International Institute of Information Technology
Bangalore

VLSI PROJECT ELECTIVE

VARIABLE PRECISION APPROXIMATE FLOATING POINT MULTIPLIER

B Sathiya Naraayanan
(IMT2020534)

November 22, 2023


iit-b
ज्ञानमुत्तमम्

INTRODUCTION

The Variable-precision approximate floating-point multiplier is proposed for energy efficient deep learning computation. The proposed architecture supports approximate multiplication with BFloat16 format.The approximation is done in form of truncation.

ARCHITECTURE

The inputs are divided into sign bits,exponent and mantissa bits seperately. There are three block rams used,one for each inputs and one for the output. Sign and exponent module is used for getting the sign and exponent of the multiplied result.The calculated exponent is used for creating mask which is used to decide the number of bits to be truncated.This is done using precision control module.The mantissas' are given as input to booth multiplier after appending '01' bits.The '1' bit is the implied bit that comes before the decimal point.The booth muliplication algorithm uses input in signed format, so '0' bit is added.All the outputs are processed and normalized for final output in the normalization module.

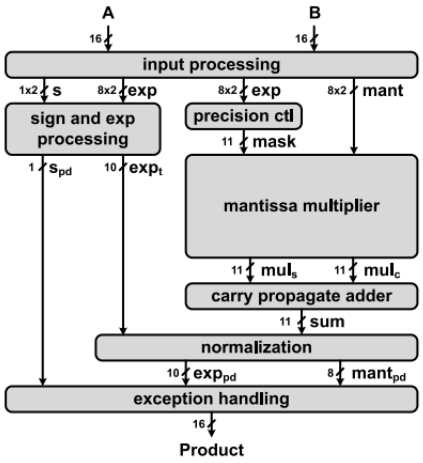

Figure 1: Multiplier Architecture

IMPLEMENTATION

The proposed architecture was implemented in verilog and implemented on Xilinx basys-3 fpga board using vivado.The multiplier was implemented without masking and precision control unit for comparing the results.

