CND 101
Introduction to Digital Design

Approximate ALU

Date: 19/12/2023

Made by

Mohamed Elsayed Abouelhamd Mohamed
V23010295

Supervised By

Dr. Omar Eldash

Outlines:

I. Introduction:

Approximate computing is a paradigm that aims to strike a balance between computational accuracy and resource efficiency. Traditional computing systems typically prioritize exactness and guarantee precise results. However, many applications, such as image and signal processing, machine learning, and multimedia, can tolerate a certain level of imprecision without significantly affecting the overall quality or usefulness of the output. This observation has led to the emergence of approximate computing as a promising approach to improve performance and energy efficiency in various domains.

a. Definition of Approximate Computing:

Approximate computing is a computing methodology that allows for intentional imprecision in the execution of computations. Instead of insisting on exact results, approximate computing techniques leverage the inherent resilience of certain applications to inaccuracies and exploit the trade-off between accuracy and computational resources. By relaxing the requirement for precision, approximate computing can reduce the energy consumption, execution time, and hardware complexity of computations.

The key idea behind approximate computing is to exploit the fact that not all computations need to be performed with high precision. By introducing controlled errors or approximations into the computation, significant gains in performance and energy efficiency can be achieved. The extent of approximation can be controlled based on the specific application requirements and the acceptable level of error.

b. Arithmetic Circuits in Approximate Computing:

Arithmetic circuits play a crucial role in approximate computing as they are responsible for executing numerical computations. These circuits are designed to perform operations such as addition, subtraction, multiplication, and division on binary numbers. In the context of approximate computing, the design of arithmetic circuits is optimized to introduce controlled errors while minimizing the impact on the overall quality of the output.

Various techniques are employed in approximate arithmetic circuit design. One common approach is to reduce the precision of the operands or intermediate results. This reduction in precision reduces the computational complexity, memory requirements, and power consumption of the arithmetic circuits. Another technique involves the use of approximate operators. These operators trade-off accuracy for improved efficiency.

For instance, approximate multipliers can be designed to sacrifice a certain level of precision in exchange for faster operation or reduced area requirements. Similarly, approximate adders can be used to perform fast but less precise additions. Therefore, approximate computing techniques can exploit the inherent redundancy in arithmetic operations to achieve efficiency gains.

II. Architecture and Analysis

a. Multiplier based on inaccurate 2x2 multiplier:

This multiplier is built by the idea of using multiple number of inaccurate 2x2 multiplier that is built in specific architecture.

The architecture of our 2x2 inaccurate multiplier is shown in figure (1). This multiplier is distinct from the other multiplier since the result of multiplication is only three bits in addition to smaller area and consequently less power than the ordinary multiplier. This is an example of how the sacrifice of accurate results can yield to smaller area in addition to smaller power.

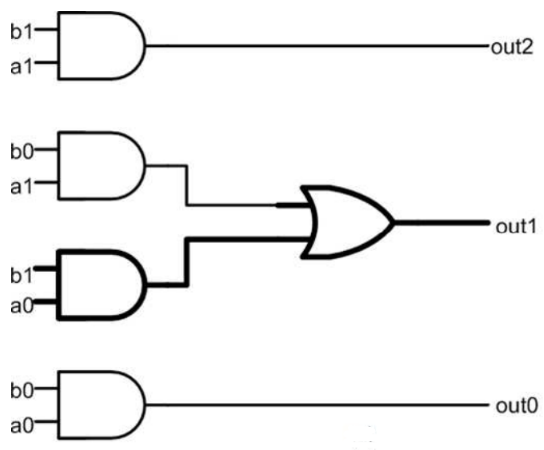


Figure 1 Inaccurate 2x2 multiplier

To quantify the error parameters of the proposed architecture, the error cases must be known. As shown in table (1) the output of the multiplier is listed with the error highlighted. By representing the output of 3*3 using three bits (111) instead of the usual four (1001), we can significantly reduce the complexity of the circuit.

Table 1 Multiplier output

A_1A_0 \ B_1B_0	00	01	11	10
00	000	000	000	000
01	000	001	011	010
11	000	011	$\boxed{111}$	110
10	000	010	110	100

The resulting circuit has an output that is correct for fifteen out of the sixteen possible inputs with a probability of $\frac{1}{16}$ (assuming a uniform input distribution). The inaccurate version has close to half the area of the accurate version in addition to shorter and faster critical path and less interconnect.

Larger multipliers can easily be built using smaller multiplier blocks. We build multipliers of higher bit width by using the inaccurate 2x2 block to produce partial products and then adding the shifted versions of the partial products. Figure (2) shows how a single 4x4 multiplier is built out of four 2x2 blocks, where A_H, X_H and A_L, X_L are the upper and lower two bits of inputs A, X respectively. Such an approach can lead to sub-optimal results. But the simplicity of the inaccurate 2x2 building block means it does not suffer from this restriction. As a result, larger multiplier blocks can be built out of the 2x2 building block, and still perform better in terms of power and area as compared to accurate architectures. It is important to note that in the larger multipliers, we introduce inaccuracy via the partial products, the adders remain accurate.

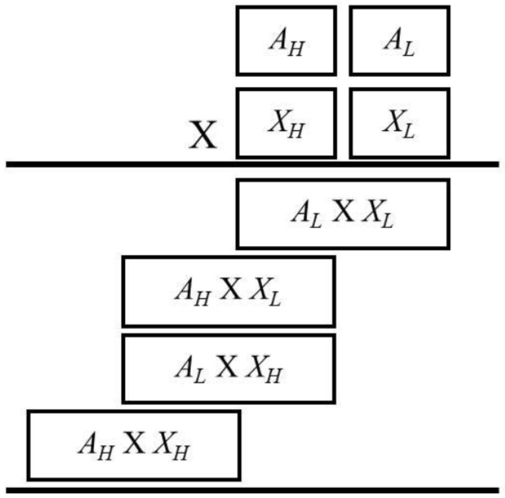
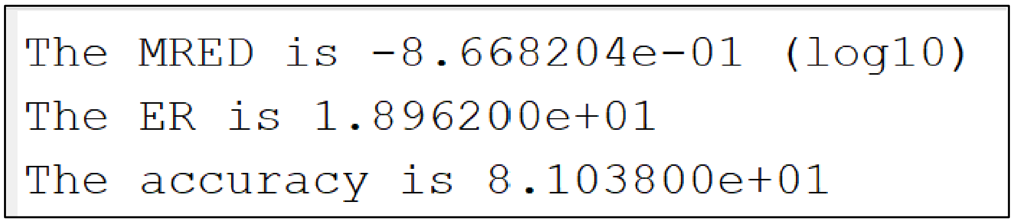


Figure 2 Building larger multipliers

To introduce the error characteristic of the design, we will calculate two quantities error rate (ER) and mean relative error distance (MRED). ER is the probability of producing an incorrect result. MRED is the average value of all possible relative error distances (REDs) usually represented in logarithmic scale. RED is the ratio of the difference between the approximate and accurate result to the accurate result. A MATLAB code is used to calculate MRED and ER for the multiplier. As shown in figure (3) the error characteristics of the 4x4 multiplier.

➢ MATLAB code:

