STUDY PROJECT
EEE F266

FINAL PROJECT REPORT

Abhinandan Sharma
2018A3PS0095P

A REPORT

ON

DESIGN OF HIGH-PERFORMANCE MULTIPLIER ARCHITECTURES

Prepared by

ABHINANDAN SHARMA 2018A3PS0095P

Submitted to

PROF. S. GURUNARAYANAN

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
July, 2020

ACKNOWLEDGEMENT

Working on this project has been a great learning experience for me and has improved my skills in programming and concepts of Digital Design to a large extent. I am grateful to Prof. S. Gurunarayanan for allowing me to pursue this project under him and giving me the right direction to work on.

I would like to thank Pawan sir for guiding me about the softwares used in this project at various steps.

Thanks are also due to my family and friends for consistently encouraging me and giving me the much needed moral support.

ABSTRACT

Three multiplier architectures have been explored during this project:

1. Array Multiplier (for signed/ unsigned numbers)
2. Booth Multiplier
3. Modified Booth Multiplier

These architectures were designed and then simulated on ModelSim (in Verilog HDL).

After successful simulation, synthesis was done in Xilinx Vivado 2019.2 and then elaborated designs were generated.

Post synthesis timing/ functional simulation was also successfully carried out to verify correctness of the synthesized design.

TABLE OF CONTENTS

Acknowledgements ii

Abstract iii

1. INTRODUCTION 1

2. Design of Architectures 2
   2.1 Architecture of 8-bit Array Multiplier for unsigned numbers 2
   2.2 Architecture of 8-bit Array Multiplier for signed numbers 4
   2.3 Architecture of 4 Bit Booth Multiplier 6
   2.4 Architecture of 4-Bit Modified Booth Multiplier 8

3. Verilog Codes and Simulation Results 10
   3.1 Array multiplier (16-bit unsigned numbers) 10
   3.2 Array multiplier (16-bit signed numbers) 12
   3.3 Booth Multiplier (16-bit signed numbers) 14
   3.4 Modified Booth Multiplier (16-bit signed numbers) 16

4. RTL Analysis (Elaborated designs post synthesis) 18
   4.1 Array multiplier (16-bit unsigned numbers) 18
   4.2 Array multiplier (16-bit signed numbers) 18
   4.3 Booth Multiplier (16-bit signed numbers 19
   4.4 Modified Booth Multiplier (16-bit signed numbers) 23

5. Summary of Timing/ Power/ Utilization 26

6. FUTURE WORK 28

References 29

1. INTRODUCTION

Three multiplier architectures were designed and simulated during the project:

1. Array Multiplier (for signed/ unsigned numbers)
2. Booth Multiplier
3. Modified Booth Multiplier

The Section-2 of the report explains the design of these architectures (in this order), the Section-3 consists of the Verilog codes and the simulation results, the Section-4 shows the elaborated designs post synthesis and Section-5 displays the final summary after synthesis and implementation.

To explain the design of the architecture, 8-bit / 4-bit architecture has been used but because of the symmetry in the designs, the designs can be easily extended for higher order bits as well.

As all the codes have a modular approach, only the code for 16-bit multiplier have been included in the report to keep it brief but instructions to extend the code are also mentioned.

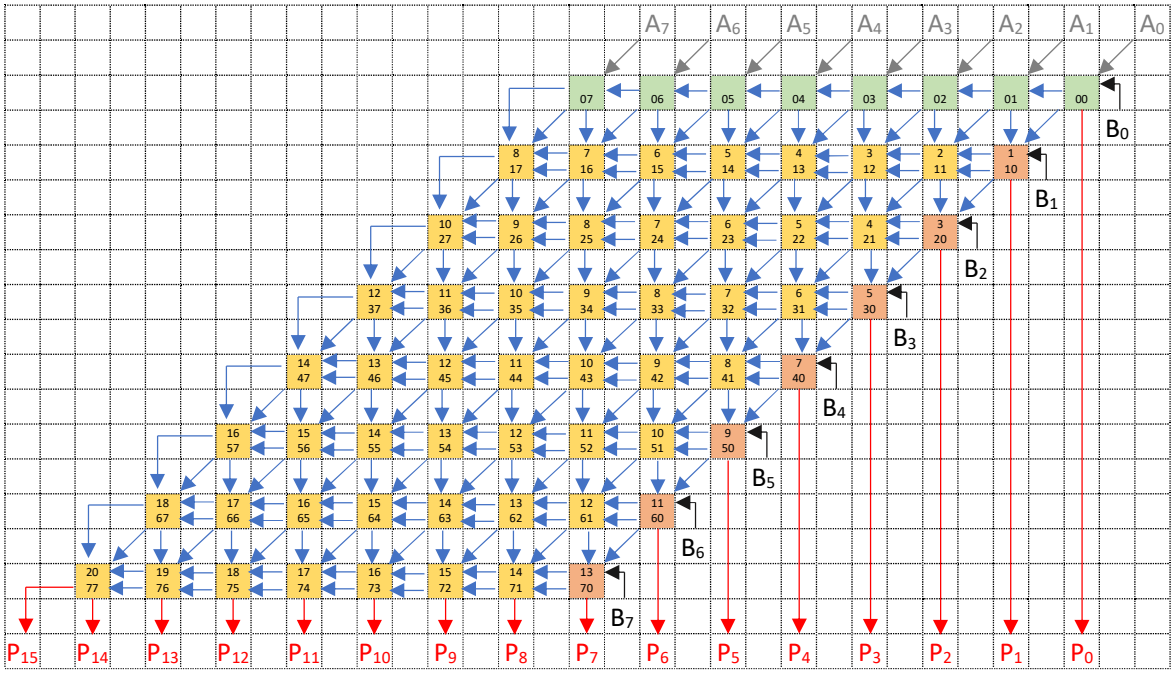
The simulation was done using ModelSim and the results have been included in the report in the form of screenshots.

For Synthesis, Xilinx Vivado 2019.2 tool is used and the board used is ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1).

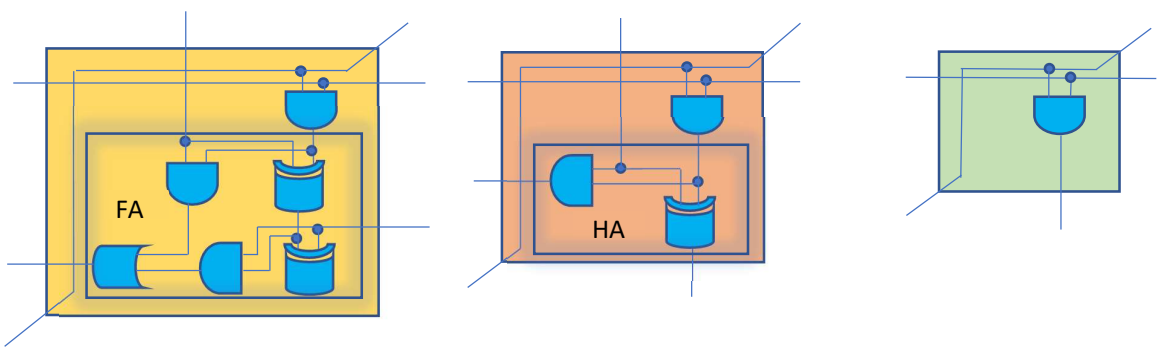
The screenshots of the schematics are included in this report.

