Homework 4: Optimized Multiplication Circuits on FPGA

EEL3792C: Computer Engineering Design Lab 2

Benjamin Betancourt

Lab Dates: March 10th and 11th, 2025

Date of Submission: March 11th, 2025

Brief Introduction:

This homework seeks to expand the knowledgebase of manual multiplication algorithms and their uses. Both algorithms introduced in this exercise are considerably shorter and easier to implement than the multiplication implemented in the “Shift, Combine, and Arithmetic Operations”. The first operation makes use of addition and subtraction per step of the algorithm. The second operation makes use of shifts. These differences can be seen in the number of cycles required for operation completion.

Verilog Implementation:

The final version of the implementation makes use of a driver level. This driver level is the top level module that handles the input, storage, and passing of input values. The driver also calls the other two modules for the computation of the chosen multiplication algorithm and the passing of the result values to hexadecimal seven segment displays for easy viewing.

Driver Code:

```
module Alg1(input clk, load, aOrb, set, //load runs the multiplier and set is there for setting values
input [7:0] currIn, //32 bit inputs and outputs for testing
output [15:0] result,
output [7:0] disp0, disp1, disp2, disp3);

//registers for storing intermediary values
reg [15:0] a;
reg [15:0] b;

initial begin
a = 0;
b = 0;
end

always @(posedge clk) begin
a = (aOrb == 0 && set == 0)? currIn : a;
b = (aOrb == 1 && set == 0)? currIn : b;
end

mul1 mult(.clk(clk), .load(load), .aIn(a), .bIn(b), .result(result));

hexSevenSegmentDecoder h0(result[3:0],disp0);
hexSevenSegmentDecoder h1(result[7:4],disp1);
hexSevenSegmentDecoder h2(result[11:8],disp2);
hexSevenSegmentDecoder h3(result[15:12],disp3);

endmodule
```

Multiplication algorithm 1:

```
module mul1 (input clk, load,
input [7:0] aIn, bIn,
output [15:0] result);

reg [15:0] s;
reg [15:0] a;
reg [15:0] b;
reg [15:0] rout;

initial begin //set initial state
    s = 0;
    a = 0;
    b = 0;
    rout = 0;
end
always @(posedge clk) begin //posedge to prevent clock related computation errors
    case (load)
        1'b1 : begin
            if (b==0) begin
                rout = s;
            end else begin
                s = s+a;
                b = b-1;
            end
        end 1'b0 : begin
            s = 0;
            a = aIn;
            b = bIn;
        end
    endcase
end
assign result = rout;
endmodule
```

Hexadecimal seven segment display decoder:

```
module hexSevenSegmentDecoder(input [3:0] in,
output [7:0] out);
reg [7:0] wout;
always@ (*) begin
    case(in)
        4'h0: wout = 8'b11000000;
        4'h1: wout = 8'b11111001;
        4'h2: wout = 8'b10100100;
        4'h3: wout = 8'b10110000;
        4'h4: wout = 8'b10011001;
        4'h5: wout = 8'b10010010;
        4'h6: wout = 8'b10000010;
        4'h7: wout = 8'b11111000;
        4'h8: wout = 8'b10000000;
        4'h9: wout = 8'b10010000;
        4'ha: wout = 8'b10001000;
        4'hb: wout = 8'b10000011;
        4'hc: wout = 8'b11000110;
        4'hd: wout = 8'b10100001;
        4'he: wout = 8'b10000110;
        4'hf: wout = 8'b10001110;
        default: wout = 8'b10000000;
    endcase
end
assign out = wout;
endmodule
```

The second algorithm makes use of a similar driver level. This module accomplishes the same tasks as the previous algorithm. The Verilog implementation of the second algorithm program is below in parts. The hexadecimal seven segment display decoder was the same between both programs so it will not be shown twice.

