FPGA Based Design
Open Ended Lab

Spring 2023	
Obtained Marks	
Total Marks	
Lab Engineer Signature & Comments	

Student Names

1.	Abdul Rehman Al Nasir
2.	Riyan Alvi
3.	Munad e Ali

Experiment No: 13	Date of Submission: May-29-2023
Experiment Title: Scalable Hardware Multiplier	
Batch: BSEE 2020-24	Teacher: Dr. Shahid Nazir
Semester: 6th	Lab Engineer: Engr. Arhama Riaz, Engr. Zubaid Ali Zafar

Contents

Table of Figures

1 Abstract

This project presents the design of a serial multiplier with variable operand lengths using an Algorithmic State Machine (ASM) approach. The objective is to develop a flexible and efficient multiplication circuit capable of handling operands of different sizes. The ASM model is employed to break down the multiplication operation into a sequence of states and transitions, allowing for dynamic adjustment based on the input operand lengths. The design incorporates initialization, shifting, and addition/subtraction states to ensure accurate and reliable multiplication results. The implementation is carried out in hardware using a hardware description language (HDL) and verified through simulation. The proposed serial multiplier with variable operand lengths offers a versatile solution for applications where operand sizes may vary dynamically.

2 Introduction

2.1 Hardware Multiplier:

A hardware multiplier, also known as a multiplier circuit or multiplier unit, is a component or module within a computer or digital system that performs multiplication operations on binary numbers. It is designed to efficiently and accurately multiply two binary values together to produce a product.

In digital systems, multiplication is typically performed using a combination of logical AND, OR, and XOR operations. A hardware multiplier circuit implements these operations in a sequential or parallel manner to compute the result.

The structure of a hardware multiplier can vary depending on the desired performance, precision, and area constraints. One common type of multiplier is the array multiplier, which uses an array of logical gates to perform the multiplication. It works by breaking down the multiplication into a series of partial products and then summing them up to obtain the final result.

2.2 Scalable Serial Multiplier

A variable operand serial multiplier is a type of hardware multiplier that performs multiplication operations using a serial or sequential approach. Unlike parallel multipliers, which process multiple bits simultaneously, a serial multiplier operates on one bit at a time, starting from the least significant bit (LSB) and progressing towards the most significant bit (MSB) of the operands.

In a variable operand serial multiplier, the number of bits in the operands can vary, hence the term "variable operand." This means that the multiplier can handle inputs of different lengths, accommodating operands with different precision requirements.

2.3 Design Methodology

Designing a serial multiplier with variable operand lengths requires a flexible approach that can accommodate operands of different sizes. Scalable serial multiplier can be designed using an Algorithmic State Machine (ASM) approach.

i. Use initial trigger to initiate the state machine.
ii. Use a shift register and shift counter to keep track of received bits.
iii. Use 1st byte of data as variable that determines the length of multiplier.
iv. Store multiplier in register and add the partial product of multiplicand bits to product register and shift the product register to left by 1 bit in every cycle.
v. Stop the multiplication when all the multiplicand bits become 0.

Additional modules are required in this approach, these modules manage input and output in such way that multiplier can get suitable data and output gets displays on LCD correctly.