2. Design of Architectures

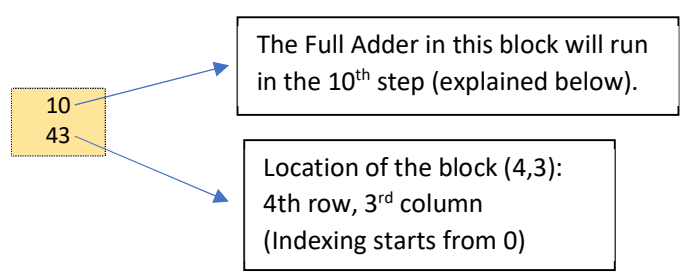
2.1. Architecture of 8-bit Array Multiplier for unsigned numbers:



Inner structure of each block in the architecture (colour coded):



Labelling of each block:



Multiplicand: $A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0$ (unsigned)

Multiplier: $B_7\ B_6\ B_5\ B_4\ B_3\ B_2\ B_1\ B_0$ (unsigned)

Product: $P_{15}\ P_{14}\ P_{13}\ P_{12}\ P_{11}\ P_{10}\ P_9\ P_8\ P_7\ P_6\ P_5\ P_4\ P_3\ P_2\ P_1\ P_0$ (unsigned)

Flow of the Architecture:

Step 0: The AND gates in each block will compute their results. For example, the AND gate in the block (4,3) will compute the value of B_4A_3.

Step 1:	10						
Step 2:	11						
Step 3:	12	20					
Step 4:	13	21					
Step 5:	14	22	30				
Step 6:	15	23	31				
Step 7:	16	24	32	40			
Step 8:	17	25	33	41			
Step 9:		26	34	42	50		
Step 10:		27	35	43	51		
Step 11:			36	44	52	60	
Step 12:			37	45	53	61	
Step 13:				46	54	62	70
Step 14:				47	55	63	71
Step 15:					56	64	72
Step 16:					57	65	73
Step 17:						66	74
Step 18:						67	75
Step 19:							76
Step 20:							77

The orange indices in steps 1, 3, 5 ,7, 9, 11, 13 represent the orange blocks in the architecture (containing Half Adders) and the yellow indices represent the yellow blocks in the architecture (containing Full Adders)

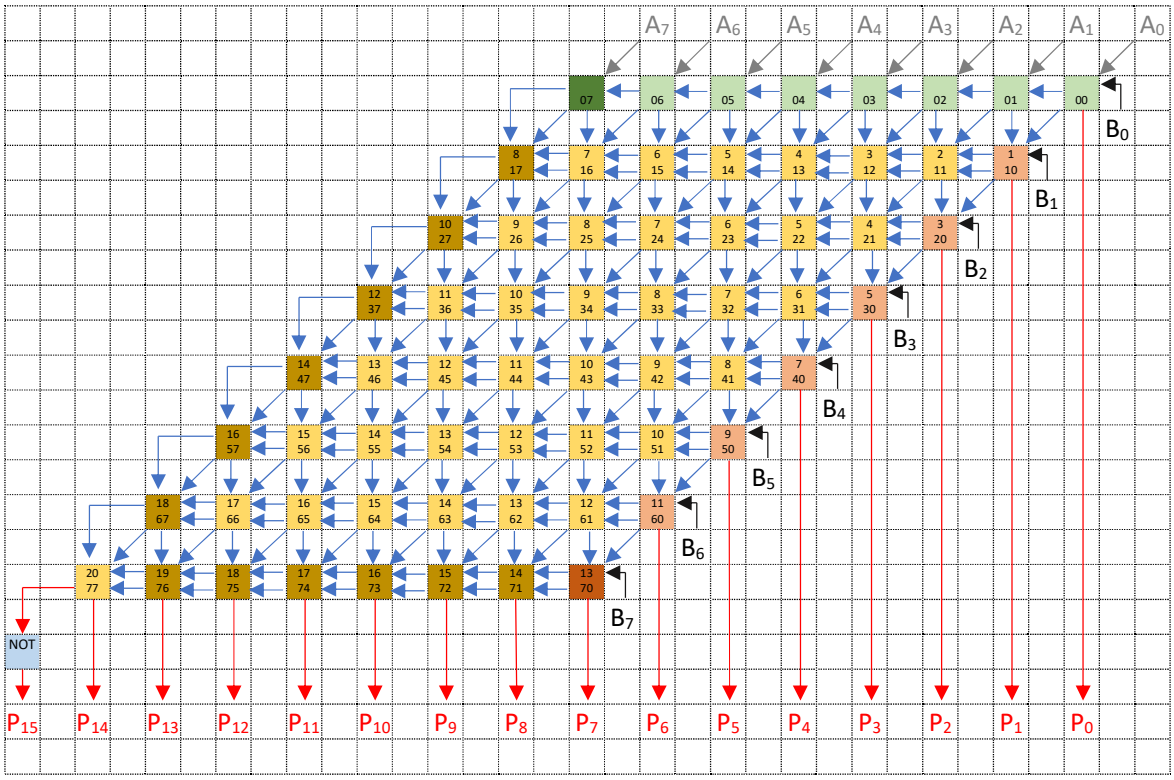
All the steps will run one by one starting from step-0 to step-20.

All the blocks with their indices written in a particular step will run in parallel at the same time. For Example, in step-5, the half adder/ full adder of blocks with indices (1,5), (2,3), (3,1) will compute their output simultaneously.