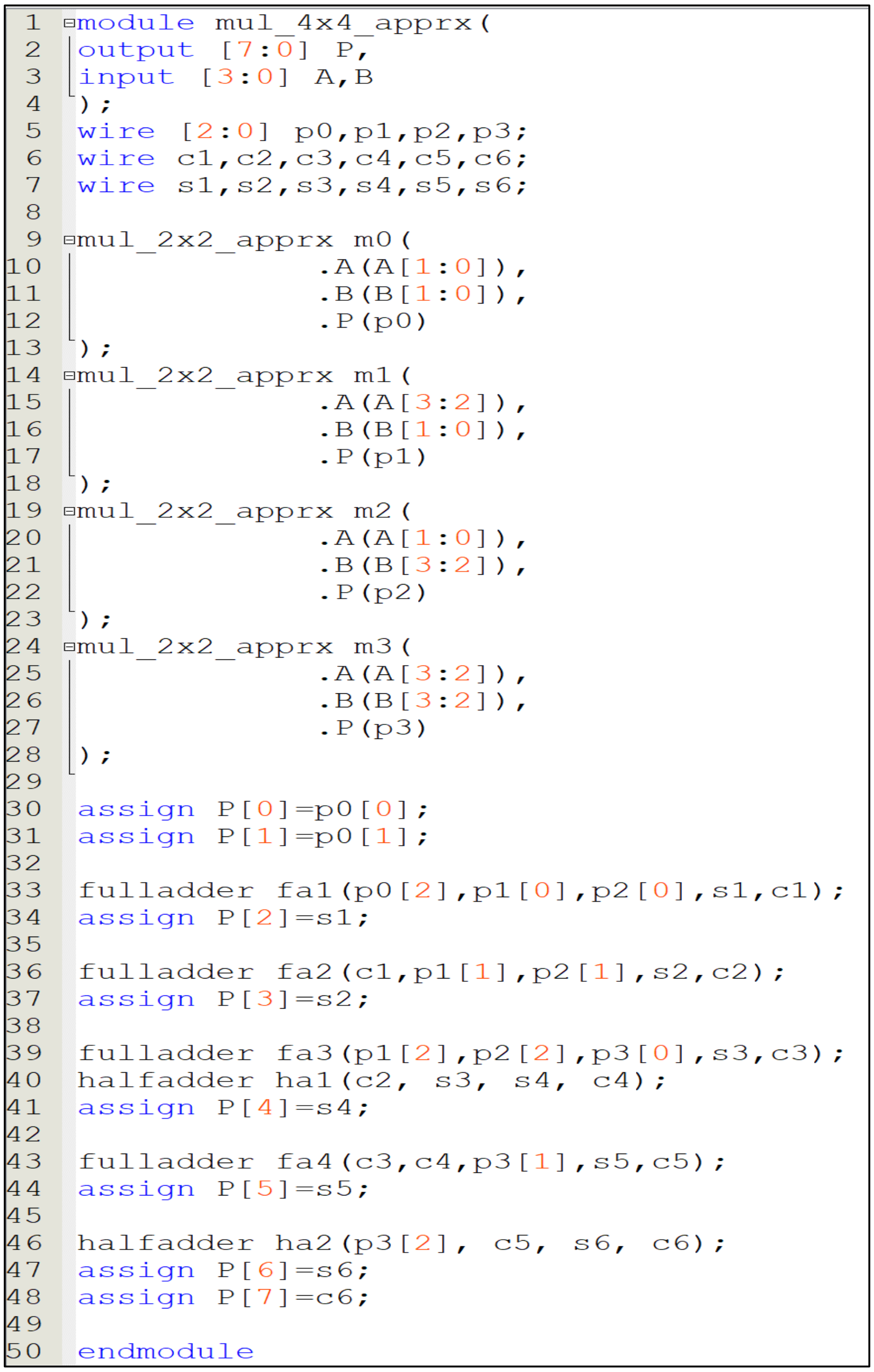
```
ER=0;
RED=[];
n=4;
samples=10^5;
i= randi([0,2^n-1],1,samples);
j= randi([0,2^n-1],1,samples);
    for l=1:1:samples
        a_low= bitget(i(l),2:-1:1);
        b_low= bitget(j(l),2:-1:1);
        a_high= bitget(i(l),4:-1:3);
        b_high= bitget(j(l),4:-1:3);
        par0= par_pro(a_low,b_low);
        par1= par_pro(a_high,b_low);
        par2= par_pro(a_low,b_high);
        par3= par_pro(a_high,b_high);
        par0_num= bin2dec(par0);
        par1_num= bin2dec(strcat(par1,'00'));
        par2_num= bin2dec(strcat(par2,'00'));
        par3_num= bin2dec(strcat(par3,'0000'));
        mul_tot= par0_num + par1_num + par2_num + par3_num;
        if mul_tot ~= i(l)*j(l)
            ER=ER+1;
            RED(ER)=abs(mul_tot-(i(l)*j(l)))/(i(l)*j(l));
        end
    end
y=sprintf('The MRED is %d (log10)',log10(mean(RED)));
x=sprintf('The ER is %d',ER*100/samples);
z=sprintf('The accuracy is %d',100-(ER*100/samples));
disp(y); disp(x); disp(z);

function p= par_pro(a,b)
    x1= a(1) & b(1);
    x2= a(2) & b(1) | a(1) & b(2);
    x3= a(2) & b(2);
    x=[x1 x2 x3];
    p=strrep(num2str(x),' ','');
end
```



```
The MRED is -8.668204e-01 (log10)
The ER is 1.896200e+01
The accuracy is 8.103800e+01
```

Figure 3 Error characteristics of multiplier

➢ Verilog code:



```
module mul_4x4_apprx(
output  [7:0]  P,
input  [3:0]  A,B
);
wire  [2:0]  p0,p1,p2,p3;
wire c1,c2,c3,c4,c5,c6;
wire s1,s2,s3,s4,s5,s6;

mul_2x2_apprx m0(
                    .A(A[1:0]),
                    .B(B[1:0]),
                    .P(p0)
);
mul_2x2_apprx m1(
                    .A(A[3:2]),
                    .B(B[1:0]),
                    .P(p1)
);
mul_2x2_apprx m2(
                    .A(A[1:0]),
                    .B(B[3:2]),
                    .P(p2)
);
mul_2x2_apprx m3(
                    .A(A[3:2]),
                    .B(B[3:2]),
                    .P(p3)
);

assign  P[0]=p0[0];
assign  P[1]=p0[1];

fulladder  fa1(p0[2],p1[0],p2[0],s1,c1);
assign  P[2]=s1;

fulladder  fa2(c1,p1[1],p2[1],s2,c2);
assign  P[3]=s2;

fulladder  fa3(p1[2],p2[2],p3[0],s3,c3);
halfadder ha1(c2,  s3,  s4,  c4);
assign  P[4]=s4;

fulladder  fa4(c3,c4,p3[1],s5,c5);
assign  P[5]=s5;

halfadder  ha2(p3[2],  c5,  s6,  c6);
assign  P[6]=s6;
assign  P[7]=c6;

endmodule
```

Figure 4 Verilog code of 4x4 approximate multiplier

➢ The schematic:

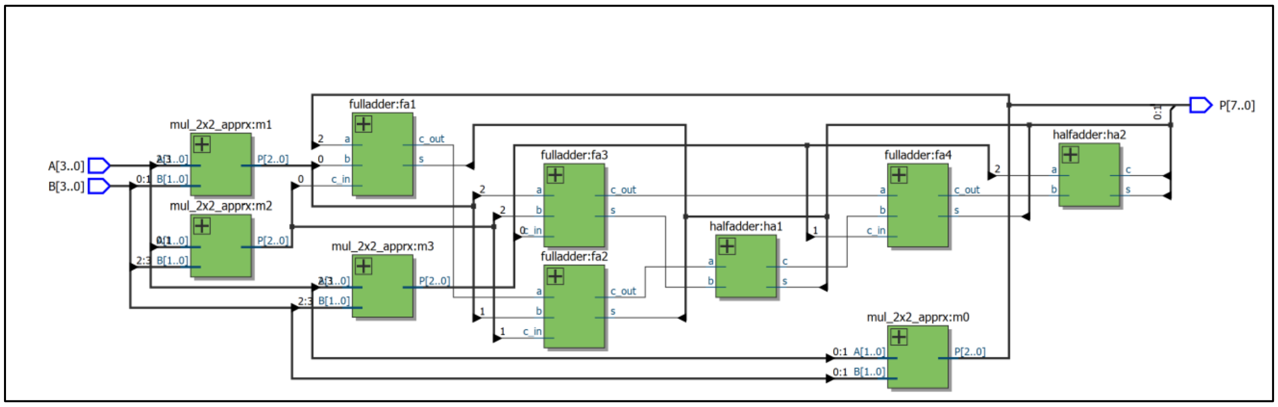


Figure 5 The schematic of 4x4 approximate multiplier

As shown in figure (4,5) the methodology of designing 4x4 approximate multiplier is by using the 2x2 building block to generate the partial product, then using a sequence of half adders and full adders to generate the sum of partial products depending on the required shift for each one. The result of this addition is the multiplication of two 4-bit numbers but with an approximate result.

Table 2 Comparison between resources usages

Exact 4x4 multiplier	Approximate 4x4 multiplier
31 logic elements	23 logic elements

To validate the reduction of area of a multiplier, a comparison was made between our design of approximate multiplier and an exact multiplier. This comparison was done and simulated on cyclone III FPGA [EP3C120F780C7]. As shown in table (2) the resources usage of our approximate multiplier is less than the exact multiplier. Therefore, our goal is reached which is reducing the accuracy to reduce the area.

b. Lower-Part-OR adder (LOA):

In a traditional binary adder, the addition of two binary numbers is performed by a series of full adders. However, in LOA, the lower part of the adder uses OR gates instead of full adders to calculate the sum. This approach can be more area-efficient compared to using full adders for all stages. LOA consists of two parts as shown in figure (6) the first part of the adder deals with least significant bits (LSBs) and the second part deals with the most significant bits (MSBs). The first part is constructed from OR gates to calculate the sum and an AND gate at the last bit to generate the carry for the second part. The second part is a precise ripple carry adder that takes the carry from the previous two bits.