2.4 Serial Multiplier Block Diagram

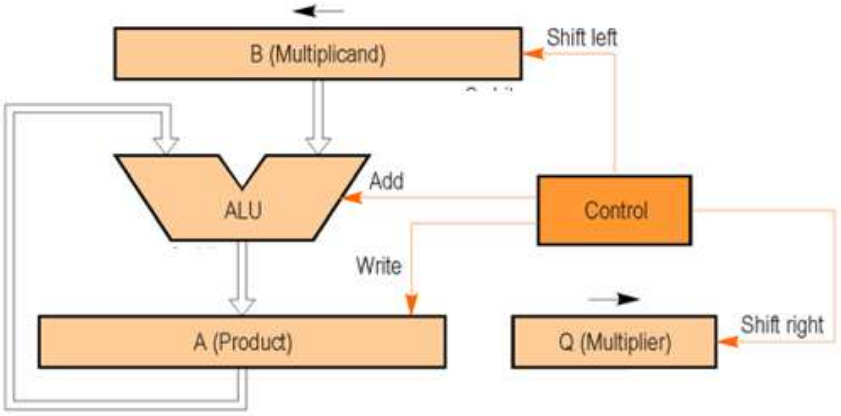

Figure 1 Serial Multiplier Block Diagram

2.5 Input/Output Handler State Machine:

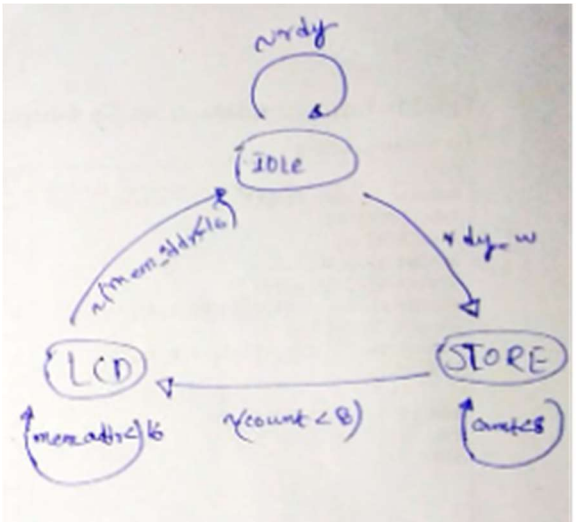

Figure 2 State Diagram

2.6 Multiplier State Machine:

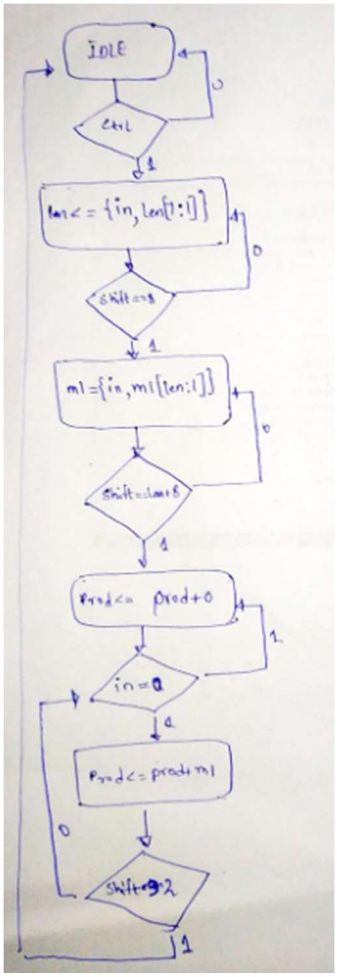

Figure 3 Algorithmic State Machine Flowchart

2.7 Implementation

2.7.1 Serial Multiplier

```
module mult(input clk,in,ctrl,rst, output reg[32:0] prod);

reg[7:0] length;
wire[5:0] shift_count;
reg[31:0] mult1;
reg[32:0] prod_reg;
wire length_bit,multiplier_bit,multiplicand_bit;
reg inbuff;

always @(posedge clk) begin
       if (rst) begin
              length<=0;
              mult1<=0;
              prod_reg<=0;
              prod<=0;
```

```
              inbuff<=0;
       end
       else begin
              inbuff<=in;
              case ({length_bit,multiplier_bit,multiplicand_bit})
              3'b100: begin length[7:0]<={inbuff,length[7:1]}; end
              3'b010: begin
                     if(shift_count==(length+8))
                     mult1<={inbuff,mult1[31:1]}>>(32-length);
                     else
                     mult1<={inbuff,mult1[31:1]};
                     end
              3'b001: begin
                     prod_reg<=prod_reg+ inbuff*(mult1);
                     mult1<=mult1<<1;
                     if (shift_count==32)begin
                     prod<=prod_reg;
                     prod_reg<=0;
                     end
                     end
              endcase
       end
end

mult_sm
sm(length,clk,rst,ctrl,shift_count,length_bit,multiplier_bit,multiplicand_bit);
endmodule
```

