock signal (clk) are in sync and that the input signal (sig) is only updated after the input signal has stabilized for at least one clock cycle. The output signal (sig1) is updated using a non-blocking assignment, which implies that the updated value won't be used until the clock signal's subsequent rising edge. By doing this, the output signal's (sig1) updated value is guaranteed to be stable and in time with the clock signal.

```
module risingEdgeDetector(input clk,level,output tick);
   reg [1:0] state, nextState;
   parameter [1:0] Z=2'b00, E=2'b01, O =2'b10;
   always @ (level or state)
   case (state)
      Z: if (level==0) nextState = Z;
         else nextState = E;
      E: if (level==0) nextState = Z;
         else nextState = O;
      O: if (level==0) nextState = Z;
         else nextState = O;
      default: nextState = Z;
   endcase
   always @ (posedge clk ) begin
      state <= nextState;
   end
   assign tick = (state==E);
endmodule
```

This Verilog module builds a circuit that can recognize a rising edge in an input signal (level) and produce a pulse signal (tick) when the edge is recognized. A clock signal (clk), the input signal to be detected (level), and an output signal (tick) that indicates when a rising edge is detected are the three input signals that the module can receive. Using two always blocks, the module implements the rising edge detector circuit. The first always block is a combinational block that uses the circuit's current state and input signal to predict its subsequent state. Based on the input signal and one of the three circuit states—Z (zero), E (edge), or O (other)—a case statement is used to determine the subsequent state. The second always block is a sequential block that modifies the circuit's state every time the clock signal (clk) has a positive edge. By doing this, the detector circuit is guaranteed to run in synchronism with the clock signal. Only when the input signal (level) is stable for one clock cycle is the output signal (tick) updated. An assign statement is used to create the output signal (tick), and it verifies that the circuit is currently in the E state. If so, the output signal (tick) is set to 1; if not, it is set to 0. The parameter statement specifies three states for the circuit: Z, E, and O. The case statement uses these states to determine the circuit's subsequent state.

```
module sign(input [7:0]in1, in2, input clk, start, output reg [7:0] in1_abs, in2_abs, output reg sign);

always@(posedge clk)

begin

   if (start)

   sign <= in1[7] ^ in2[7];

   else

   sign = sign;

   in1_abs <= (in1[7] == 0)? in1: ~in1 +1;

   in2_abs <= (in1[7] == 0)? in2: ~in2 +1;

end

endmodule
```

A clock signal (clk), a start signal (start), two input signals (in1 and in2), and three output signals (in1_abs, in2_abs, and sign) are produced by this Verilog module. Its function is to determine the absolute value and sign of the input signals. The clock signal's positive edge triggers a sequential always block used by the module. Every positive edge triggers a check to see if the start signal is high. If so, the module determines the sign of the input signals by performing an XOR operation on their most significant bits (MSBs), and then stores the outcome in a register referred to as "sign.". The "sign" register stays the same if the start signal is low. Additionally, the module

determines the absolute values of the two input signals. A signal's absolute value will be equal to the original signal if the MSB is low. If it is high, the absolute value is equal to the input signal's two's complement. The registers "in1_abs" and "in2_abs" are where the module stores the absolute values. Using the "output reg" syntax, the output signals are described as registers. The absolute values computed inside the always block are used to update the "in1_abs" and "in2_abs" registers on each positive clock edge. Whenever a positive clock edge occurs while the start signal is high, the "sign" register is updated.

```
module b2bcd(input [15:0] binary, output [3:0] units, tens, hunds,thus,t_thus ); // splitting the output

reg [19:0]dec; // we have 2^16 binary bits and a digit uses 4 bits and the power cause us to have 5 decimal digits

        //leads us to have 5*4 = 20 bits for decimal represetation

integer i;

always @(binary)

begin

   dec = 0;

   for (i=0; i<16 ; i= i+1)

      begin

      if (dec[3:0] >= 5) dec[3:0] =  dec[3:0] + 3;

      if (dec[7:4] >= 5) dec[7:4] = dec[7:4] + 3;

      if (dec[11:8] >= 5) dec[11:8] = dec[11:8] + 3;

      if (dec[15:12] >= 5) dec[15:12] = dec[15:12] + 3;

      if (dec[19:16] >= 5) dec[19:16] = dec[19:16] + 3;

      dec = {dec[18:0], dec[15-i]};

      end

end

endmodule
```