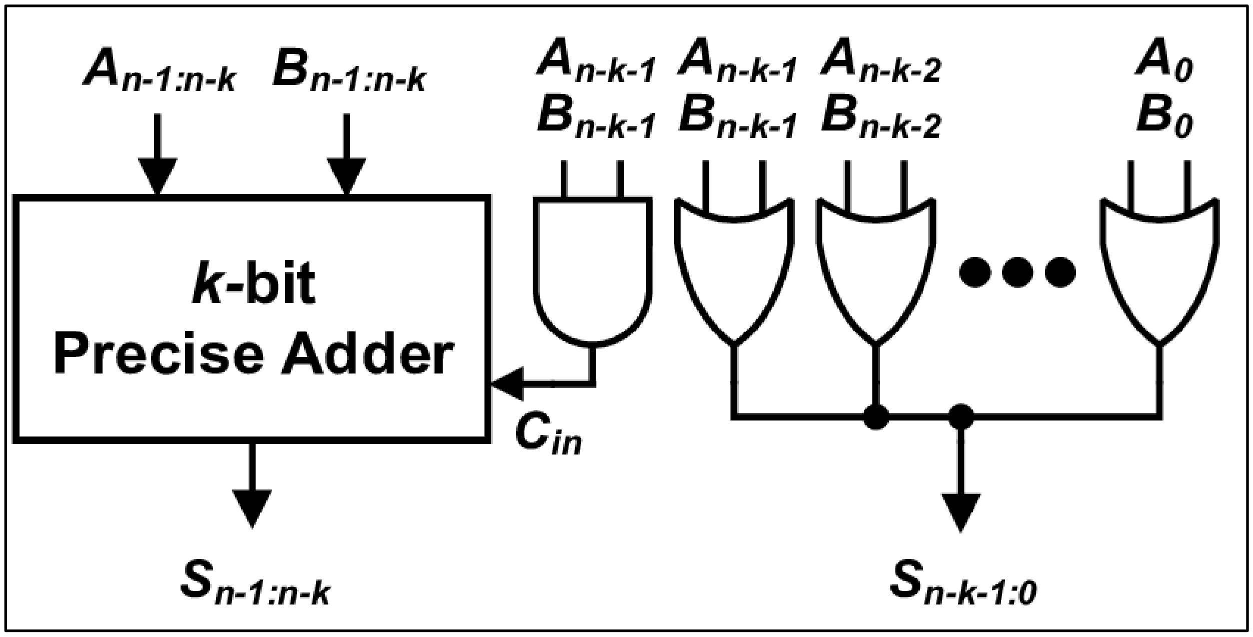


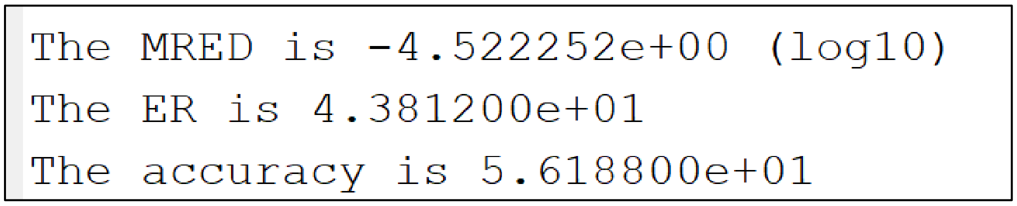
Figure 6 Lower-part-OR adder

Now, we want to quantify the errors of this adder. This adder replaces XOR gates with OR gates at the LSBs. Therefore, the error appears when the input bits for the OR gate are both ones only. Thus, each OR gate has an error of $\frac{1}{4}$ and consequently the total error of the LOA is dependent on number of OR gates $(n-k)$. Using this information, we can deduce the ER. $ER = 1 - (1 - \frac{1}{4})^{n-k}$.

A MATLAB code is used to calculate MRED and ER for the LOA. As shown in figure (7) the error characteristics of the 2-LOA, where 2 is the number of OR gates.

- **MATLAB code:**

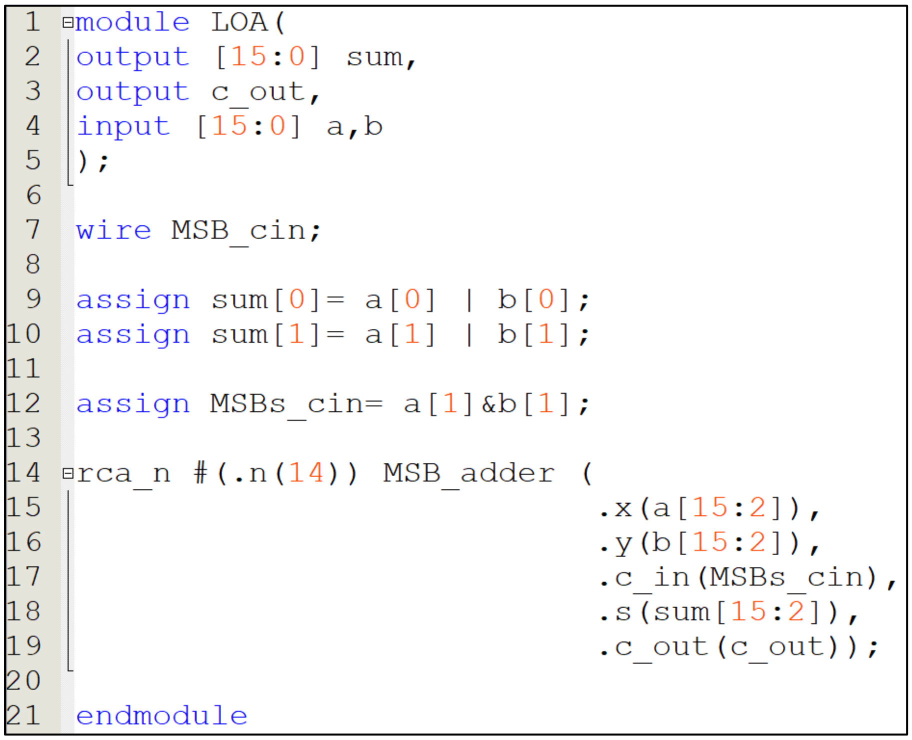
```
ER=0;
RED=[];
n=16;
m=2;
samples=10^5;
i= randi([0,2^n-1],1,samples);
j= randi([0,2^n-1],1,samples);
lsb_zeros= repmat('0', 1, m);
    for l=1:1:samples
        a= dec2bin(i(l),n);
        b= dec2bin(j(l),n);
        a_low= bitget(i(l),m:-1:1);
        b_low= bitget(j(l),m:-1:1);
        a_high= a(1:n-m);
        b_high= b(1:n-m);
        a_num= bin2dec(strcat(a_high,lsb_zeros));
        b_num= bin2dec(strcat(b_high,lsb_zeros));
        s_low= bitor(a_low,b_low);
        carry_in =(2^m)*double((a_low(1)&b_low(1)));
        s_low_num= bin2dec((strrep(num2str(s_low),' ','')));
        s_high_num = a_num +b_num;
        s_tot= s_high_num + s_low_num + carry_in;
        if s_tot ~= i(l)+j(l)
            ER=ER+1;
            RED(ER)=abs(s_tot-(i(l)+j(l)))/(i(l)+j(l));
        end
    end
y=sprintf('The MRED is %d (log10)',log10(mean(RED)));
x=sprintf('The ER is %d',ER*100/samples);
z=sprintf('The accuracy is %d',100-(ER*100/samples));
disp(y); disp(x); disp(z);
```



```
The MRED is -4.522252e+00 (log10)
The ER is 4.381200e+01
The accuracy is 5.618800e+01
```

Figure 7 Error characteristics of 2-LOA

➢ Verilog code:



```
module LOA(
output [15:0] sum,
output c_out,
input [15:0] a,b
);

wire MSB_cin;

assign sum[0]= a[0] | b[0];
assign sum[1]= a[1] | b[1];

assign MSBs_cin= a[1]&b[1];

rca_n #(.n(14)) MSB_adder (
                             .x(a[15:2]),
                             .y(b[15:2]),
                             .c_in(MSBs_cin),
                             .s(sum[15:2]),
                             .c_out(c_out));

endmodule
```

Figure 8 Verilog code of 2-LOA

➢ The schematic:

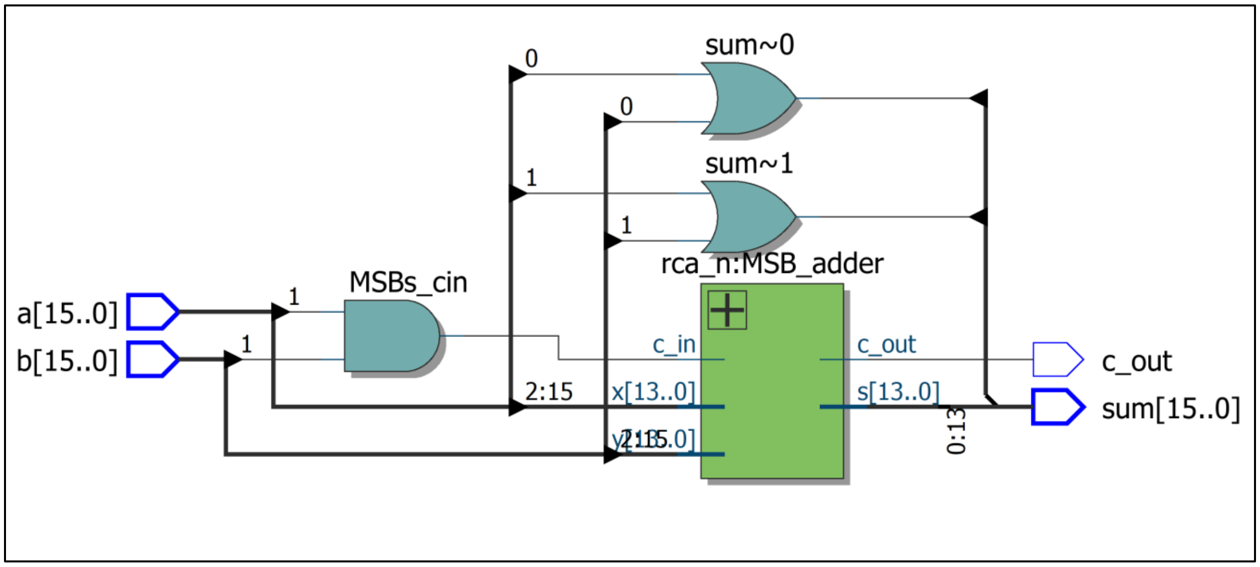


Figure 9 The schematic of 2-LOA

As shown in figure (8,9) the 2 LSBs are applied to OR gates rather than full adders and the rest of bits are applied to an ordinary ripple carry adder. Its first input carry is generated from the previous 2 bits by an AND gate.

Table 3 Comparison between resources usages

Exact ripple carry adder	2-LOA
32 logic elements	30 logic elements

To validate the area reduction of an adder, a comparison was made between 16-bits approximate LOA and a 16-bits exact ripple carry adder. This comparison was done and simulated on cyclone III FPGA [EP3C120F780C7]. As shown in table (3) the resources usage of our approximate 2-LOA is less than the exact ripple carry adder. Therefore, our goal is reached which is reducing the accuracy to reduce the area.

We noticed that the reduction is not as big as the multiplier, this is because of the required accuracy of the adder. Increasing the number of OR gates will reduce accuracy exponentially. We can increase the accuracy of the adder by using XOR gates rather than OR gates, this will make the accuracy increase to 75% and uses the same number of logic elements (LUTs).

On the other hand, LOA has a great advantage which is the error distance between the numbers is not big. The result is almost the same. This is very predictable because errors always happen in the LSBs.

c. Approximate ALU:

An ALU (Arithmetic Logic Unit) is a digital circuit within a computer processor that performs arithmetic and logical operations on binary data such as addition, multiplication, logical AND, OR, XOR, and more.

On the other hand, an Approximate ALU is a variation of the traditional ALU that is designed to provide approximate results rather than exact results aiming to trade off accuracy for benefits such as improved performance, reduced power consumption, or lower hardware complexity. Approximate ALUs can achieve performance improvements or energy savings compared to traditional ALUs, particularly in scenarios where exact precision is not critical.

The main difference between an ALU and an Approximate ALU lies in the way they handle computations and the level of accuracy they provide. While a traditional ALU is designed to deliver precise and deterministic results, an Approximate ALU uses different techniques to introduce controlled errors in the calculations. These errors are usually within acceptable bounds and can be compensated or corrected through error mitigation techniques.

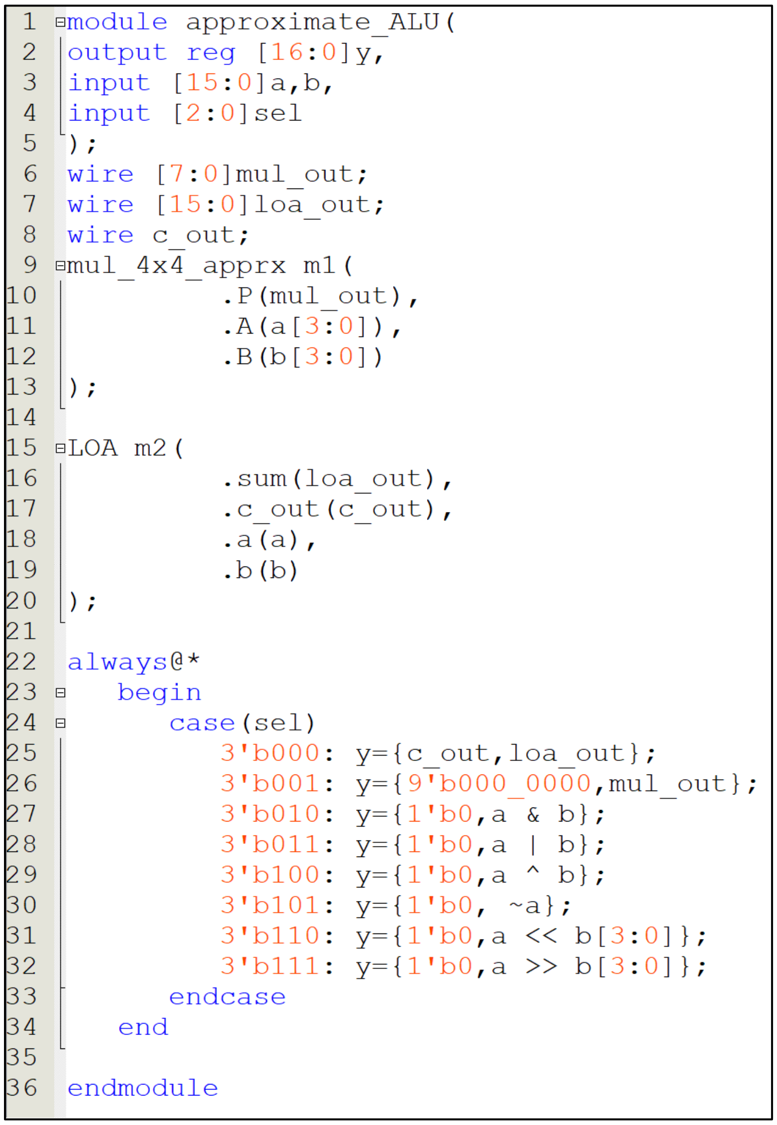
Our proposed architecture for approximate ALU is as follows, using the predesigned approximate multiplier and approximate adder as an arithmetic unit of the ALU in addition to logical and shift operation.

Table 4 Opcode of Approximate ALU

000	A + B
001	A [3:0] * B [3:0]
010	A & B
011	A | B
100	A ^ B
101	~A
110	A << B [3:0]
111	A >> B [3:0]

As shown in table (4), there are 8 operations that can be performed by our ALU. In general, the ALU takes two inputs, A and B, 16-bits each and a 3-bit selector. The first two operations are using the predesigned approximate adder and approximate multiplier and gives an approximate result. Since our multiplier is only 4-bits, we designed the ALU to take the 4 least significant bits of each operand. Also, Regarding the shift's operations, we designed the ALU to use only the 4 least significant bits of B to shift A either right or left depending on the selector.