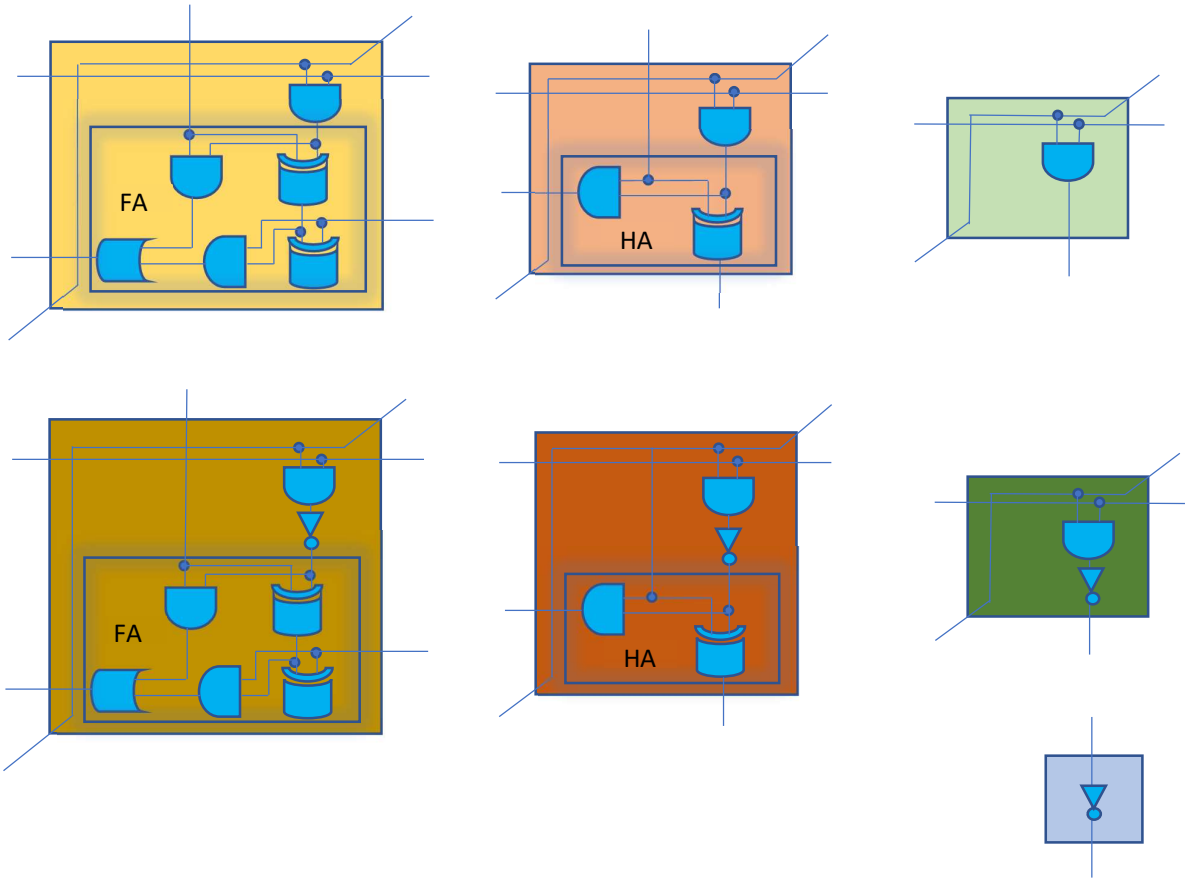
The 21 steps have been divided into 4 groups which represent the four major 'For loops' in the code.

This procedure is expanded symmetrically (the indices in the four major 'for' loops in the code are changed accordingly) to easily obtain higher order multipliers like 16 bit, 32 bit and so on.

2.2. Architecture of 8-bit Array Multiplier for signed numbers:



Inner structure of each block in the architecture (colour coded):



Labelling of each block is similar to the architecture for unsigned numbers.

Multiplicand: $A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0$ **(signed)**

Multiplier: $B_7\ B_6\ B_5\ B_4\ B_3\ B_2\ B_1\ B_0$ (signed)

Product: $P_{15}\ P_{14}\ P_{13}\ P_{12}\ P_{11}\ P_{10}\ P_9\ P_8\ P_7\ P_6\ P_5\ P_4\ P_3\ P_2\ P_1\ P_0$ (signed)

Flow of the Architecture:

Step 0: The AND gates in each block will compute their results. For example, the AND gate in the block (4,3) will compute the value of B_4A_3. Then, the result of some selected AND gates will be inverted using NOT gates. The blocks containing these selected AND gates have been coloured with a darker shade of the colour compared to the architecture for unsigned numbers in previous section.

Step 1:	10						
Step 2:	11						
Step 3:	12	20					
Step 4:	13	21					
Step 5:	14	22	30				
Step 6:	15	23	31				

Step 7:	16	24	32	40			
Step 8:	17	25	33	41			
Step 9:		26	34	42	50		
Step 10:		27	35	43	51		
Step 11:			36	44	52	60	
Step 12:			37	45	53	61	
Step 13:				46	54	62	70
Step 14:				47	55	63	71

Step 15:					56	64	72
Step 16:					57	65	73
Step 17:						66	74
Step 18:						67	75
Step 19:							76
Step 20:							77

The orange indices in steps 1, 3, 5 ,7, 9, 11, 13 represent the orange blocks in the architecture (containing Half Adders) and the yellow indices represent the yellow blocks in the architecture (containing Full Adders). The blocks with indices (0,7), (1,7), (2,7), (3,7), (4,7), (5,7), (6,7), (7,0), (7,1), (7,2), (7,3), (7,4), (7,5), (7,6) have an extra NOT gate attached to the AND gate. These are represented with a darker shade in the architecture.

All the steps will run one by one starting from step-0 to step-20.

All the blocks with their indices written in a particular step will run in parallel at the same time. For Example, in step-5, blocks with indices (1,5), (2,3), (3,1) will compute their output simultaneously.

The 21 steps have been divided into 4 groups which represent the four major 'for loops' in the code.

This procedure is expanded symmetrically (the indices in the four major 'for loops' in the code are changed accordingly) to easily obtain higher order multipliers like 16 bit, 32 bit and so on.

NOTE-1: This architecture is similar to the architecture for unsigned numbers but it has additional 15 (i.e.

2*(k-1)+1, where k is the number of bits in the multiplicand) NOT gates.

NOTE-2: If the multiplier or multiplicand or both are negative numbers, then the 2's complement representation of the number has to be given as inputs to the multiplier.

2.3. Architecture of 4 Bit Booth Multiplier

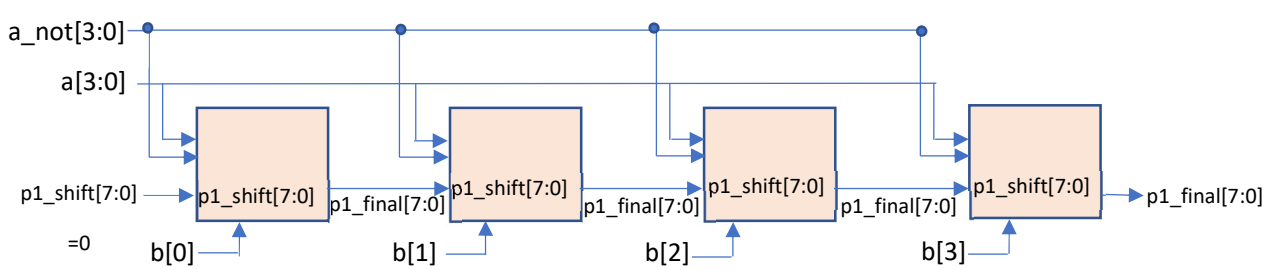
The architecture for a 4-bit Booth Multiplier is displayed below. This structure can be symmetrically expanded to obtain Multiplier for numbers with higher number of bits (like 8, 16, 32).