2.7.2 Multiplier State Machine

```
module   mult_sm(input[7:0]   mult_length,input   clk,rst,ctrl,output   reg[5:0]
shift_count, output length_bit,multiplier_bit,multiplicand_bit);
parameter [3:0] IDLE          = 4'b0001 ;
parameter [3:0] LENGTH_BIT_ST  = 4'b0010 ;
parameter [3:0] MULT1_BIT_ST  = 4'b0100 ;
parameter [3:0] MULT2_BIT_ST   = 4'b1000 ;


reg [3:0] state ;
reg [2:0] out;
always @(posedge clk) begin
       if (rst) begin
       state<=IDLE;
       out<=3'b000;
       shift_count<=0;
       end
       else begin

              case(state)
              IDLE :begin
              shift_count<=0;
                     if (ctrl) begin
                     state<=LENGTH_BIT_ST;
                     out<=3'b100;
                     end
                     else state<=IDLE;
```

```
            end
            LENGTH_BIT_ST :begin
                  shift_count<=shift_count+1;
                  if (shift_count==8) begin
                              if(shift_count>=(8+mult_length)) begin
                              state<=MULT2_BIT_ST;
                              out<=3'b001;
                              end
                              else begin
                              state<=MULT1_BIT_ST;
                              out<=3'b010;
                              end
                              end
                  else state<=LENGTH_BIT_ST;
            end
            MULT1_BIT_ST :begin
                  shift_count<=shift_count+1;
                  if (shift_count==(8+mult_length)) begin
                  state<=MULT2_BIT_ST;
                  out<=3'b001;
                  end
                  else state<=MULT1_BIT_ST;
            end
            MULT2_BIT_ST :begin
                  shift_count<=shift_count+1;
               if (shift_count==32) begin
                  state<=IDLE;

                  out<=3'b000;
                  end
                  else state<=MULT2_BIT_ST;
            end
            endcase
      end
end

            assign {length_bit,multiplier_bit,multiplicand_bit}=out[2:0];

endmodule
```

2.7.3 I/O Handler

```
module IO_w_LCD(clk,rst,ctrl,data_in,
d,rs,rw,e,sf_e,LED
    );
input clk,rst,ctrl;
input[7:0] data_in;
output[3:0] d;
output rs,rw,e,sf_e;
output reg[7:0] LED;
reg [3:0] count;
reg[32:0] buffer;
reg[7:0] RAM[0:7],mem_bus;
reg[2:0] state;
wire[32:0] data_out_w;
```

```
wire set_w;
wire[4:0] mem_addr;

parameter[2:0]  IDLE=3'b001 , STORE=3'b010 , LCD=3'b100;
assign sf_e=1;
always @(posedge clk) begin
if(rst) begin
state<=IDLE;
buffer<=0;
count<=0;
end
else begin
count<=count+1;
       case(state)
       IDLE: begin
              if(set_w) begin
                     buffer[32:0]<=data_out_w[32:0];
                     LED<=data_out_w[7:0];
                     state<=STORE;
                     count<=0;
              end
              else
                     state<=IDLE;
       end
       STORE: begin
              if (count<8)begin
              buffer<={4'b0,buffer[32:4]};
              if (buffer[3:0]<10)
              RAM[7-count[2:0]]<={4'd3,buffer[3:0]};
              else
              RAM[7-count[2:0]]<={4'd4,(buffer[3:0]-4'd9)};
              end
              else state<=LCD;
       end
       LCD: begin
       if(mem_addr<8) mem_bus[7:0]<=RAM[mem_addr[2:0]];
       else if(mem_addr<16) mem_bus<=0;
       else state<=IDLE;
       end
       endcase end
end

io_module io(data_in, ctrl,clk,rst,data_out_w[32:0],set_w);
lcd_driver display(clk,rst,rs,rw,e,d,mem_addr[4:0],mem_bus[7:0]);

endmodule
```