Driver code:

```
module Alg2(input clk, load, aOrb, set, //load runs the multiplier and set is there for setting values
input [7:0] currIn,
output [15:0] result,
output [7:0] disp0, disp1, disp2, disp3);
reg [15:0] a;
reg [15:0] b;
initial begin
    a = 0;
    b = 0;
    //result = 0;
end

always @(posedge clk) begin

    a = (aOrb == 0 && set == 0)? currIn : a;
    b = (aOrb == 1 && set == 0)? currIn : b;

end
mul2 mult(.clk(clk), .load(load), .aIn(a), .bIn(b), .result(result));

hexSevenSegmentDecoder h0(result[3:0],disp0);
hexSevenSegmentDecoder h1(result[7:4],disp1);
hexSevenSegmentDecoder h2(result[11:8],disp2);
hexSevenSegmentDecoder h3(result[15:12],disp3);
endmodule
```

Multiplication algorithm 2:

```
module mul2 (input clk, load,
input [7:0] aIn, bIn,
output reg [15:0] result);
reg [15:0] s;
reg [15:0] c;
reg [15:0] a;
reg [15:0] b;

initial begin //set initial state
    s = 0;
    c = 8;
    a = 0;
    b = 0;
    result = 0;
end

always @(posedge clk) begin //posedge to prevent clock related computation errors
    case (load)
        1'b1 : begin
            if (c==0) begin
                result = s;
            end else begin
                if (b[0] == 1) begin
                    s = s+a;
                end else begin
                    s = s;
                end
                a = a<<1;
                b = b>>1;
                c=c-1;
            end
        end 1'b0 : begin
            s = 0;
            c = 8;
            a = aIn;
            b = bIn;
        end
    endcase
end
endmodule
```

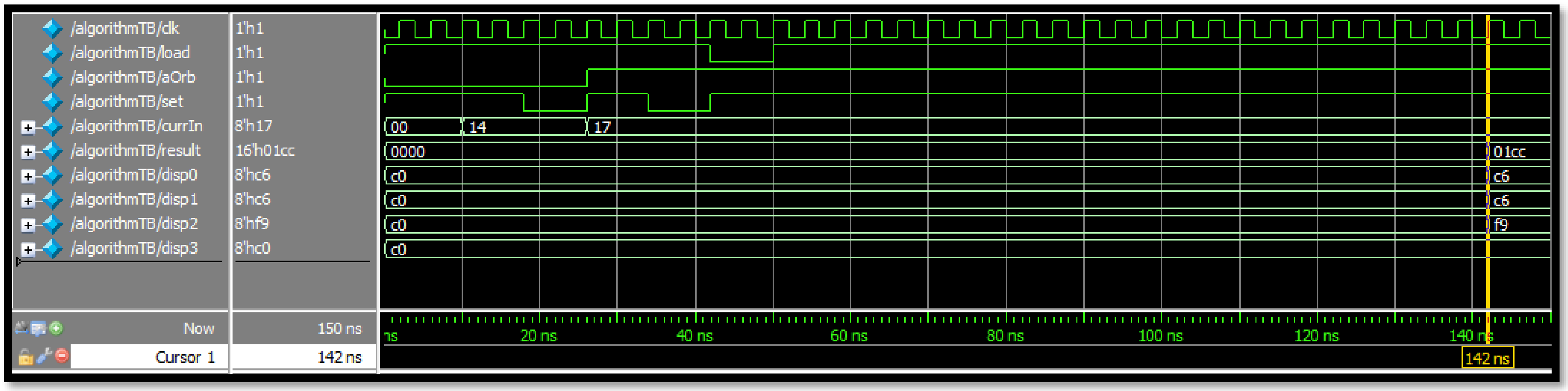
Algorithm results and simulations:

Both test benches make use of the same code with the only difference being the called algorithm. This is to allow for direct comparison of code execution times in nanoseconds and clock cycles to be drawn between the two algorithms. The test benches for both can be seen below alongside their resulting waveforms.