There are two conditions involved in this architecture (used in multiplexers).

a. The value of b[i] decides whether the multiplicand (a) will be passed to the adder or its complement.
b. The value of (b[i] ^ z) decides whether the multiplicand (or its complement) should be added or just a shift operation is needed.

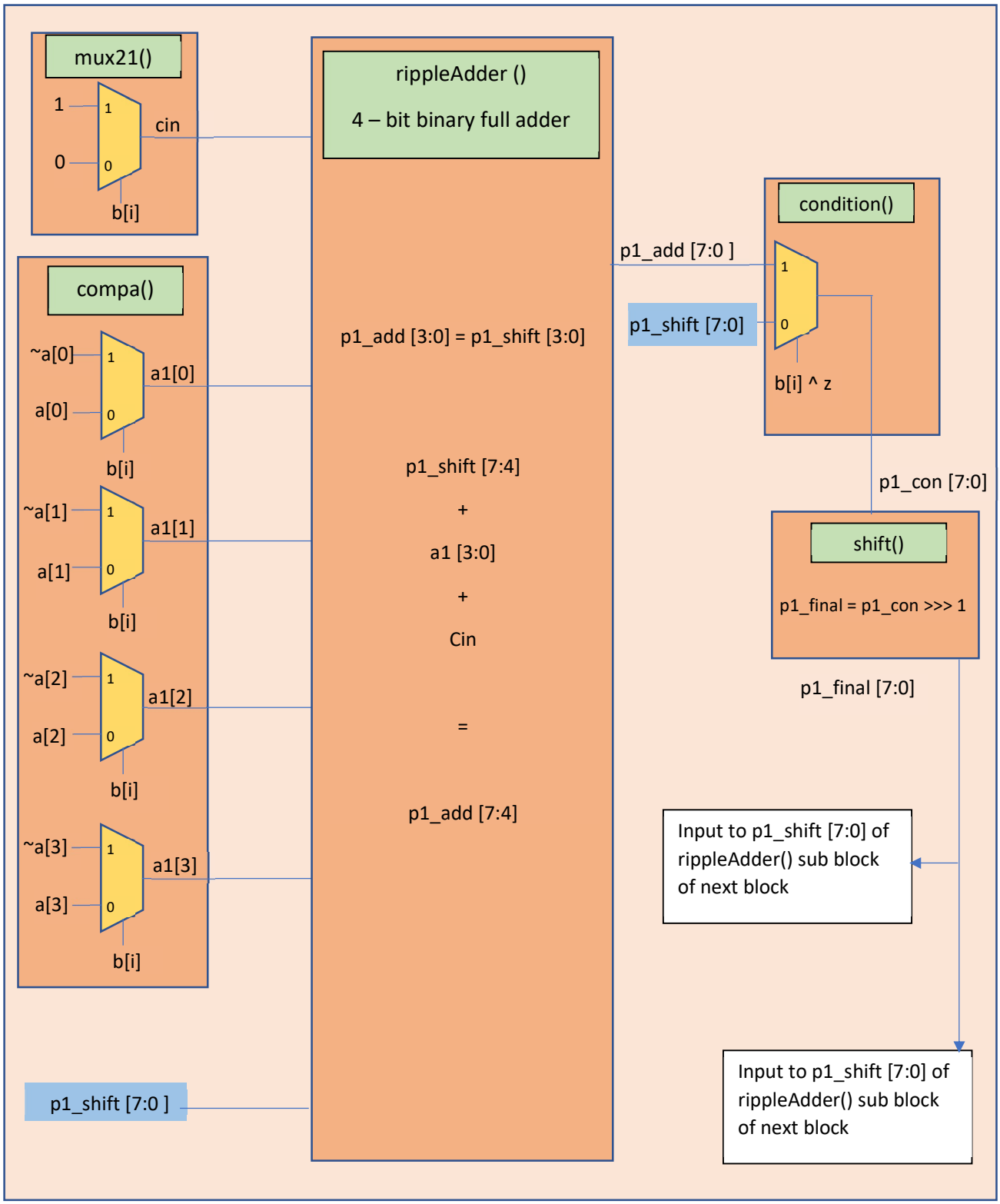
Bit Priorities

b[0]	z	a1_variants
0	0	0*a = 0
0	1	1*a = a1
1	0	-1*a = a1_not
1	1	0*a = 0



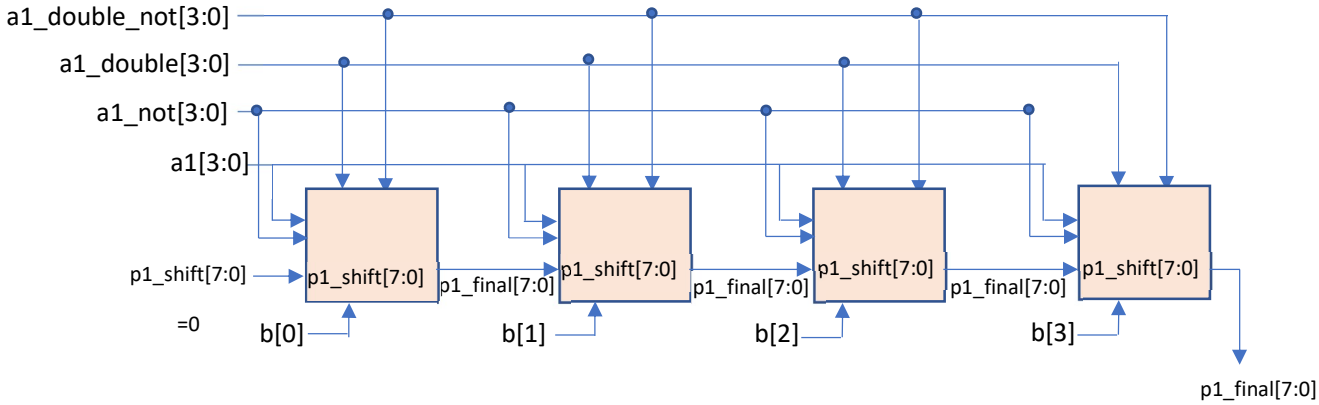
The inner structure of the block corresponding to b[i] is displayed below.

Inner structure of each block:



There will be 4 such blocks corresponding to the four bits of b[i], (i = 0, 1, 2, 3). We can add more such blocks to multiply numbers with higher number of bits. Moreover, if either of the multiplier or multiplicand are negative numbers, then their 2's complement should be feeded as input to the Multiplier.