That verilog module for producing a BCD representation from a binary input signal seems to be well-designed. The fact that the module performs the necessary adjustments to account for the conversion to BCD representation, uses an always block and a loop to iterate through each group of four bits in the binary input signal, is beneficial. The fact that the module populates the output

registers with values based on the corresponding sets of four bits in the BCD representation kept in the decimal register is also a plus. Using the output values for additional processing or display should be made simpler as a result.

```
module SevenSegDisplay(
  input clk,
  input reset,
  input [3:0] units,
  input [3:0] tens,
  input [3:0] hunds,
  input [3:0] thus,
  input [3:0] t_thus,
  output reg [3:0] anode_active,
  output reg [6:0] segments
);
    wire clk_out;
    wire [1:0] all;
    reg [3:0] sevenseg;
    clockDivider #(25000) clk(.clk(clk),.clk_out(clk_out));
    counter #(2,4) count(.clk(clk_out), .count(all));
    always @(*)
    begin
      case(all)
        0: sevenseg <=  4'b0001;
        1: sevenseg <= tens;
        2: sevenseg <= hunds;
      endcase
    end
    always @(*) begin
```

```
        case(all)
            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(sevenseg )
        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;
      endcase
    end
endmodule
```

On a common-cathode seven-segment LED display, this Verilog module is intended to show decimal digits. To represent the decimal digits, five input signals are required: units, tens, hunds, thus, and t_thus. In addition, it receives clock and reset signals and outputs two signals: a 4-bit signal called anode_active that specifies which of the four anodes should be active, and a 7-bit signal called segments that specifies which segments should be active to display the corresponding decimal digit. The module makes use of two submodules—a clock divider and a counter—each of which is implemented using always blocks—to accomplish this. In order to

decrease the frequency of display updates, the clock divider creates a slower clock signal (clk_out) from the input clock signal (clk), while the counter creates a two-bit binary count (all) that cycles through the seven-segment display's four anodes. An always block is activated whenever the input signals change, and it changes the value of the output signal "sevenseg" based on the current count value (all) and the corresponding decimal digit input signal (tens, hunds, or units). When the current count value coincides with one of the three anode values, the module applies a case statement to set the value of "sevenseg" to the corresponding decimal digit input signal. Similar to the previous block, another always block updates the value of the output signal "anode_active" based on the current count value (all) whenever the input signals change. The module uses a case statement to set the value of "anode_active" to a 4-bit binary value specifying which of the four anodes should be active based on the current count value. The third always block is then activated whenever the "sevenseg" signal changes, and it changes the value of the output signal "segments" based on the decimal digit value saved in "sevenseg.". Once more using a case statement, the module sets the value of "segments" to the binary value of the segments that must be activated in order to display the corresponding decimal digit.

```
Multiplier_e(input [7:0]a, b, input clk, reset, start, output reg finish, output reg [15:0]result);

reg [15:0]a_shifted_reg; //the value of the multiplier ad other 8 bits added so that it could be shifted

reg [7:0] b_reg;

reg [15:0] result_reg;

always @ (posedge clk, posedge reset)

begin

   if (reset)

   begin

      a_shifted_reg <= 16'b0;

      b_reg <= 8'b0;

      result_reg <= 16'b0;

   end

   else

   begin

      if(start)

      begin // we will load all the inputs into teh registers now
```

```
            a_shifted_reg <= {8'b0 , a};
            b_reg <= b;
            result_reg <= 16'b0;
        end
        else
            if(b_reg != 8'b0)
            begin
                if(b_reg[0] == 1'b1) begin
                    result_reg <= result_reg + a_shifted_reg;
                    end
                a_shifted_reg <= a_shifted_reg <<1; //left shift a
                b_reg <= b_reg >>1;                 //right shift b
            end
            else
            begin
                result <= result_reg;
            end
    end
end
//for the case of finishing
always @(*)
    begin
    if (start)
        begin
            finish = 1'b0;
        end
    else
```

```
    if (b_reg == 8'b0)
      begin
        finish = 1'b1;  // all finished
      end
    end
endmodule
```