The verilog implementation of the multiplier with masking is as follows.

```
    ////////////////////////////////////////////////////////////////////////
module float_mul(input [0:15] num1,input [0:15] num2,output [0:15] out);

    wire s1,s2;
    wire [0:7] ex1,ex2;
    wire [0:8] m1,m2;

    wire s;
    wire [0:9] exp;
    wire [0:10] mant;
    wire [0:10] mask;
    wire [0:9] exponent;
    wire [0:6] mantissa;

    assign s1=num1[0];
    assign s2=num2[0];
    assign ex1=num1[1:8];
    assign ex2=num2[1:8];
    assign m1={2'b01,num1[9:15]};
    assign m2={2'b01,num2[9:15]};


    sign_exp se(s1,s2,ex1,ex2,s,exp);
    prec_ctrl pc(exp,mask);
    booth_mul bm(m1,m2,mask,mant);
    normal nz(mant,exp,exponent,mantissa);

    assign out={s,exponent[2:9],mantissa};
endmodule

module sign_exp(input s1,input s2,input [0:7] ex1,input [0:7] ex2,output s,output [0:9] exp);
    assign s=s1^s2;
    assign exp={2'b0,ex1}+{2'b0,ex2}-10'd254;
endmodule

module normal(input [0:10] mant,input [0:9] exp,output [0:9] exponent,output [0:6] mantissa);
   assign mantissa=mant[2:8];
   assign exponent=exp+1'b1+10'd127;
endmodule

module prec_ctrl(input [0:9] exp,output reg [0:10] mask);
    wire [0:3] rg;
    assign rg=exp[2:5];
    always@(*) begin
    case(rg)
```

```
    4'b0000:mask=11'b11111111111;
    4'b0001:mask=11'b11111111110;
    4'b0010:mask=11'b11111111100;
    4'b0011:mask=11'b11111111000;
    4'b0100:mask=11'b11111110000;
    4'b0101:mask=11'b11111100000;
    4'b0110:mask=11'b11111000000;
    4'b0111:mask=11'b11110000000;
    4'b1000:mask=11'b11110000000;
    4'b1001:mask=11'b11111000000;
    4'b1010:mask=11'b11111100000;
    4'b1011:mask=11'b11111110000;
    4'b1100:mask=11'b11111111000;
    4'b1101:mask=11'b11111111100;
    4'b1110:mask=11'b11111111110;
    4'b1111:mask=11'b11111111111;
    default:mask=11'b11111111111;
    endcase
    end
endmodule


module booth_mul(input wire [8:0] A,
                 input wire [8:0] B,
                 input [10:0] mask,
                 output wire [10:0] P);
        reg [2:0] bits[4:0];
        reg [9:0] pp[4:0];

        //wire [10:0] p1,p2,p3,p4;
        wire [8:0] A_;//minus A
        wire [15:0] pp1;
        wire [15:0] pp2;
        wire [15:0] pp3;
        wire [15:0] pp4;
        wire [15:0] pp5;
        integer m1;
        assign A_=~A+1;

        always@(A or B or A_) begin

        bits[0]={B[1],B[0],1'b0};
        bits[4]=3'b001;

        for(m1=1;m1<4;m1=m1+1)
            bits[m1]={B[2*m1+1],B[2*m1],B[2*m1-1]};

        for(m1=0;m1<5;m1=m1+1) begin
```

```
            case(bits[m1])

            3'b001:pp[m1]={1'b0,A};
            3'b010:pp[m1]={1'b0,A};
            3'b011:pp[m1]={A,1'b0};
            3'b100:pp[m1]={A_,1'b0};
            3'b101:pp[m1]={A_[8],A_};
            3'b110:pp[m1]={A_[8],A_};
            default:pp[m1]=0;

            endcase
        end
        end

        assign pp1={{6{pp[0][9]}},pp[0]};
        assign pp2={{4{pp[1][9]}},pp[1],2'b0};
        assign pp3={{2{pp[2][9]}},pp[2],4'b0};
        assign pp4={pp[3],6'b0};
        assign pp5={pp[4][7:0],8'b0};