2.4. Architecture of 4-Bit Modified Booth Multiplier



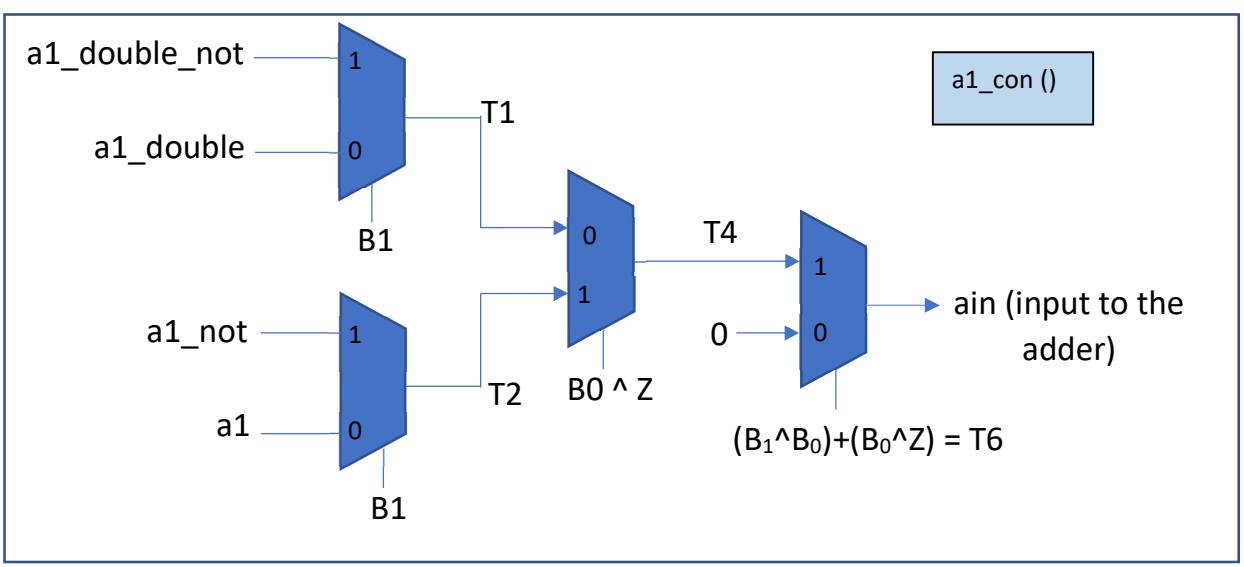
Bit Priorities

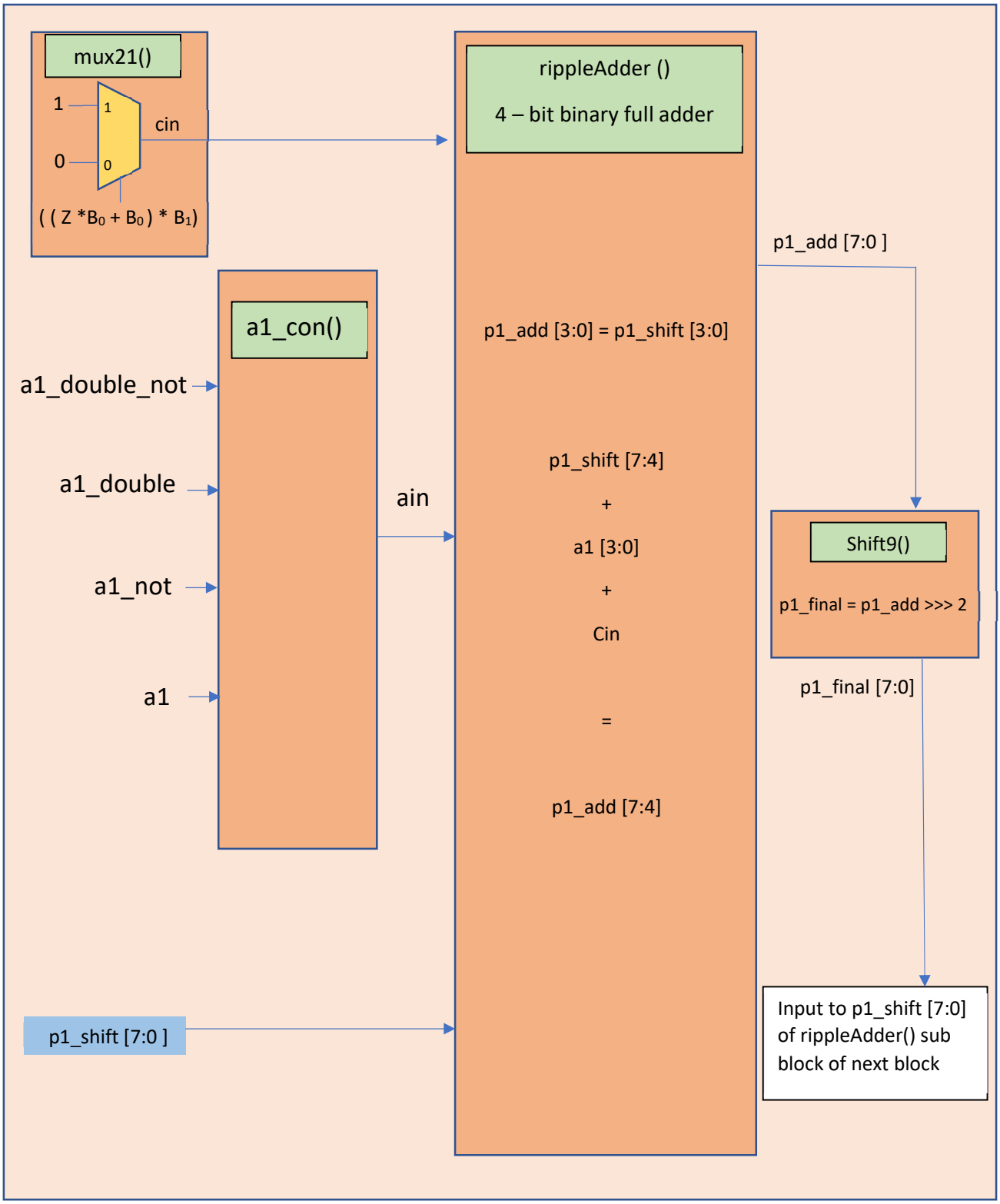
b[1]	b[0]	z	A1_variants0
0	0	0	0*a = 0
0	0	1	1*a = a1
0	1	0	1*a = a1
0	1	1	2*a = a1_double
1	0	0	-2*a = al_double_not
1	0	1	-1*a = a1_not
1	1	0	-1*a = a1_not
1	1	1	0*a = 0