Testbench algorithm 1:

```
module algorithmTB();


reg clk, load, aOrb, set; //default states, load and set = 1, aOrb = 0
reg [7:0] currIn;

wire [15:0] result;
wire [7:0] disp0, disp1, disp2, disp3;

Alg1 DUT(clk, load, aOrb, set, currIn, result, disp0, disp1, disp2, disp3);

always begin
    #2 clk=~clk;
end

initial begin
    clk = 0; load = 1; set = 1; aOrb = 0; currIn = 0; #10;//set defaults

    load = 1; set = 1; aOrb = 0; currIn = 20; #8; //putting the value 20 into A
    load = 1; set = 0; aOrb = 0; currIn = 20; #8; //button press to set 20 as A

    load = 1; set = 1; aOrb = 1; currIn = 23; #8; //putting the value 23 into B
    load = 1; set = 0; aOrb = 1; currIn = 23; #8; //button press and switch set to set B as 23

    load = 0; set = 1; aOrb = 1; currIn = 23; #8; //executing multiplication instruction
    load = 1; set = 1; aOrb = 1; currIn = 23; #100; //waiting for instruction to complete
    $stop; //end simulation with completed instruction

end
endmodule
```

Testbench 1 waveforms:

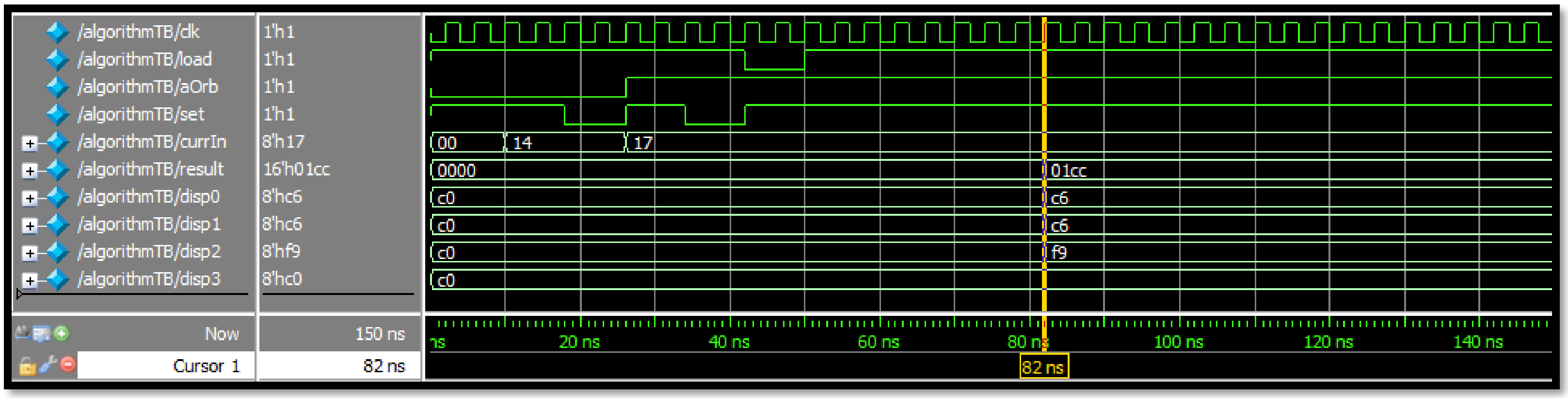


```
/algorithmTB/clk      1'h1
/algorithmTB/load     1'h1
/algorithmTB/aOrb     1'h1
/algorithmTB/set      1'h1
/algorithmTB/currIn   8'h17
/algorithmTB/result   16'h01cc
/algorithmTB/disp0    8'hc6
/algorithmTB/disp1    8'hc6
/algorithmTB/disp2    8'hf9
/algorithmTB/disp3    8'hc0

Now       150 ns
Cursor 1  142 ns
```