        //assign P=pp1+pp2+pp3+pp4;
        pp_adder ppa(mask,pp1,pp2,pp3,pp4,pp5,P);//,p1,p2,p3,p4);

endmodule

module pp_adder(input [10:0] mask,
                input [15:0] pp1,
                input [15:0] pp2,
                input [15:0] pp3,
                input [15:0] pp4,
                input [15:0] pp5,
                output [10:0] prod);/
    wire [10:0] p1,p2,p3,p4,p5;
    assign p1 = pp1[15:5] & mask;
    assign p2 = pp2[15:5] & mask;
    assign p3 = pp3[15:5] & mask;
    assign p4 = pp4[15:5] & mask;
    assign p5 = pp5[15:5] & mask;

    assign prod=pp1[15:5]+pp2[15:5]+pp3[15:5]+pp4[15:5]+pp5[15:5];
endmodule
    ////////////////////////////////////////////////////////////////////////


    The verilog implementation of the multiplier without masking is as follows.


    ////////////////////////////////////////////////////////////////////////
```

```
module float_mul_maskless(input [0:15] num1,input [0:15] num2,output [0:15] out);

    wire s1,s2;
    wire [0:7] ex1,ex2;
    wire [0:8] m1,m2;

    wire s;
    wire [0:9] exp;
    wire [0:10] mant;
    wire [0:9] exponent;
    wire [0:6] mantissa;

    assign s1=num1[0];
    assign s2=num2[0];
    assign ex1=num1[1:8];
    assign ex2=num2[1:8];
    assign m1={2'b01,num1[9:15]};
    assign m2={2'b01,num2[9:15]};

    sign_exp se(s1,s2,ex1,ex2,s,exp);
    booth_mul bm(m1,m2,mant);
    normal nz(mant,exp,exponent,mantissa);

    assign out={s,exponent[2:9],mantissa};
endmodule

module sign_exp(input s1,input s2,input [0:7] ex1,input [0:7] ex2,output s,output [0:9] exp);
    assign s=s1^s2;
    assign exp={2'b0,ex1}+{2'b0,ex2}-10'd254;
endmodule

module normal(input [0:10] mant,input [0:9] exp,output [0:9] exponent,output [0:6] mantissa);
   assign mantissa=mant[2:8];
   assign exponent=exp+1'b1+10'd127;
endmodule


module booth_mul(input wire [8:0] A,input wire [8:0] B,output wire [10:0] P);

        reg [2:0] bits[4:0];
        reg [9:0] pp[4:0];

        wire [8:0] A_;//minus A
        wire [15:0] pp1;
        wire [15:0] pp2;
        wire [15:0] pp3;
        wire [15:0] pp4;
        wire [15:0] pp5;
```

```
        integer m1;
        assign A_=~A+1;

        always@(A or B or A_) begin

        bits[0]={B[1],B[0],1'b0};
        bits[4]=3'b001;

        for(m1=1;m1<4;m1=m1+1)
            bits[m1]={B[2*m1+1],B[2*m1],B[2*m1-1]};

        for(m1=0;m1<5;m1=m1+1) begin
            case(bits[m1])

            3'b001:pp[m1]={1'b0,A};
            3'b010:pp[m1]={1'b0,A};
            3'b011:pp[m1]={A,1'b0};
            3'b100:pp[m1]={A_,1'b0};
            3'b101:pp[m1]={A_[8],A_};
            3'b110:pp[m1]={A_[8],A_};
            default:pp[m1]=0;

            endcase
        end
        end

        assign pp1={{6{pp[0][9]}},pp[0]};
        assign pp2={{4{pp[1][9]}},pp[1],2'b0};
        assign pp3={{2{pp[2][9]}},pp[2],4'b0};
        assign pp4={pp[3],6'b0};
        assign pp5={pp[4][7:0],8'b0};

        assign P=pp1[15:5]+pp2[15:5]+pp3[15:5]+pp4[15:5]+pp5[15:5];

endmodule

    ////////////////////////////////////////////////////////////////////////
```