** Table is for first block

** The A1_variants are calculated in the module a1_variants().

NOTE: cin =1 iff (($\overline{Z}$ * B_0 + $\overline{B}_0$) * B_1) =1



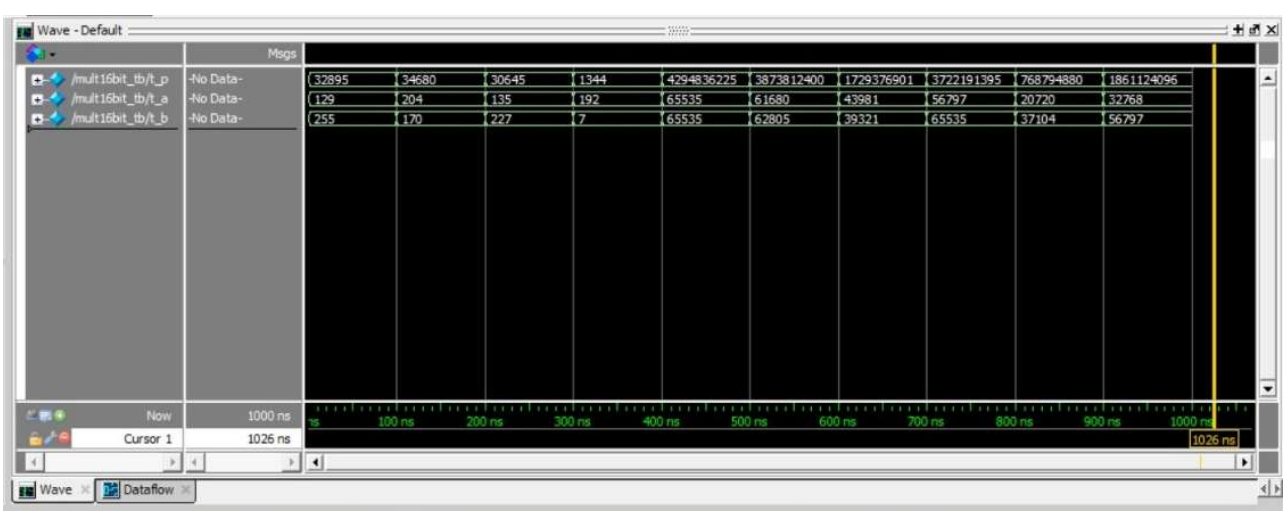

mux21()
1
1
0
0
cin
((Z *B0 + B0) * B1)
rippleAdder ()
4 – bit binary full adder
p1_add [7:0]
a1_con()
p1_add [3:0] = p1_shift [3:0]
a1_double_not
a1_double
a1_not
a1
ain
p1_shift [7:4]
+
a1 [3:0]
+
Cin
=
p1_add [7:4]
Shift9()
p1_final = p1_add >>> 2
p1_final [7:0]
p1_shift [7:0]
Input to p1_shift [7:0] of rippleAdder() sub block of next block

3. Verilog Codes and Simulation results

3.1 Array multiplier (16-bit unsigned numbers)

```verilog
module mult16bit(a,b,p);
//inputs
input [15:0]a,b;
//outputs
output reg [31:0]p;

//wires
reg w[15:0][15:0];
reg s[15:0][15:0];
reg c[15:0][15:0];

integer i,j,m,n,l,x,g,h;
always@*
begin

//AND GATES
for (i=0; i<16; i=i+1)
begin
    for (j=0; j<16; j=j+1)
    begin
        w[i][j] = (a[j]&b[i]);
    end
end

//-----------------------------------------------------------------
//CHANGE OF TERMINOLOGY
for (j=0; j<16; j=j+1)
begin
    s[0][j] = w[0][j];
end

c[0][15] = 0;

//-----------------------------------------------------------------
//m=0;
//while(m<5)
l=1;
for (m=0; m<13; m=m+2)
begin
    i=1;
    j=m;
    for (n=1 ; n<l; n=n+1)
    begin
        s[i][j] = (w[i][j] ^ s[i-1][j+1] ^ c[i][j-1]);
        c[i][j] = ((w[i][j]&s[i-1][j+1]) | (w[i][j]&c[i][j-1]) | (s[i-1][j+1]&c[i][j-1]));
        i=i+1;
        j=j-2;
    end
    s[i][j] = (w[i][j] ^ s[i-1][j+1]);
    c[i][j] = (w[i][j] & s[i-1][j+1]);

    g=1;
    h=m+1;
    for (x=1 ; x<l+1; x=x+1)
    begin
        s[g][h] = (w[g][h] ^ s[g-1][h+1] ^ c[g][h-1]);
        c[g][h] = ((w[g][h]&s[g-1][h+1]) | (w[g][h]&c[g][h-1]) | (s[g-1][h+1]&c[g][h-1]));
        g=g+1;
        h=h-2;
    end
    l=l+1;
//  m=m+2;
end

//-----------------------------------------------------------------
//m=1;
//while(m<5)
for (m=1; m<9; m=m+1)
begin
    i=m;
    j=14;
    for (n=1 ; n<8; n=n+1)
    begin
        s[i][j] = (w[i][j] ^ s[i-1][j+1] ^ c[i][j-1]);
        c[i][j] = ((w[i][j]&s[i-1][j+1]) | (w[i][j]&c[i][j-1]) | (s[i-1][j+1]&c[i][j-1]));
        i=i+1;
        j=j-2;
    end

    s[i][j] = (s[i-1][j+1]^w[i][j]);
    c[i][j] = (s[i-1][j+1]&w[i][j]);

    g=m;
    h=15;
    s[g][h] = (w[g][h] ^ c[g-1][h] ^ c[g][h-1]);
    c[g][h] = ((w[g][h]&c[g-1][h]) | (w[g][h]&c[g][h-1]) | (c[g-1][h]&c[g][h-1]));
    g=g+1;
    h=h-2;
```