2.7.4 Parallel to Serial Converter

```
module io_module(
data_in, ctrl,clk,rst,data_out,set);
input ctrl,clk,rst;
input[7:0] data_in;
output [32:0] data_out;
output reg set;
```

```
reg rdy;
reg[31:0] shift,buffer;
reg[5:0] shift_count;
reg[2:0] press_count,state;

parameter [3:0] IDLE=4'b0001,READY=4'b0010,SHIFT=4'b0100;


always@(posedge ctrl or posedge rst) begin
if (rst) begin
press_count<=0;
shift<=0;
//press_count<=4;
//shift<={4'd3,4'd5,8'd4};
end
else if (press_count<4) begin
shift[31:24]<=data_in;
shift[23:16]<=shift[31:24];
shift[15:8]<=shift[23:16];
shift[7:0]<=shift[15:8];
press_count<=press_count+1'b1;
end

end
always @(posedge clk) begin
 if (rst) begin
       state<=IDLE;
       shift_count<=0;
       rdy<=0;
       buffer<=32'b0;
       set<=0;
       end
 else begin
        case (state)
               IDLE: begin
                      if (press_count==4) begin
                      state<=READY;
                      rdy<=1'b1;
                      end
                      else state<=IDLE;
                      end
             READY: begin
                    if (rdy) begin
                    state<=SHIFT;
                    set<=0;
                    rdy<=0;
                    buffer<=shift;
                    end
                    else state<=IDLE;
                    end
             SHIFT: begin
                    if (shift_count<33)begin
                    state<=SHIFT;
                    buffer<=buffer>>1'b1;
```

```
                    shift_count<=shift_count+1;
                    end
                    else begin
                    state<=IDLE;
                    shift_count<=0;
                    set<=1;
                    end
                    end

             endcase
             end
end

      mult multiplier(clk,buffer[0],rdy,rst, data_out);

//lcd_driver lcd(clk,rs,rw,e,d,memaddr,membus);
endmodule
```

2.7.5 LCD Driver

```
module lcd_driver (
      input            clk,
      input                        rst,
      output reg       lcd_rs,
      output reg       lcd_rw,
      output reg       lcd_e,
      output reg [7:4] lcd_d,
      output     [4:0] mem_addr,
      input      [7:0] mem_bus
      );

      parameter        n = 25;
      parameter       j = 6;          // Initialization is slow, runs at clk/2^(j+2)
~95Hz
      parameter         k = 6;             // Writing/seeking is fast, clk/2^(k_2)
~6KHz
      parameter       noop = 6'b010000; // Allows LCD to drive lcd_d, can be safely
written any time

      reg        [n:0] count;
      reg        [5:0] lcd_state;
      reg              init;          // Start in initialization on power on
      reg              row;          // Writing to top or or bottom row

      assign mem_addr = {row, count[k+6:k+3]};

      always @ (posedge clk) begin
      if(rst) begin
      count<=0;
      lcd_state<=noop;
      init<=1;
      row<=0;
      lcd_rs<=0;
      lcd_rw<=0;
      lcd_e<=0;
      lcd_d<=0;
```

```
	end
	else begin
		count <= count + 1;
		if (init) begin                    // initalization
			case (count[j+7:j+2])
				1: lcd_state <= 6'b000010;    // function set
				2: lcd_state <= 6'b000010;
				3: lcd_state <= 6'b001000;
				4: lcd_state <= 6'b000000;    // display on/off control
				5: lcd_state <= 6'b001100;
				6: lcd_state <= 6'b000000;    // display clear
				7: lcd_state <= 6'b000001;
				8: lcd_state <= 6'b000000;    // entry mode set
				9: lcd_state <= 6'b000110;
				10: begin init <= ~init; count <= 0; end
			endcase
	// Write lcd_state to the LCD and turn lcd_e high for the middle half of each
lcd_state
	{lcd_e,lcd_rs,lcd_rw,lcd_d[7:4]} <= {(^count[j+1:j+0] & ~lcd_rw),lcd_state};
	end else begin                     // Continuously update screen from memory
	case (count[k+7:k+2])
	32: lcd_state <= {3'b001,~row,2'b00};  // Move cursor to begining of next line
	33: lcd_state <= 6'b000000;
	34: begin count <= 0; row <= ~row; end // Restart and switch which row is
being written
	default: lcd_state <= {2'b10, ~count[k+2] ? mem_bus[7:4] : mem_bus[3:0]}; //
Pull characters from bus
	endcase
	// Write lcd_state to the LCD and turn lcd_e high for the middle half of each
lcd_state
	{lcd_e,lcd_rs,lcd_rw,lcd_d[7:4]} <= {(^count[k+1:k+0] & ~lcd_rw),lcd_state};
	end
	end
	end
endmodule
```