- **Verilog code:**



```
module approximate_ALU(
output reg [16:0]y,
input [15:0]a,b,
input [2:0]sel
);
wire [7:0]mul_out;
wire [15:0]loa_out;
wire c_out;
mul_4x4_apprx m1(
         .P(mul_out),
         .A(a[3:0]),
         .B(b[3:0])
);

LOA m2(
         .sum(loa_out),
         .c_out(c_out),
         .a(a),
         .b(b)
);

always@*
   begin
      case(sel)
         3'b000: y={c_out,loa_out};
         3'b001: y={9'b000_0000,mul_out};
         3'b010: y={1'b0,a & b};
         3'b011: y={1'b0,a | b};
         3'b100: y={1'b0,a ^ b};
         3'b101: y={1'b0, ~a};
         3'b110: y={1'b0,a << b[3:0]};
         3'b111: y={1'b0,a >> b[3:0]};
      endcase
   end

endmodule
```

Figure 10 Verilog code of approximate ALU

- **The schematic:**

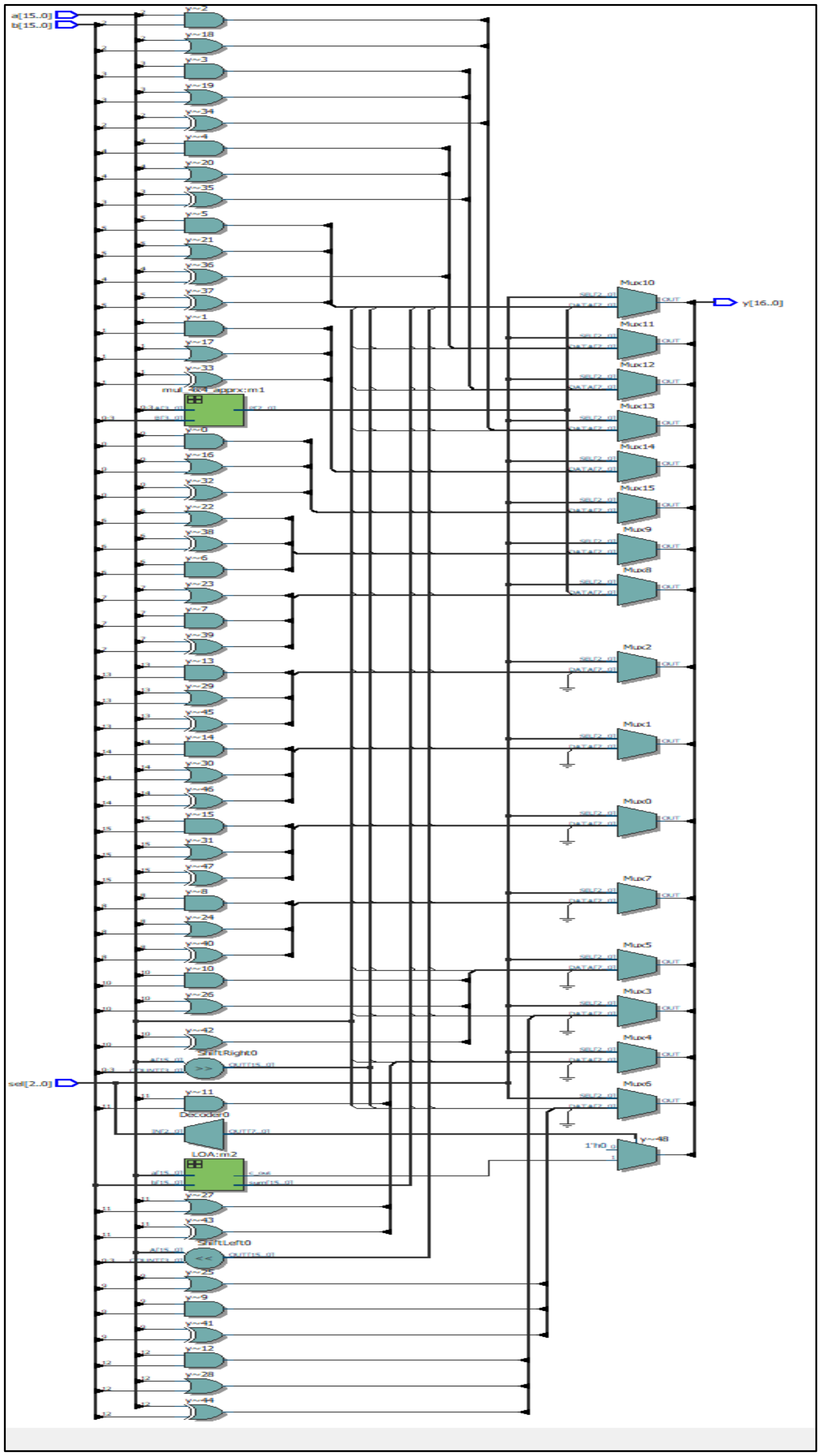


Figure 11 Approximate ALU schematic

Table 5 Area and Power of Approximate ALU

Resources usage	Thermal Power Dissipation
256 logic elements	120.26 mW

After designing the approximate ALU, its features must be determined. The most important features are the resources usage from the FPGA, the thermal power dissipation, and the propagation delay.

In table (5), two of these features are determined on cyclone III FPGA [EP3C120F780C7]. As seen, the used resources are only combinational as there are not any type of sequential logic or flip-flops.

Moreover, thermal power dissipation is generated from Core Static Thermal Power Dissipation and I/O Thermal Power Dissipation with values of 99.07 mW and 21.19 mW respectively. There is no dynamic thermal power dissipation because the module is pure combinational.

The propagation delay here is not important as the module is not included in a sequential environment or another sequential module. There are a lot of propagation delays values since we have three inputs with 35 input bins. Each input bin has its propagation delay value.

d. Finite state machine correction for the ALU:

The second goal for us is to correct the errors of the approximate adder and approximate multiplier since their errors are known and determined. One method or technique for that correction is to put the approximate ALU in a finite state machine (FSM). The objective of this method is to use the rest of ALU operations to correct the errors through multiple clock cycles. This method will decrease the error of the ALU, but this decrease will affect the time to complete either the addition or subtraction. The designed FSM for this approximate ALU corrects only some cases not all, since the rest of error cases will take many clock cycles to be corrected and consequently introduce a lot of delay to the system. After using this FSM, the ER of approximate adders becomes 6.25% and the ER of approximate multiplier becomes 6.25% also.

The error in the adder happens when the inputs to the first OR gate are (1,1) and, at the same time, the inputs to the second OR gate are (0,0). On the other hand, the error in the multiplier happens when the addition uses a number that has bits in the first two positions because we use the approximate adder to correct the multiplication.

Table 6 The area of FSM

Total combinational functions	Dedicated logic registers
334	62

Table 7 The power dissipated.

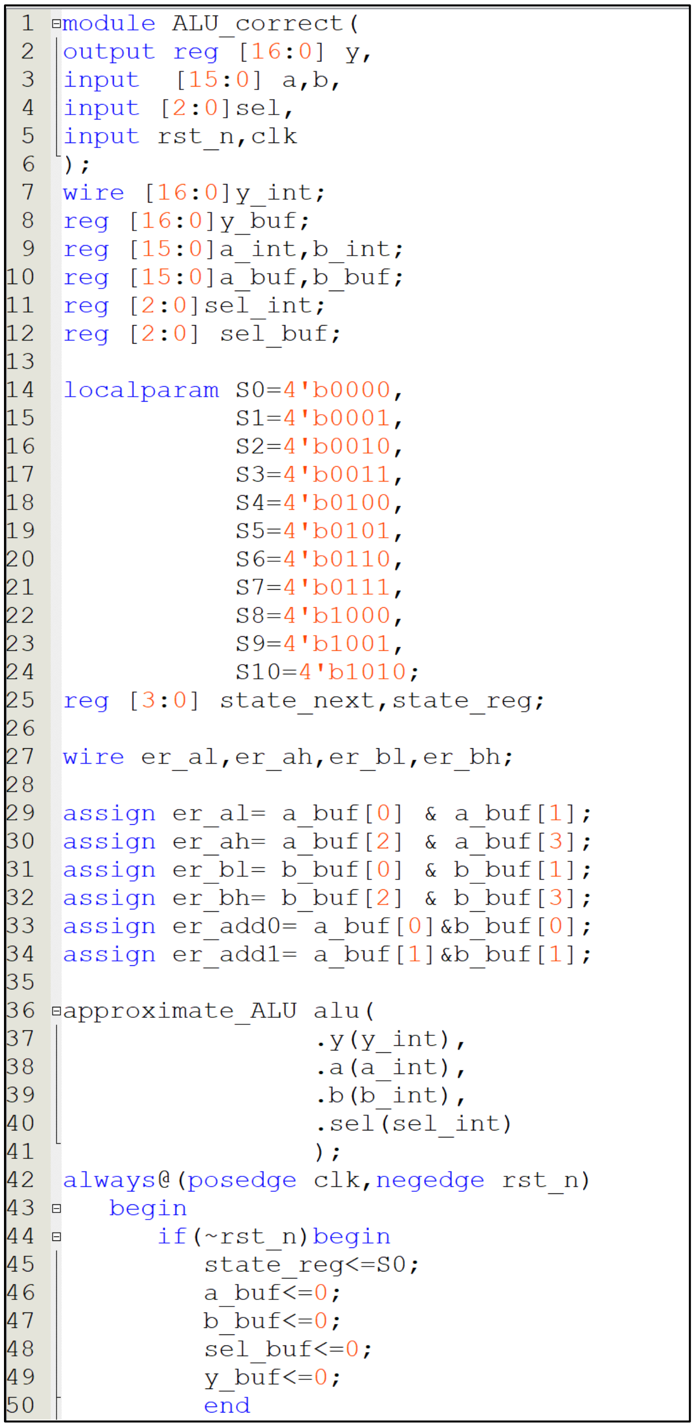
Dynamic Power Dissipation	I/O Power Dissipation	Static Power Dissipation
9.38 mW	28.40 mW	99.11 mW
Total Thermal Power Dissipation = 136.88 mW		