FPGA AND ASIC RESULTS

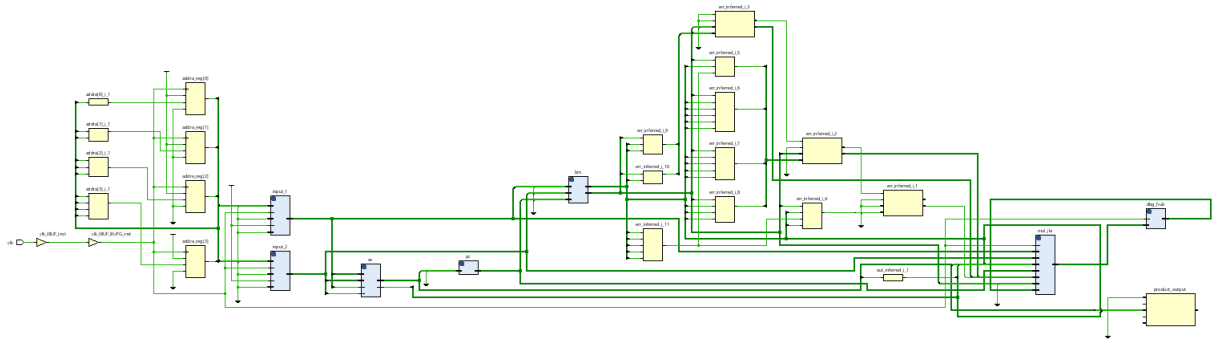

Figure 2: Synthesized result

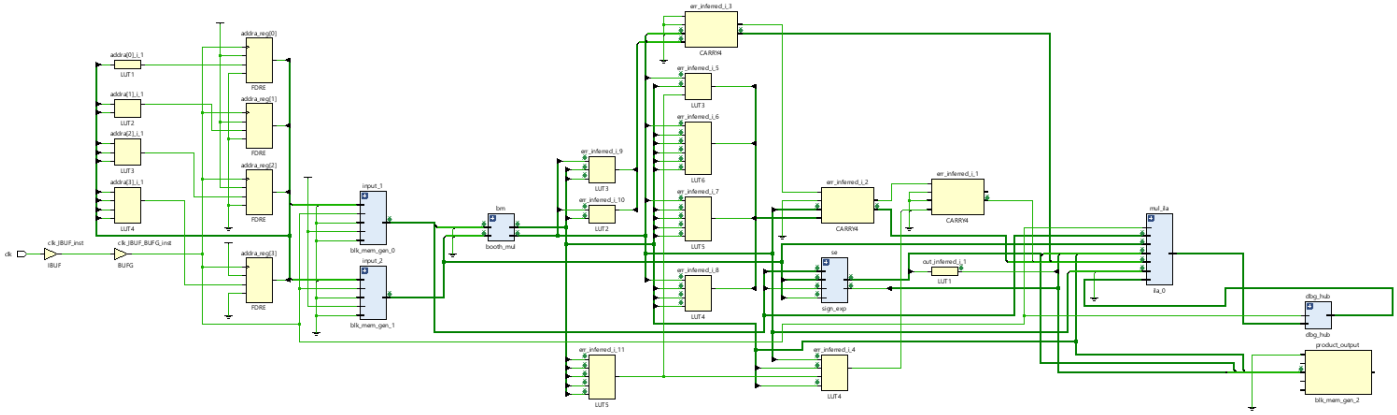

Figure 3: Synthesized Design(without masking)

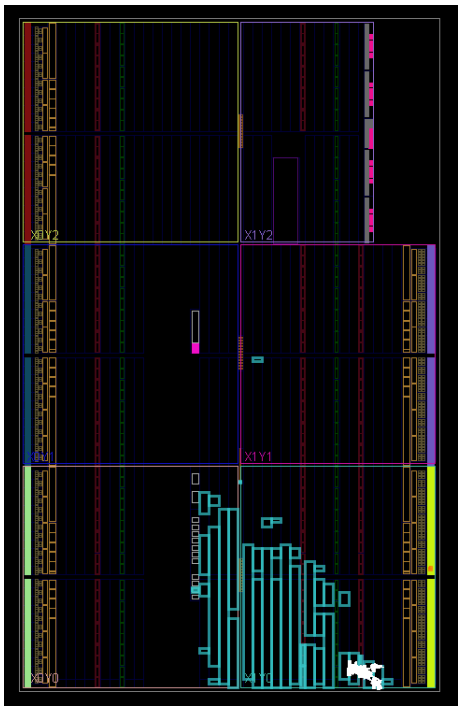

Figure 4: Implementated Result

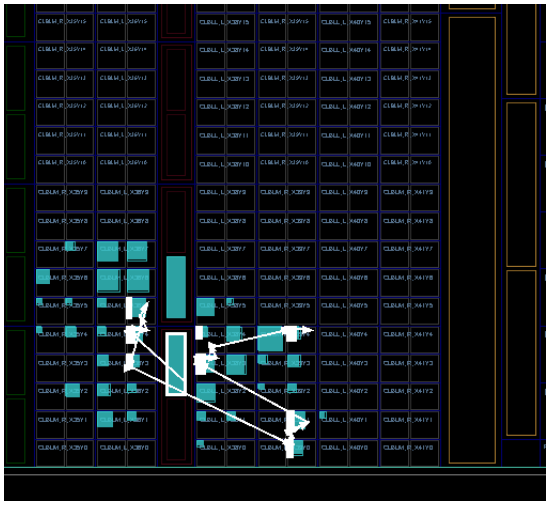

Figure 5: Implementation

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)	BSCANE2 (4)
top	1609	2376	3	768	1436	173	45	1	2	1
bm (booth_mul)	82	0	0	32	82	0	0	0	0	0
dbg_hub (dbg_hub)	449	741	0	240	425	24	0	0	1	1
input_1 (blk_mem_gen_0)	0	0	0	0	0	0	4.5	0	0	0
input_2 (blk_mem_gen_1)	0	0	0	0	0	0	4.5	0	0	0
mul_ila (ila_0)	1054	1631	3	520	905	149	36	0	0	0
pc (prec_ctrl)	5	0	0	3	5	0	0	0	0	0
product_output (blk_mem_gen_2)	0	0	0	0	0	0	0	0	0	0
se (sign_exp)	9	0	0	3	9	0	0	0	0	0