2.7.6 TestBench

```
module LCD_test;

	// Inputs
	reg clk;
	reg rst;
	reg ctrl;
	reg [7:0] data_in;

	// Outputs
	wire [3:0] d;
	wire rs;
	wire rw;
	wire e;
	wire sf_e;

	// Instantiate the Unit Under Test (UUT)
	IO_w_LCD uut (
		.clk(clk),
```

```
            .rst(rst),
            .ctrl(ctrl),
            .data_in(data_in),
            .d(d),
            .rs(rs),
            .rw(rw),
            .e(e),
            .sf_e(sf_e)
      );
      reg[31:0] data=32'b00000000010101010_1000101011_00001010;
      integer i=0;
      initial begin
            // Initialize Inputs
            data_in = 0;
            ctrl = 0;
            clk = 0;
            rst = 1;

            // Wait 100 ns for global reset to finish
            #40;
     rst=0;
            // Add stimulus here
            for (i=0;i<32;i=i+8) begin
            data_in=data[7:0];
            #30 ctrl=1;
            #30 ctrl=0;
            data=data>>8;
            end
      end
      always #10 clk=~clk;

endmodule
```

3 Results

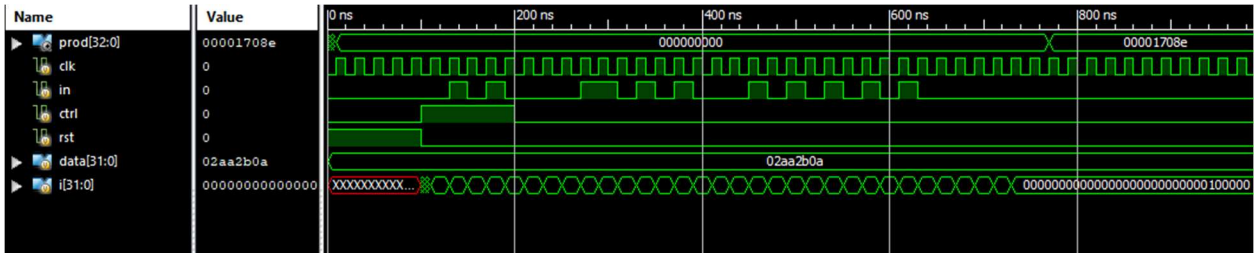

Figure 4 Individual Multiplier Output

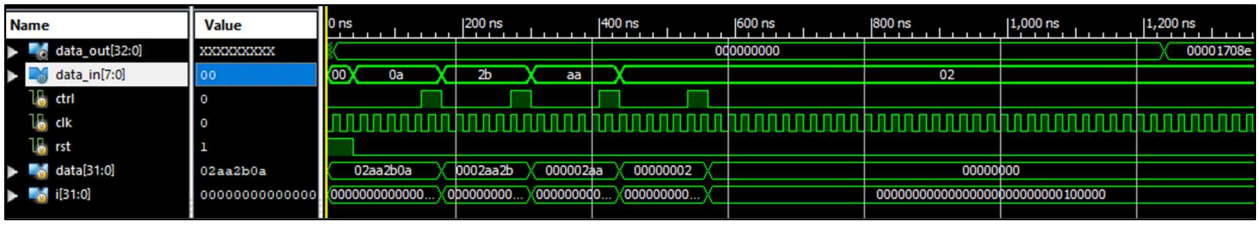

Figure 5 I/O handler Operation

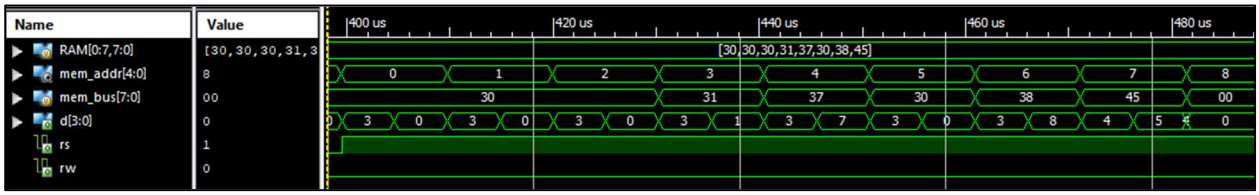

Figure 6 LCD Driver Output

3.1.1 Result Discussion

Multiplier Module was tested individually, it was provided with the serial input of '40'b10101010_1000101011_00001010', the least significant 8 bits represent the size of multiplier and rest out of 40 bits are multiplicand. The 10101010 * 1000101011 operation results in 0x1708e which can be seen in data_out port in figure 4.