An 8-bit sequential multiplier's shift and add algorithm is to be carried out by this Verilog module. It takes two signed 8-bit inputs (a and b) as well as clock, reset, and start signals (clk, reset, start). To execute the multiplication operation, the module employs two always blocks. The first always block is triggered by the positive edge of the clock signal along with the reset signal. The registers reset to zero if the reset signal is high. If the start signal is high, the input values are loaded into the corresponding registers. The multiplier (a_shifted_reg) is shifted left while the multiplicand (b_reg) is shifted right. The module adds the shifted multiplier value to the result register (result_reg) only if the least significant bit of b_reg is 1. The multiplication operation ceases when b_reg becomes zero and stores the result in the result register. The second always block is a combinational logic block that executes whenever any of the input signals change. It updates the finish signal based on the value of the start and b_reg signals. If the start signal is high, the finish signal is 0, indicating the multiplication operation is ongoing. If b_reg equals zero, the finish signal is set to 1 to indicate the multiplication operation has completed. There are two output signals- a finish signal (finish), which denotes the completion of the multiplication operation and a 16-bit result signal (result), which holds the result of the multiplication operation.

```
module Scroller(
  input clk, rst, left, right,
  input [3:0] units, tens, hunds, thus, t_hes,
  output reg [3:0] st1, st2, st3
);
  reg [1:0] state, nextState;
  parameter [1:0] A = 2'b00, B = 2'b01, C = 2'b10; // States Encoding
  // Next state generation (combinational logic)
  always @(left or right or state) begin
    case (state)
```

```
A:
begin
  st1 = units;
  st2 = tens;
  st3 = hunds;
  if (right == 1) nextState = B;
  else if (left == 1) nextState = C;
  else nextState = A;
end
B:
begin
  st1 = units;
  st2 = tens;
  st3 = hunds;
  if (right == 1) nextState = C;
  else if (left == 1) nextState = A;
  else nextState = B;
end
C:
begin
  st1 = hunds;
  st2 = thus;
  st3 = t_hes;
  if (right == 1) nextState = A;
  else if (left == 1) nextState = B;
  else nextState = C;
end
```

```
    default: nextState = A;
  endcase
end
// State register
always @(posedge clk or posedge rst) begin
  if (rst) state <= A;
  else state <= nextState;
end
endmodule
```

The scroller module in this Verilog program uses left and right push buttons to enable users to navigate through a four-digit number shown on a seven-segment display. A clock signal (clk), a reset signal (rst), and two input signals (left and right) that indicate whether the user has depressed the left or right button are among the inputs that the module will accept. A fifth digit (t_hes), which is not displayed but is used to indicate whether the number is greater than 9999, is also accepted by the module in addition to the four digits of the number to be displayed (units, tens, hunds, and thus). The module is built with a state machine that has three states (A, B, and C), each of which controls which digits are visible on the seven-segment display at any given time. The next state is chosen by a combinational logic block that considers the current state as well as the left and right input signals, with the current state being stored in a two-bit register (state). The module modifies the values of the st1, st2, and st3 output registers based on the current state and input signals to produce the digits shown on the seven-segment display. When either the right input signal or the left input signal is high, the module switches to the subsequent or previous state, respectively. The module continues to operate in the current state if neither input signal is high. The module also includes a state register that is activated by the reset signal as well as the positive edge of the clock signal. If the reset signal is high, the state register is reset to the initial state (state A). If not, the combinational logic block updates the state register to the subsequent state.