Figure 6: Resource Utilisation

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)	BSCANE2 (4)
top	1484	2286	3	735	1324	160	40	1	2	1
bm (booth_mul)	57	0	0	20	57	0	0	0	0	0
dbg_hub (dbg_hub)	448	741	0	230	424	24	0	0	1	1
input_1 (blk_mem_gen_0)	0	0	0	0	0	0	4.5	0	0	0
input_2 (blk_mem_gen_1)	0	0	0	0	0	0	4.5	0	0	0
mul_ila (ila_0)	959	1541	3	486	823	136	31	0	0	0
product_output (blk_mem_gen_2)	0	0	0	0	0	0	0	0	0	0
se (sign_exp)	9	0	0	3	9	0	0	0	0	0

Figure 7: Resource Utilisation(without masking)

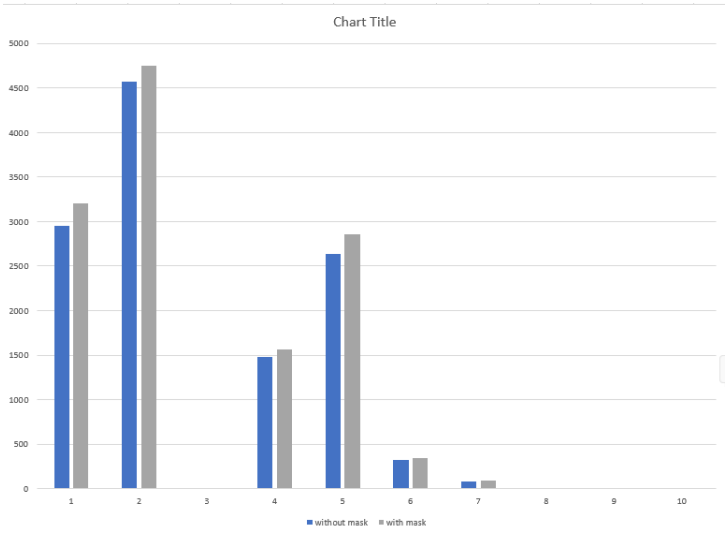

Figure 8: Resource Utilisation Comparison

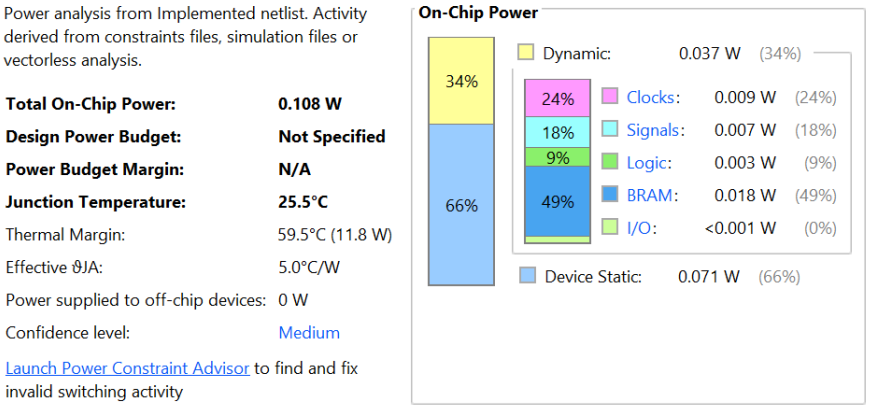

Figure 9: Power

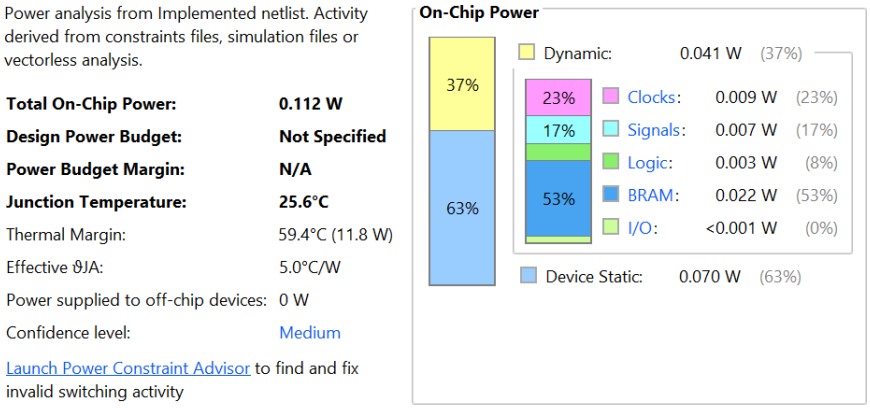

Figure 10: Power(without masking)