The I/O handler deals with receiving data in the form of bytes from either manual entry or UART. The ctrl signal is trigger which is used to pick the byte from input bus. The data is stored in shift register until all 4 bytes are received then this data is passed on to a state machine that saves it in a distributed RAM in ASCII format by concatenating it with appropriate nibble so we may be able to see hexadecimal digits on character LCD. All this operation can be seen in figure 5.

The LCD driver operates on a slow clock and this clock derives a counter, the system is set up in such a ways that on the count value is connected with RAM address bus and RAM data bus is connected with LCD data line, on every cycle LCD driver reads on byte from RAM and splits it into nibbles which are transmitted to LCD one by one. The counter counts till 16 because each hexadecimal digit requires 1 nibble and there are 32 character places on LCD.

4 Discussion

The project implements a serial multiplier with variable operand lengths using Verilog HDL. The design consists of several modules that work together to perform the multiplication operation, handle input/output operations, and display the results on an LCD screen.

The serial multiplier module (`mult`) utilizes an Algorithmic State Machine (ASM) approach to perform the multiplication. It takes inputs such as `clk`, `in`, `ctrl`, and `rst`, and produces the output `prod`. The module includes a state machine (`mult_sm`) that manages the different stages of the multiplication process. The state machine controls the shifting of the operands, the accumulation of partial products, and the final result generation.

The Multiplier State Machine module (`mult_sm`) receives inputs such as `mult_length`, `clk`, `rst`, and `ctrl`, and generates control signals (`shift_count`, `length_bit`, `multiplier_bit`, `multiplicand_bit`) for the serial multiplier module. The state machine transitions through different states based on the control signal `ctrl` and controls the sequence of operations in the serial multiplier module.

The I/O Handler module (`IO_w_LCD`) handles the input/output operations and interfaces with an LCD display. It manages the storage of input data and displays the results on the LCD screen. The module includes a state machine (`state`) that controls the I/O operations and updates the LCD display with the calculated result.

The Parallel to Serial Converter module (`io_module`) converts parallel input data (`data_in`) into serial format for the serial multiplier module. It takes inputs such as `data_in`, `ctrl`, `clk`, `rst`, and produces the output `data_out` and control signal `set`. The module includes a state machine that controls the conversion process, shifting the parallel data, and generating the appropriate control signals for the serial multiplier module.

The LCD Driver module (`lcd_driver`) handles the communication with the LCD display. It receives inputs such as `clk`, `rst`, `rs`, `rw`, `e`, `d`, `mem_addr`, `mem_bus`, and drives the appropriate control signals to the LCD display to write data on it.

The overall working of the project involves taking input data, converting it into serial format, performing the multiplication using the serial multiplier, and displaying the result on the LCD screen. The state machines in each module ensure the proper sequencing and coordination of operations, allowing the system to perform the multiplication accurately and display the result in real-time.
The overall system works by taking input data, converting it into serial format, and feeding it to the serial multiplier module for multiplication. The result is then displayed on an LCD screen using the LCD driver module. The various state machines and control signals ensure the proper sequencing and coordination of the operations within the system.

Note: The code provided lacks some necessary modules and connections to fully understand and analyze its functionality. It's important to have a complete and interconnected design to ensure proper functioning.

5 Conclusion

In this project, we have successfully designed a serial multiplier capable of handling variable operand lengths using an Algorithmic State Machine (ASM) approach. The ASM model allowed us to break down the multiplication operation into a series of states and transitions, accommodating operands of different sizes dynamically. By incorporating initialization, shifting, and addition/subtraction states, we achieved accurate and reliable multiplication results.

The implementation of the serial multiplier was carried out in hardware using a hardware description language (HDL). We verified the functionality and correctness of the design through simulation, ensuring that the circuit produced the expected results for operands of varying lengths.