Table 8 The maximum frequency

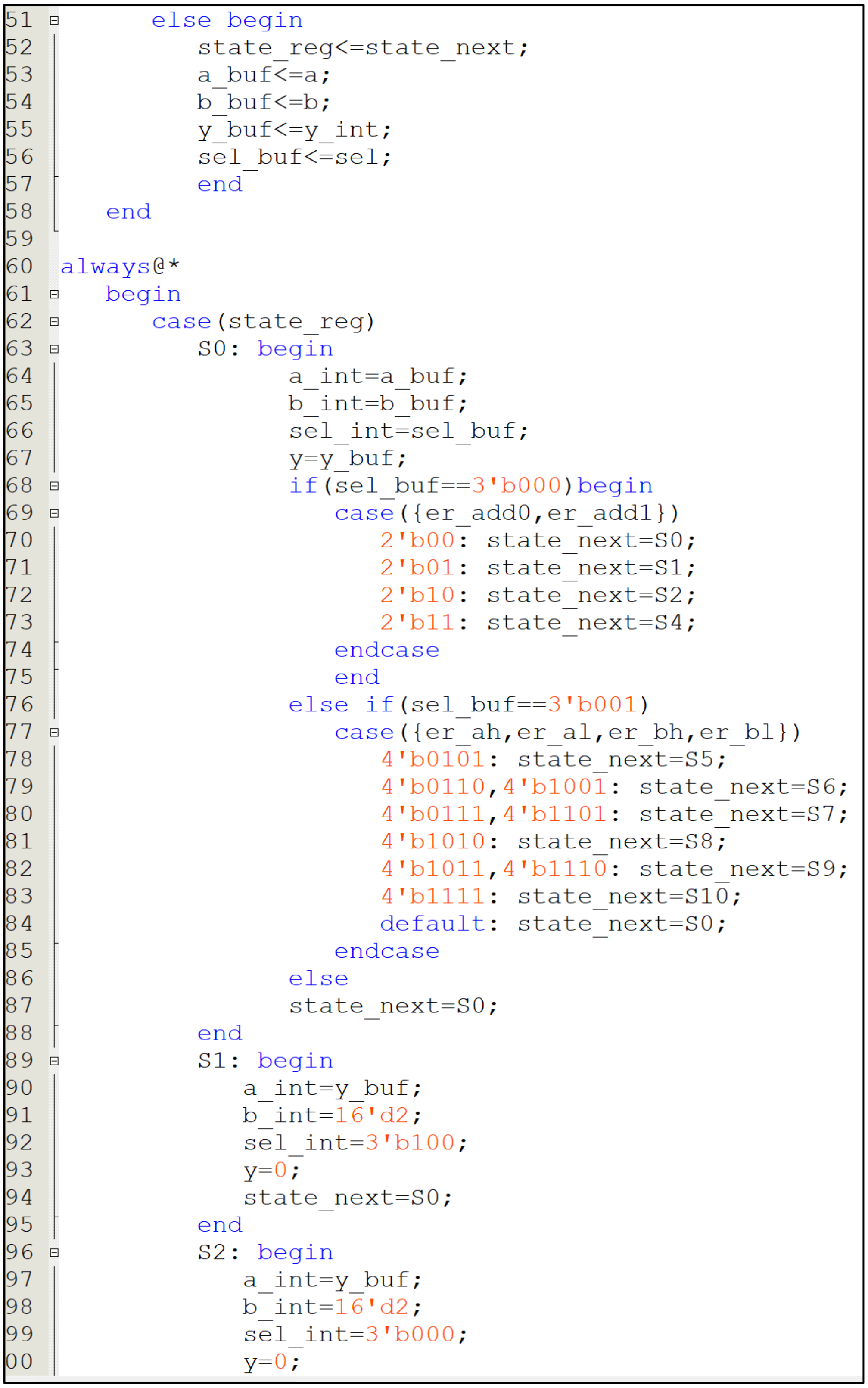
Frequency	Period
127.23 MHz	7.860 ns

As seen in table (6,7,8), the three main parameters and features of any digital design are obtained: the area, the power, and the delay. This information was got and simulated on cyclone III FPGA [EP3C120F780C7].