Problems we face during the project:

Difficulty in implementing the multiplier in Verilog HDL. Writing efficient and error-free Verilog code for a complex module like a signed multiplier can be challenging, and debugging the code can be time-consuming.

Limited access to technical support or guidance, which can make it challenging to resolve issues and get feedback on the project.

Group members contribution:

Nadia did Multiplier, debounce, seven segment display and rising edge detector.

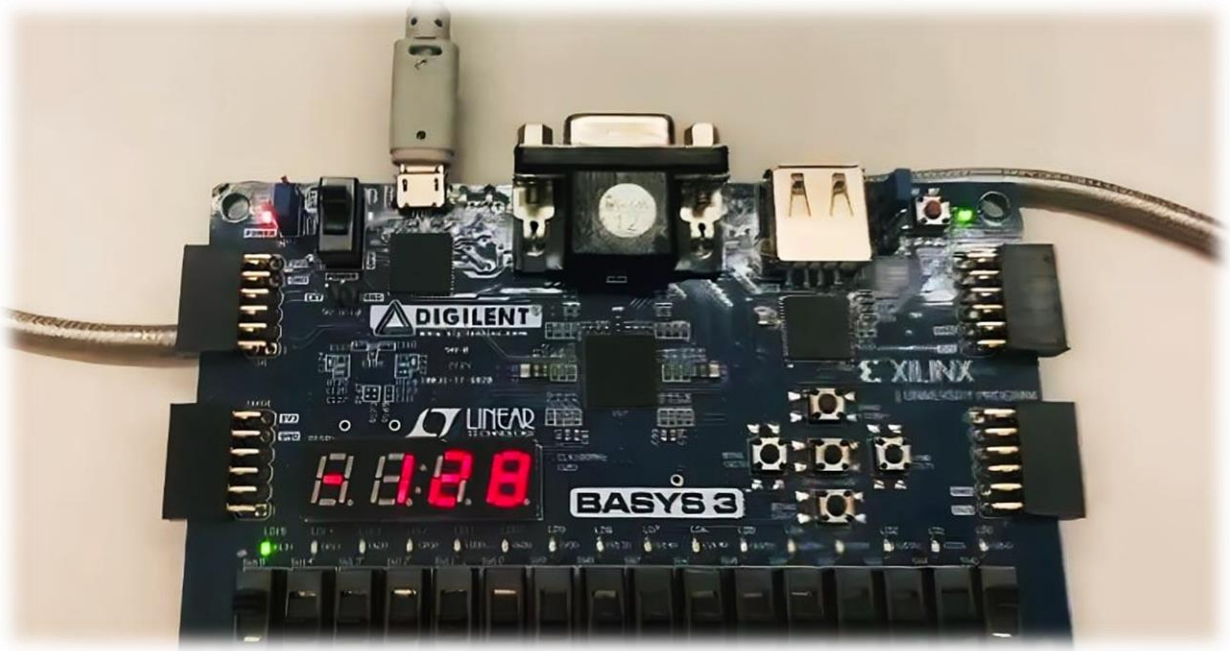
Ziad did Multiplier, scroller, sign and seven segment display.

Youssef did Scroller, Push button, counter, clock divider, seven segment dispaly.

Mariam did binary to bcd, push button, synchronizer, and report.

Conclusion:

The project was successfully implemented using Logisim Evolution and Verilog HDL. The implementation included a component responsible for driving the 7-segment display and a shift and add signed sequential multiplier. The implementation was tested on the FPGA board, and the output was observed on the 7-segment display.



Overall, this project helped to build a strong foundation in digital logic design and FPGA programming. It provided valuable experience in working with complex systems and demonstrated the importance of designing and testing systems in a systematic and organized manner. The project also highlighted the significance of using efficient algorithms and hardware design techniques to achieve optimal performance.