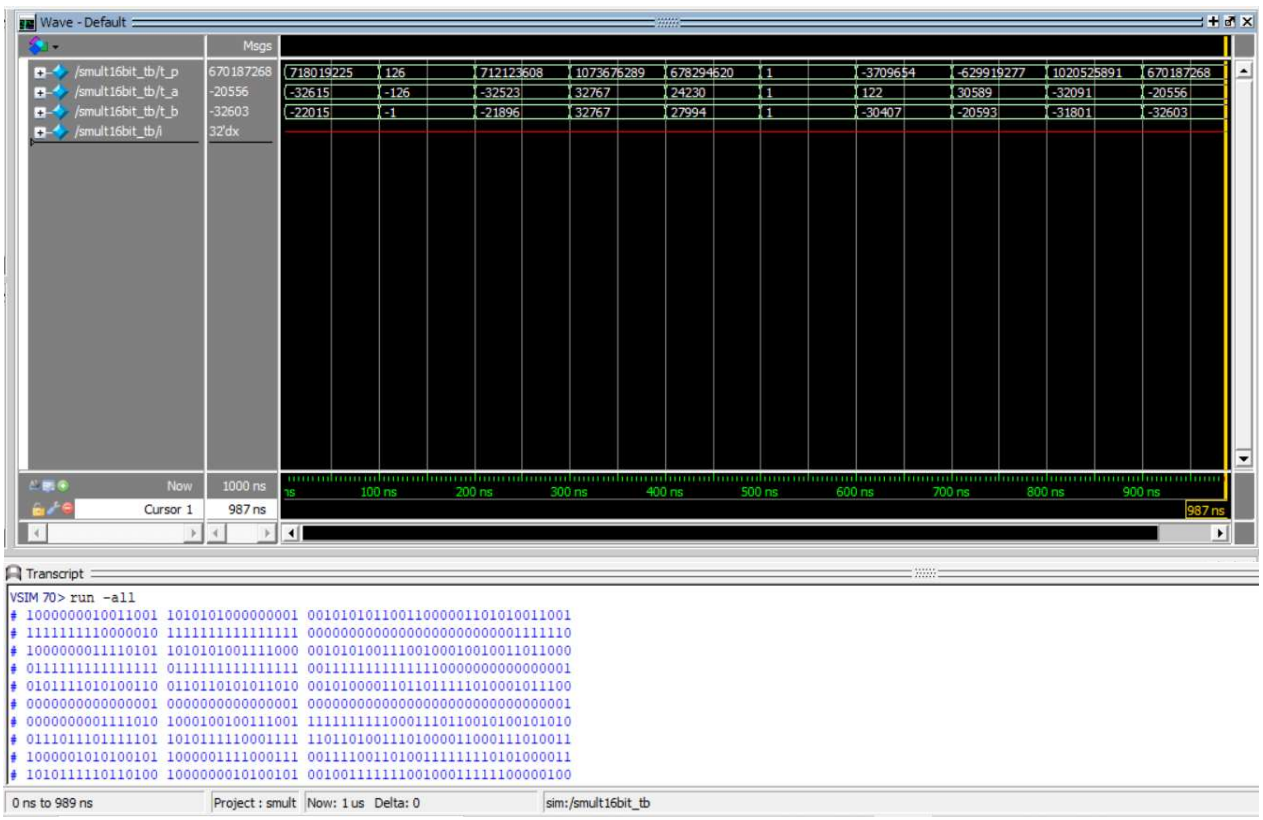
```
    for (x=1 ; x<8; x=x+1)
    begin
        s[g][h] = (w[g][h] ^ s[g-1][h+1] ^ c[g][h-1]);
        c[g][h] = ((w[g][h]&s[g-1][h+1]) | (w[g][h]&c[g][h-1]) | (s[g-1][h+1]&c[g][h-1]));
        g=g+1;
        h=h-2;
    end
//  m=m+1;
end

//-----------------------------------------------------------------------
//m=9;
//while(m<16)
l=7;
for (m=9; m<16; m=m+1)
begin
    i=m;
    j=14;
    for (n=1 ; n<l+1; n=n+1)
    begin
        s[i][j] = (w[i][j] ^ s[i-1][j+1] ^ c[i][j-1]);
        c[i][j] = ((w[i][j]&s[i-1][j+1]) | (w[i][j]&c[i][j-1]) | (s[i-1][j+1]&c[i][j-1]));
        i=i+1;
        j=j-2;
    end

    g=m;
    h=15;
    s[g][h] = (w[g][h] ^ c[g-1][h] ^ c[g][h-1]);
    c[g][h] = ((w[g][h]&c[g-1][h]) | (w[g][h]&c[g][h-1]) | (c[g-1][h]&c[g][h-1]));
    g=g+1;
    h=h-2;

    for (x=1 ; x<l; x=x+1)
    begin
        s[g][h] = (w[g][h] ^ s[g-1][h+1] ^ c[g][h-1]);
        c[g][h] = ((w[g][h]&s[g-1][h+1]) | (w[g][h]&c[g][h-1]) | (s[g-1][h+1]&c[g][h-1]));
        g=g+1;
        h=h-2;
    end
    l=l-1;
//  m=m+1;
end

//-----------------------------------------------------------------------
//OUTPUT ASSIGNMENTS
p[0] = w[0][0];
for (i=1; i<16; i=i+1)
begin
//assign p[i] = s[i][0];
p[i] = s[i][0];
end

for (i=1; i<16; i=i+1)
begin
//assign p[i+15] = s[15][i];
p[i+15] = s[15][i];
end

//assign p[31] = c[15][15];
p[31] = c[15][15];

end
endmodule
```