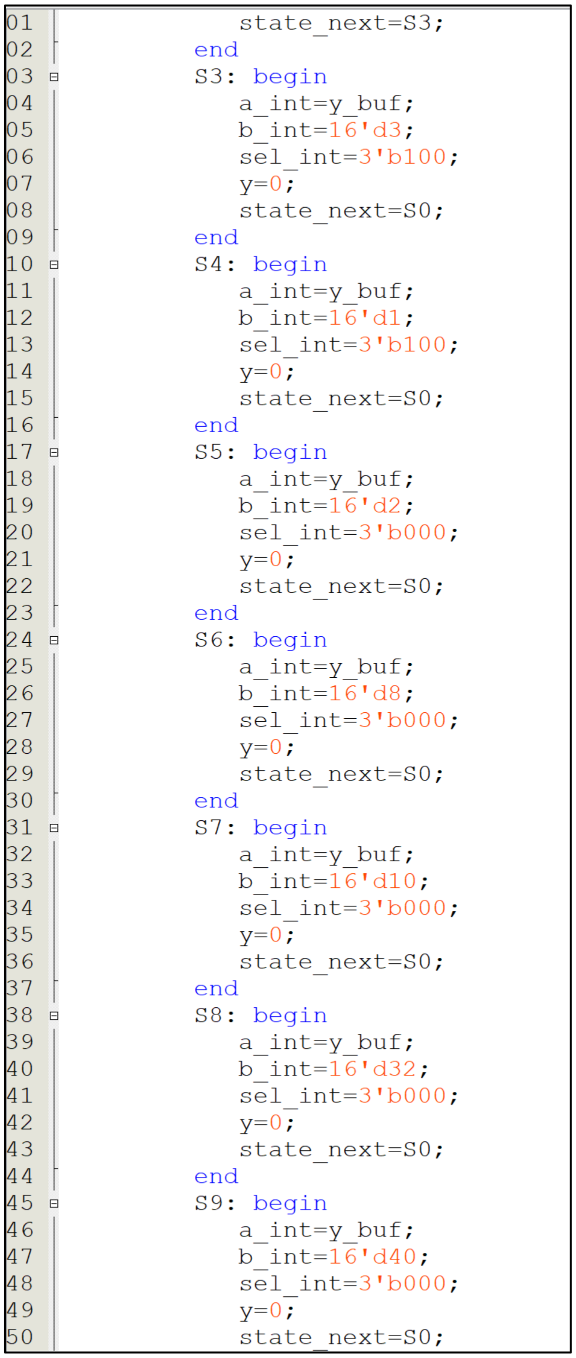
➤ **Verilog code:**



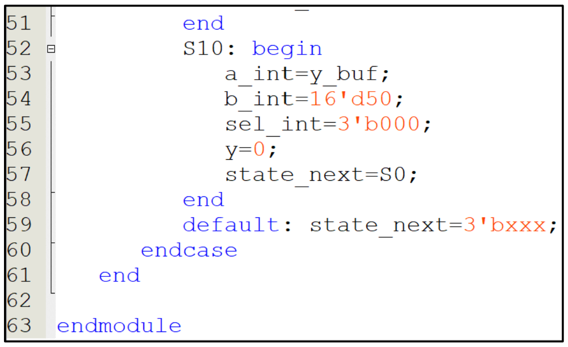
```
module ALU_correct(
output reg [16:0] y,
input  [15:0] a,b,
input [2:0]sel,
input rst_n,clk
);
wire [16:0]y_int;
reg [16:0]y_buf;
reg [15:0]a_int,b_int;
reg [15:0]a_buf,b_buf;
reg [2:0]sel_int;
reg [2:0] sel_buf;

localparam S0=4'b0000,
           S1=4'b0001,
           S2=4'b0010,
           S3=4'b0011,
           S4=4'b0100,
           S5=4'b0101,
           S6=4'b0110,
           S7=4'b0111,
           S8=4'b1000,
           S9=4'b1001,
           S10=4'b1010;
reg [3:0] state_next,state_reg;

wire er_al,er_ah,er_bl,er_bh;

assign er_al= a_buf[0] & a_buf[1];
assign er_ah= a_buf[2] & a_buf[3];
assign er_bl= b_buf[0] & b_buf[1];
assign er_bh= b_buf[2] & b_buf[3];
assign er_add0= a_buf[0]&b_buf[0];
assign er_add1= a_buf[1]&b_buf[1];

approximate_ALU alu(
                .y(y_int),
                .a(a_int),
                .b(b_int),
                .sel(sel_int)
                );
always@(posedge clk,negedge rst_n)
   begin
      if(~rst_n)begin
         state_reg<=S0;
         a_buf<=0;
         b_buf<=0;
         sel_buf<=0;
         y_buf<=0;
         end
```



```
      else begin
         state_reg<=state_next;
         a_buf<=a;
         b_buf<=b;
         y_buf<=y_int;
         sel_buf<=sel;
         end
   end

always@*
   begin
      case(state_reg)
         S0: begin
               a_int=a_buf;
               b_int=b_buf;
               sel_int=sel_buf;
               y=y_buf;
               if(sel_buf==3'b000)begin
                  case({er_add0,er_add1})
                     2'b00: state_next=S0;
                     2'b01: state_next=S1;
                     2'b10: state_next=S2;
                     2'b11: state_next=S4;
                  endcase
                  end
               else if(sel_buf==3'b001)
                  case({er_ah,er_al,er_bh,er_bl})
                     4'b0101: state_next=S5;
                     4'b0110,4'b1001: state_next=S6;
                     4'b0111,4'b1101: state_next=S7;
                     4'b1010: state_next=S8;
                     4'b1011,4'b1110: state_next=S9;
                     4'b1111: state_next=S10;
                     default: state_next=S0;
                  endcase
               else
               state_next=S0;
         end
         S1: begin
            a_int=y_buf;
            b_int=16'd2;
            sel_int=3'b100;
            y=0;
            state_next=S0;
         end
         S2: begin
            a_int=y_buf;
            b_int=16'd2;
            sel_int=3'b000;
            y=0;
```



```
            state_next=S3;
         end
         S3: begin
            a_int=y_buf;
            b_int=16'd3;
            sel_int=3'b100;
            y=0;
            state_next=S0;
         end
         S4: begin
            a_int=y_buf;
            b_int=16'd1;
            sel_int=3'b100;
            y=0;
            state_next=S0;
         end
         S5: begin
            a_int=y_buf;
            b_int=16'd2;
            sel_int=3'b000;
            y=0;
            state_next=S0;
         end
         S6: begin
            a_int=y_buf;
            b_int=16'd8;
            sel_int=3'b000;
            y=0;
            state_next=S0;
         end
         S7: begin
            a_int=y_buf;
            b_int=16'd10;
            sel_int=3'b000;
            y=0;
            state_next=S0;
         end
         S8: begin
            a_int=y_buf;
            b_int=16'd32;
            sel_int=3'b000;
            y=0;
            state_next=S0;
         end
         S9: begin
            a_int=y_buf;
            b_int=16'd40;
            sel_int=3'b000;
            y=0;
            state_next=S0;
```



```
         end
         S10: begin
            a_int=y_buf;
            b_int=16'd50;
            sel_int=3'b000;
            y=0;
            state_next=S0;
         end
         default: state_next=3'bxxx;
      endcase
   end

endmodule
```

➢ The schematic:

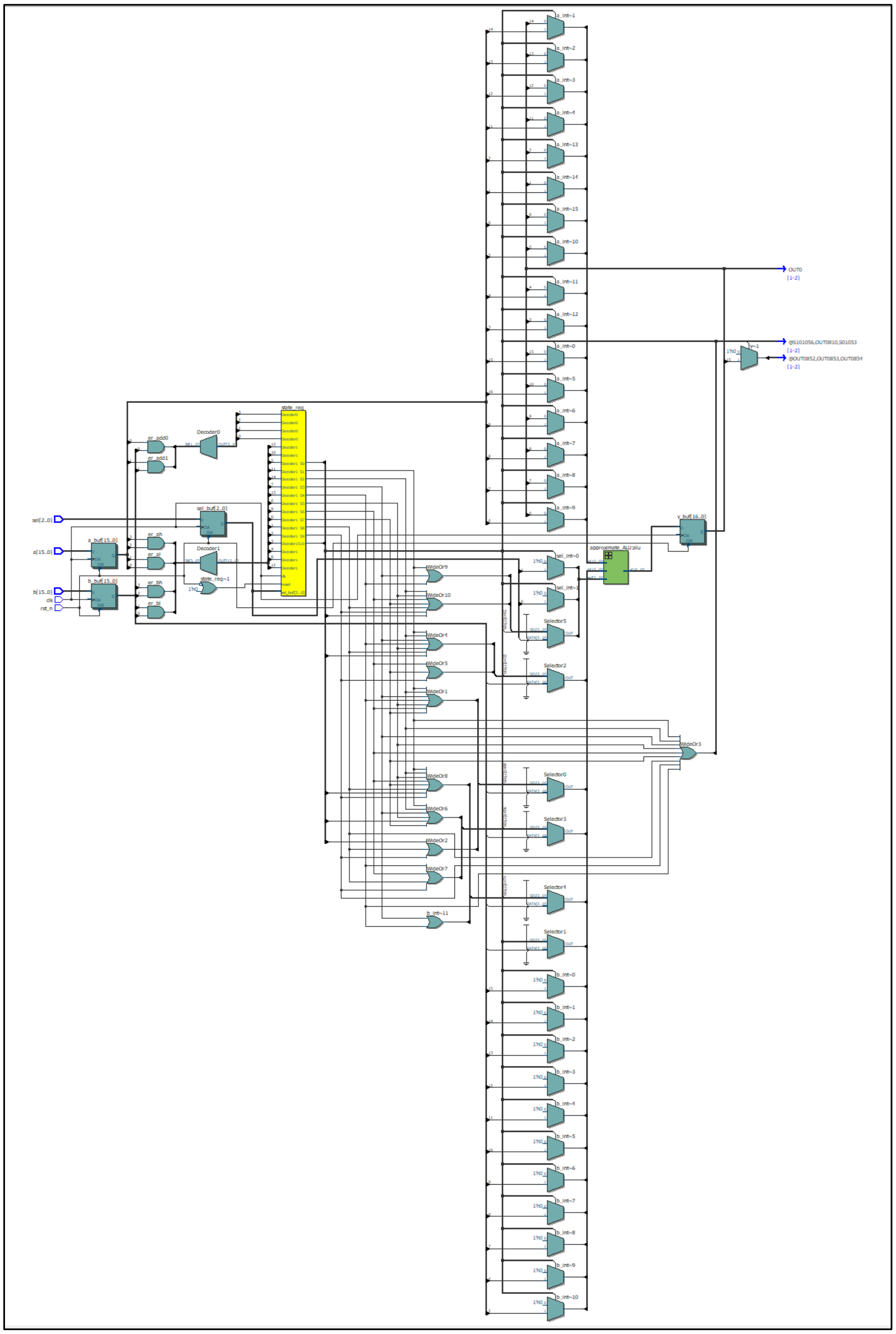
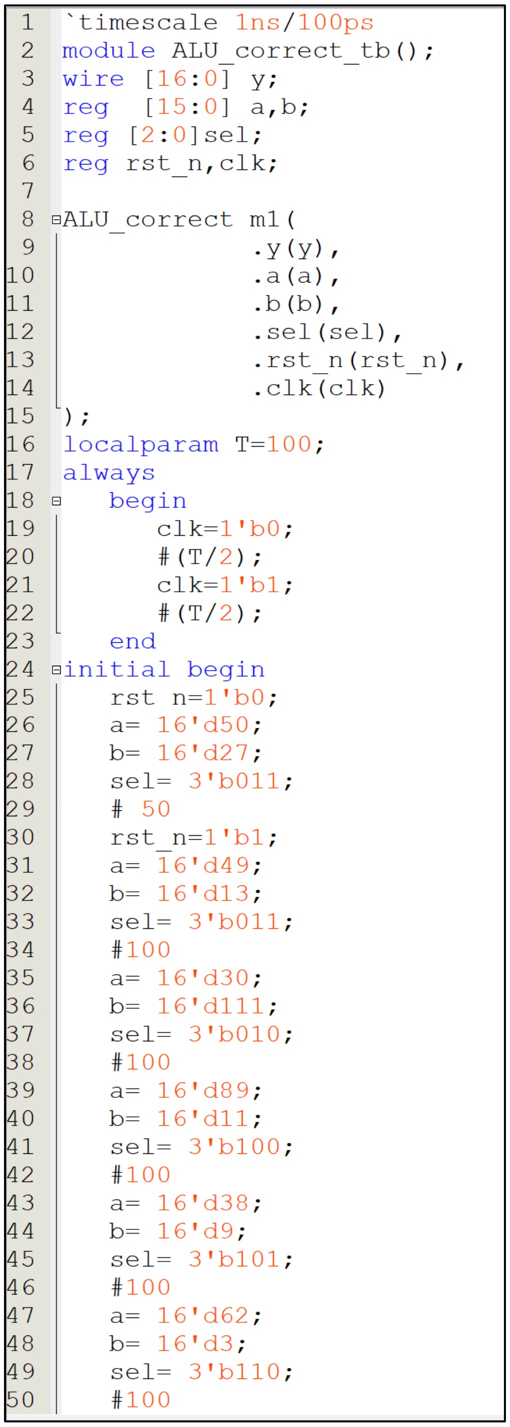