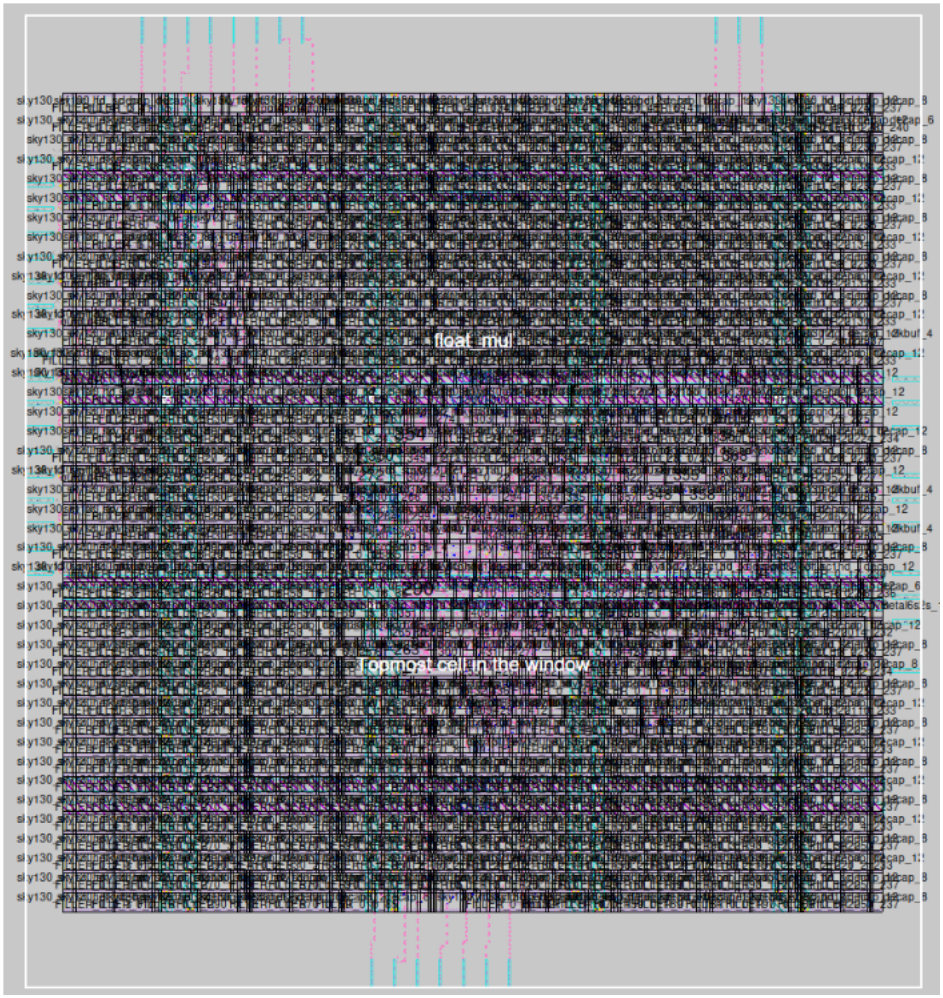

Figure 11: Layout

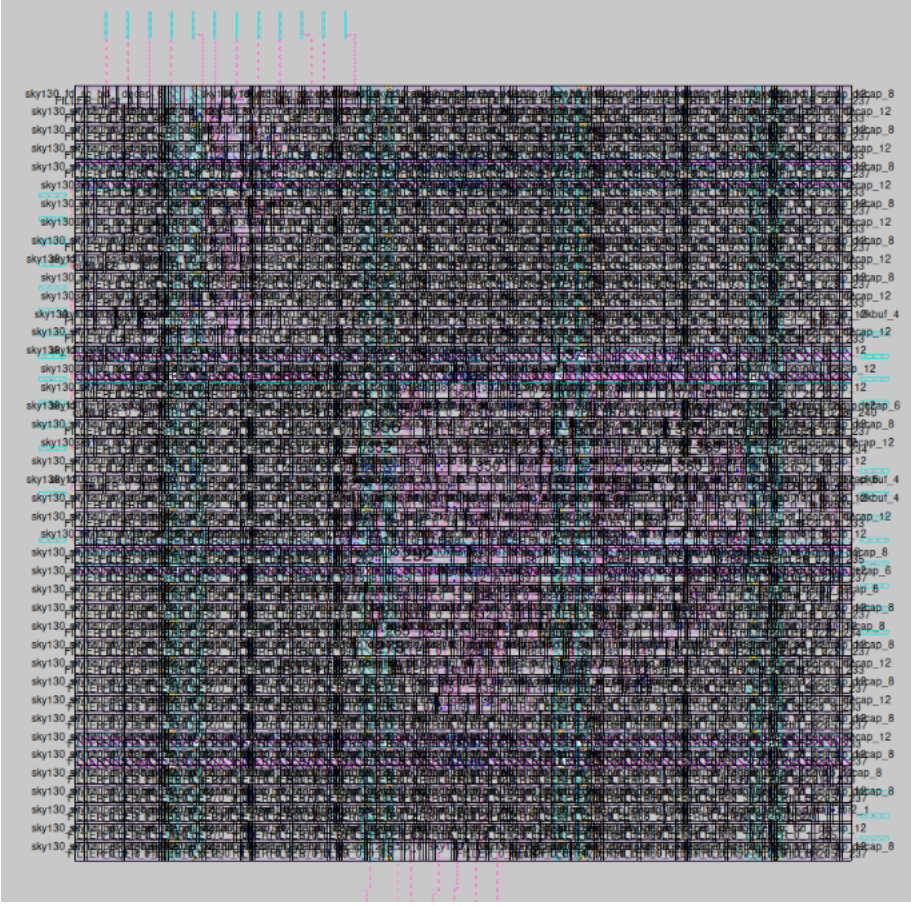

Figure 12: Layout(without masking)

```
% box
Root cell box:
          width x height  (   llx,   lly  ), (   urx,   ury  )  area (units^2)

microns:  125.630 x 136.350  ( 0.000,  0.000), ( 125.630,  136.350)  17129.650
lambda:    12563 x 13635   (     0,  0    ), ( 12563,  13635)  171296505
%
```

Figure 13: Area

```
% box
Root cell box:
          width x height  (   llx,   lly  ), (   urx,   ury  )  area (units^2)

microns:  122.840 x 118.890  (-0.390,  7.980), ( 122.450,  126.870)  14604.447
lambda:    12284 x 11889   (   -39,  798  ), ( 12245,  12687)  146044476
%
```

Figure 14: Area(without masking)

Results

The result utilisation of the multiplier with masking is more than the one without masking.A significant part of it is due to ILA of Vivado. The proposed design however has less power consumption than the one without variable precission approximation.The number of bits to be truncated is decided using the exponent value.The Bit error rate against truncated bit-width is given below.