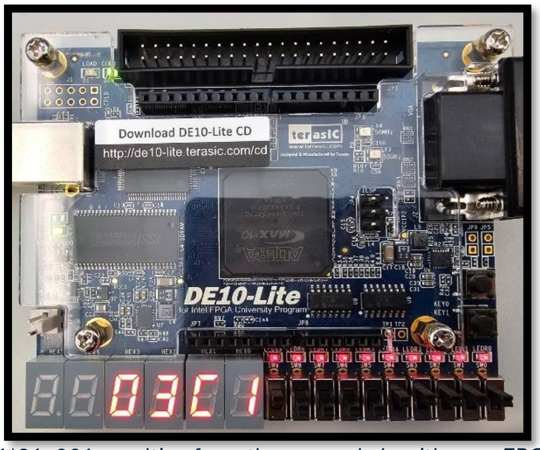
Testbench algorithm 1:

```
module algorithmTB();
reg clk, load, aOrb, set; //default states, load and set = 1, aOrb = 0
reg [7:0] currIn;
wire [15:0] result;
wire [7:0] disp0, disp1, disp2, disp3;

Alg2 DUT(clk, load, aOrb, set, currIn, result, disp0, disp1, disp2, disp3);

always begin
    #2 clk=~clk;
End

initial begin
    clk = 0; load = 1; set = 1; aOrb = 0; currIn = 0; #10;//set defaults

    load = 1; set = 1; aOrb = 0; currIn = 20; #8; //putting the value 20 into A
    load = 1; set = 0; aOrb = 0; currIn = 20; #8; //button press to set 20 as A

    load = 1; set = 1; aOrb = 1; currIn = 23; #8; //putting the value 23 into B
    load = 1; set = 0; aOrb = 1; currIn = 23; #8; //button press and switch set to set B as 23

    load = 0; set = 1; aOrb = 1; currIn = 23; #8; //executing multiplication instruction
    load = 1; set = 1; aOrb = 1; currIn = 23; #100; //waiting for instruction to complete
    $stop; //end simulation with completed instruction
end
endmodule
```

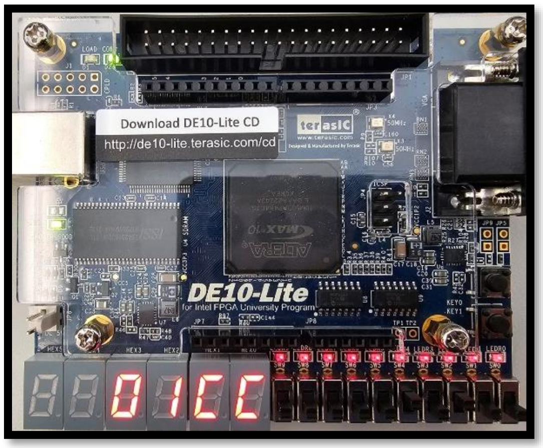
Testbench 1 waveforms:



```
/algorithmTB/clk      1'h1
/algorithmTB/load     1'h1
/algorithmTB/aOrb     1'h1
/algorithmTB/set      1'h1
/algorithmTB/currIn   8'h17      00 | 14 | 17
/algorithmTB/result   16'h01cc   0000 | 01cc
/algorithmTB/disp0    8'hc6      c0 | c6
/algorithmTB/disp1    8'hc6      c0 | c6
/algorithmTB/disp2    8'hf9      c0 | f9
/algorithmTB/disp3    8'hc0      c0

Now       150 ns
Cursor 1  82 ns
```



*31*31=961 resulting from the second algorithm on FPGA*



*23*20=460 resulting from first algorithm on FPGA*

Conclusions:

Both algorithms are a large improvement in execution time and complexity from previously used multiplication algorithms. These algorithms are directly scalable with bus size. The previous one would increase in size exponentially as bus lanes are added to the inputs and result. However, both algorithms demonstrated in this homework assignment have their own strengths and weaknesses. The primary strength of algorithm 1 is the ease of implementation and code size. The first algorithm only makes use of arithmetic in its computations. This means that the code size is smaller than that of algorithm 2. However, this comes at the expense of computation time. When compared to algorithm 2, the execution of the algorithm takes 60 more nanoseconds than that of algorithm 1. This is a 74% increase in execution time! The second algorithm however makes use of bit shifting alongside the arithmetic seen in algorithm 1. This shifting requires additional consideration for the least and most significant bits which increases potential bus sizes. The code space is also larger and slightly more complex than that of algorithm 1. However, the massively reduced computation time more than makes up for the slightly increased complexity. Therefore, from the required computations for this homework, algorithm 2 seems to be the better option to employ in a future project out of the two.