Figure 12 FSM Schematic

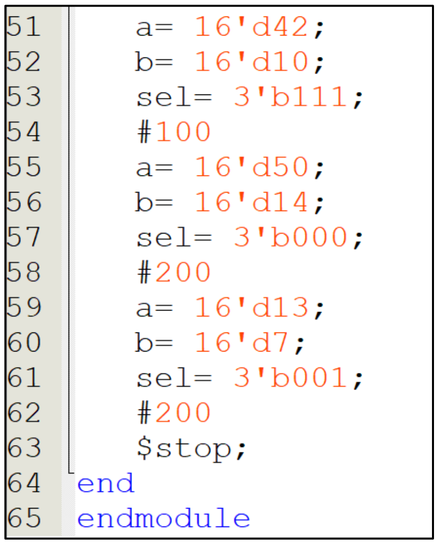
III. Simulation and testbench:

After designing the circuit, a testbench must be applied to it to ensure that the circuit works properly.

➢ Verilog code:



```verilog
`timescale 1ns/100ps
module ALU_correct_tb();
wire  [16:0]  y;
reg    [15:0]  a,b;
reg  [2:0]sel;
reg  rst_n,clk;

ALU_correct m1(
                .y(y),
                .a(a),
                .b(b),
                .sel(sel),
                .rst_n(rst_n),
                .clk(clk)
);
localparam T=100;
always
   begin
      clk=1'b0;
      #(T/2);
      clk=1'b1;
      #(T/2);
   end
initial begin
   rst_n=1'b0;
   a= 16'd50;
   b= 16'd27;
   sel= 3'b011;
   # 50
   rst_n=1'b1;
   a= 16'd49;
   b= 16'd13;
   sel= 3'b011;
   #100
   a= 16'd30;
   b= 16'd111;
   sel= 3'b010;
   #100
   a= 16'd89;
   b= 16'd11;
   sel= 3'b100;
   #100
   a= 16'd38;
   b= 16'd9;
   sel= 3'b101;
   #100
   a= 16'd62;
   b= 16'd3;
   sel= 3'b110;
   #100
```



```verilog
   a= 16'd42;
   b= 16'd10;
   sel= 3'b111;
   #100
   a= 16'd50;
   b= 16'd14;
   sel= 3'b000;
   #200
   a= 16'd13;
   b= 16'd7;
   sel= 3'b001;
   #200
   $stop;
end
endmodule
```

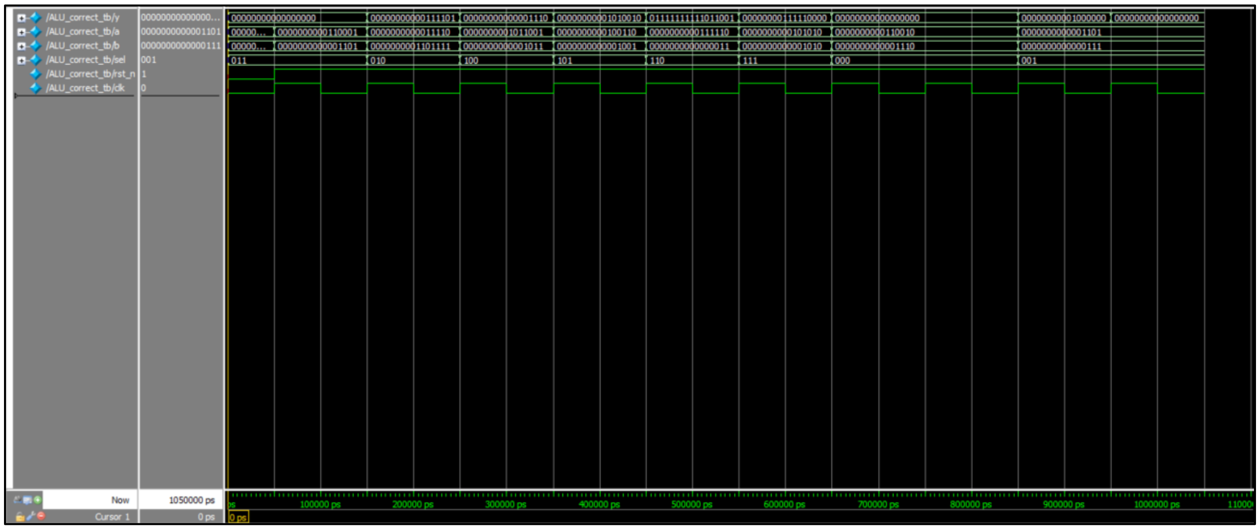


Figure 13 The output waveform of textbench

As seen in figure (13), the 8 operations are tested, and the output is illustrated above. As a result, the circuit works properly.

IV. Conclusion

In conclusion, an approximate Arithmetic Logic Unit (ALU) and Approximate Computing offer promising solutions in improving the efficiency and performance of computing systems while tolerating a certain degree of error or imprecision. By relaxing the requirement for exact precision, approximate ALUs and Approximate Computing techniques enable significant reductions in power consumption, area utilization, and computational time, making them particularly suitable for applications where a small loss in accuracy can be tolerated.

Approximate Computing, on the other hand, extends beyond the ALU and encompasses a broader approach to designing computing systems that embrace imprecision. It involves employing approximation techniques at various levels, including circuit design, architecture, and system-level optimization. This approach allows for tailoring the degree of approximation to match the requirements of specific applications, thereby achieving a trade-off between accuracy and resource utilization.

The benefits of approximate ALUs and Approximate Computing are manifold. Firstly, they offer significant energy savings by reducing the number of operations performed or relaxing the precision requirements. This energy efficiency is particularly valuable in battery-constrained devices and large-scale data centers that consume vast amounts of power. Secondly, approximate computing techniques can lead to smaller and more compact designs, enabling higher integration densities and reduced manufacturing costs. Lastly, these approaches often result in faster computations as the relaxed precision allows for the use of simplified and faster arithmetic circuits.

However, it is important to note that the adoption of approximate ALUs and Approximate Computing is highly dependent on the specific application and the tolerance for errors. While these techniques excel in domains where a small loss in accuracy is acceptable, they may not be suitable for applications that require exact results, such as safety-critical systems or financial computations. Additionally, careful consideration must be given to the design and evaluation of approximation techniques to ensure that the introduced errors do not compromise the overall system functionality or introduce unexpected side effects.

In summary, approximate ALUs and Approximate Computing offer a viable approach to improve the efficiency and performance of computing systems by trading off accuracy for gains in energy consumption, area utilization, and computational speed. These techniques provide flexible solutions that can be tailored to meet the specific requirements of various applications, opening up new possibilities for energy-efficient computing in a wide range of domains.

V. References

1- Kulkarni, P., Gupta, P., & Ercegovac, M. (2011). Trading accuracy for power with an underdesigned multiplier architecture. 2011 24th Internatioal Conference on VLSI Design. https://doi.org/10.1109/vlsid.2011.51

2- Jiang, H., Han, J., & Lombardi, F. (2015). A comparative review and evaluation of approximate adders. Proceedings of the 25th Edition on Great Lakes Symposium on VLSI. https://doi.org/10.1145/2742060.2743760