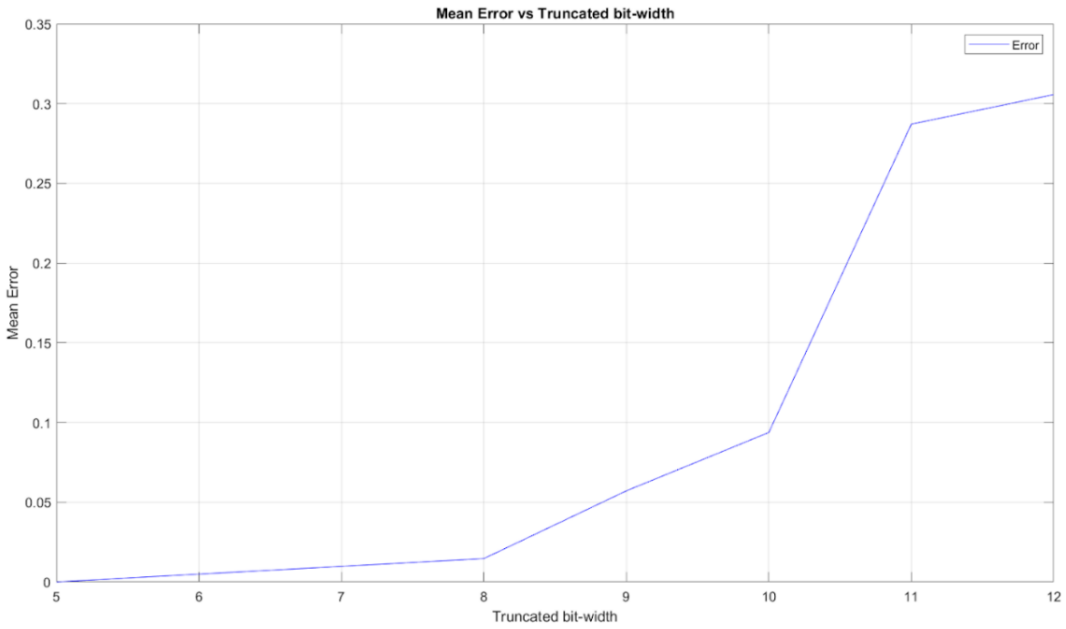

Figure 15: Error vs Truncated bit-width

References

- Variable-Precision Approximate Floating-Point Multiplier for Efficient Deep Learning Computation Hao Zhang , Member, IEEE, and Seok-Bum Ko , Senior Member, IEEE
- J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in Proc. 18th IEEE Eur. Test Symp. (ETS), May 2013, pp. 1–6.
- Novel, Configurable Approximate Floating-point Multipliers for Error-Resilient Applications Vishesh Mishra1 , Sparsh Mittal2 , Rekha Singhal3 , Manoj Nambiar3 1 IIT Kanpur, 2 IIT Roorkee, 3TCS Research, India vishesh
- Kaur, Navdeep Patial, Rajeev. (2013). Implementation of Modified Booth Multiplier using Pipeline Technique on FPGA. International Journal of Computer Applications. 68. 38-41. 10.5120/11666-7261.
- Design and Implementation of Radix 4 Based Multiplication on FPGA Supriya S. Saste1 Dept. of Electronics Telecommunication Trinity College of Engineering Research Pune,

India. Prof. Anil G. Sawant2 Dept. of Electronics Telecommunication Trinity College of Engineering Research Pune, India.

- Implementation of Modified Booth Algorithm (Radix 4) and its Comparison with Booth Algorithm (Radix-2) Sukhmeet Kaur1 , Suman2 and Manpreet Signh Manna3 1 ECE, SSIET (P.T.U), Derabassi, Punjab, India. 2 ECE, SSIET, Derabassi, Punjab, India. 3 EIE, SLIET (Deemed University), Longowal, Sangrur, India.