3.2 Array multiplier (16-bit signed numbers)

```
module smult16bit(a,b,p);
//inputs
input [15:0]a,b;
//outputs
output reg [31:0]p;

//wires
reg w[15:0][15:0];
reg s[15:0][15:0];
reg c[15:0][15:0];

integer i,j,m,n,l,x,g,h;
integer k=16;       //change this and the register sizes accordingly
always@*
begin

//AND GATES
for (i=0; i<k; i=i+1)
begin
    for (j=0; j<k; j=j+1)
    begin
        w[i][j] = (a[i]&b[j]);
    end
end

for (i=0; i<(k-1); i=i+1)
begin
    w[i][k-1] = (~w[i][k-1]);
end

for (j=0; j<(k-1); j=j+1)
begin
    w[k-1][j] = (~w[k-1][j]);
end

//-----------------------------------------------------------------------
//CHANGE OF TERMINOLOGY
for (j=1; j<k; j=j+1)
begin
    s[0][j] = w[0][j];
end

c[0][k-1] = 1;

//-----------------------------------------------------------------------
l=1;
for (m=0; m<(k-3); m=m+2)
begin
    i=1;
    j=m;
    for (n=1 ; n<l; n=n+1)
    begin
        s[i][j] = (w[i][j] ^ s[i-1][j+1] ^ c[i][j-1]);
        c[i][j] = ((w[i][j]&s[i-1][j+1]) | (w[i][j]&c[i][j-1]) | (s[i-1][j+1]&c[i][j-1]));
        i=i+1;
        j=j-2;
    end
    s[i][j] = (w[i][j] ^ s[i-1][j+1]);
    c[i][j] = (w[i][j] & s[i-1][j+1]);

    g=1;
    h=m+1;
    for (x=1 ; x<l+1; x=x+1)
    begin
        s[g][h] = (w[g][h] ^ s[g-1][h+1] ^ c[g][h-1]);
        c[g][h] = ((w[g][h]&s[g-1][h+1]) | (w[g][h]&c[g][h-1]) | (s[g-1][h+1]&c[g][h-1]));
        g=g+1;
        h=h-2;
    end
    l=l+1;
end

//-----------------------------------------------------------------------
for (m=1; m<(k/2+1); m=m+1)
begin
    i=m;
    j=(k-2);
    for (n=1 ; n<(k/2); n=n+1)
    begin
        s[i][j] = (w[i][j] ^ s[i-1][j+1] ^ c[i][j-1]);
        c[i][j] = ((w[i][j]&s[i-1][j+1]) | (w[i][j]&c[i][j-1]) | (s[i-1][j+1]&c[i][j-1]));
        i=i+1;
        j=j-2;
    end

    s[i][j] = (s[i-1][j+1]^w[i][j]);
    c[i][j] = (s[i-1][j+1]&w[i][j]);

    g=m;
    h=(k-1);
    s[g][h] = (w[g][h] ^ c[g-1][h] ^ c[g][h-1]);
    c[g][h] = ((w[g][h]&c[g-1][h]) | (w[g][h]&c[g][h-1]) | (c[g-1][h]&c[g][h-1]));
    g=g+1;
    h=h-2;

    for (x=1 ; x<(k/2); x=x+1)
    begin
        s[g][h] = (w[g][h] ^ s[g-1][h+1] ^ c[g][h-1]);
        c[g][h] = ((w[g][h]&s[g-1][h+1]) | (w[g][h]&c[g][h-1]) | (s[g-1][h+1]&c[g][h-1]));
        g=g+1;
        h=h-2;
    end
end

//-----------------------------------------------------------------------
l=(k/2-1);
for (m=(k/2+1); m<k; m=m+1)
begin
    i=m;
    j=(k-2);
```

```
    for (n=1 ; n<l+1; n=n+1)
    begin
        s[i][j] = (w[i][j] ^ s[i-1][j+1] ^ c[i][j-1]);
        c[i][j] = ((w[i][j]&s[i-1][j+1]) | (w[i][j]&c[i][j-1]) | (s[i-1][j+1]&c[i][j-1]));
        i=i+1;
        j=j-2;
    end

    g=m;
    h=(k-1);
    s[g][h] = (w[g][h] ^ c[g-1][h] ^ c[g][h-1]);
    c[g][h] = ((w[g][h]&c[g-1][h]) | (w[g][h]&c[g][h-1]) | (c[g-1][h]&c[g][h-1]));
    g=g+1;
    h=h-2;

    for (x=1 ; x<l; x=x+1)
    begin
        s[g][h] = (w[g][h] ^ s[g-1][h+1] ^ c[g][h-1]);
        c[g][h] = ((w[g][h]&s[g-1][h+1]) | (w[g][h]&c[g][h-1]) | (s[g-1][h+1]&c[g][h-1]));
        g=g+1;
        h=h-2;
    end
    l=l-1;
end

//-----------------------------------------------------------------------------
//OUTPUT ASSIGNMENTS
p[0] = w[0][0];
for (i=1; i<k; i=i+1)
begin
p[i] = s[i][0];
end

for (i=1; i<k; i=i+1)
begin
p[i+k-1] = s[k-1][i];
end

p[2*k-1] = (c[k-1][k-1]^1);

end
endmodule
```



3.2. Booth Multiplier (16-bit signed numbers)

```
module booth16bit(a, b, p);
    input [15:0] a, b;
    output reg [31:0] p;

    wire z;
    wire [31:0] pl_shift;
    wire [31:0] pl_final [15:0];

    assign pl_shift = 32'b0;
    assign z = 1'b0;

    partialProduct pp00 (.a(a), .b(b[0]), .pl_shift(pl_shift), .zin(z), .pl_final(pl_final[0]));
    partialProduct pp01 (.a(a), .b(b[1]), .pl_shift(pl_final[0]), .zin(b[0]), .pl_final(pl_final[1]));
    partialProduct pp02 (.a(a), .b(b[2]), .pl_shift(pl_final[1]), .zin(b[1]), .pl_final(pl_final[2]));
    partialProduct pp03 (.a(a), .b(b[3]), .pl_shift(pl_final[2]), .zin(b[2]), .pl_final(pl_final[3]));
    partialProduct pp04 (.a(a), .b(b[4]), .pl_shift(pl_final[3]), .zin(b[3]), .pl_final(pl_final[4]));
    partialProduct pp05 (.a(a), .b(b[5]), .pl_shift(pl_final[4]), .zin(b[4]), .pl_final(pl_final[5]));
    partialProduct pp06 (.a(a), .b(b[6]), .pl_shift(pl_final[5]), .zin(b[5]), .pl_final(pl_final[6]));
    partialProduct pp07 (.a(a), .b(b[7]), .pl_shift(pl_final[6]), .zin(b[6]), .pl_final(pl_final[7]));
    partialProduct pp08 (.a(a), .b(b[8]), .pl_shift(pl_final[7]), .zin(b[7]), .pl_final(pl_final[8]));
    partialProduct pp09 (.a(a), .b(b[9]), .pl_shift(pl_final[8]), .zin(b[8]), .pl_final(pl_final[9]));
    partialProduct pp10 (.a(a), .b(b[10]), .pl_shift(pl_final[9]), .zin(b[9]), .pl_final(pl_final[10]));
    partialProduct pp11 (.a(a), .b(b[11]), .pl_shift(pl_final[10]), .zin(b[10]), .pl_final(pl_final[11]));
    partialProduct pp12 (.a(a), .b(b[12]), .pl_shift(pl_final[11]), .zin(b[11]), .pl_final(pl_final[12]));
    partialProduct pp13 (.a(a), .b(b[13]), .pl_shift(pl_final[12]), .zin(b[12]), .pl_final(pl_final[13]));
    partialProduct pp14 (.a(a), .b(b[14]), .pl_shift(pl_final[13]), .zin(b[13]), .pl_final(pl_final[14]));
    partialProduct pp15 (.a(a), .b(b[15]), .pl_shift(pl_final[14]), .zin(b[14]), .pl_final(pl_final[15]));
    // more modules of partialProduct can be added to multiply numbers with more number of bits

    always@(pl_final[15])
    begin
    p[31:0] = pl_final[15];
    end
endmodule

module partialProduct(a, b, pl_shift, zin, pl_final);
    input [15:0] a;
    input b;
    input [31:0] pl_shift;
    input zin;
    output [31:0] pl_final;

    wire cin;
    wire temp;
    wire [15:0] al;
    wire cout;
    wire [31:0] pl_add;
    wire [31:0] pl_con;
    wire [31:0] pl_final;

    mux21 m0 (.Y(cin), .D0(1'b0), .D1(1'b1), .S(b));
    compa c0 (.a(a), .b(b), .al(al));

    rippleAdder r0 (.x(pl_shift), .y(al), .cin(cin), .sum(pl_add), .cout(cout));

    xor (temp, b, zin);

    condition con0 (.pl_add(pl_add), .pl_shift(pl_shift), .temp(temp), .pl_con(pl_con));
    shift s0 (.pl_con(pl_con), .pl_shift(pl_final));
endmodule

module shift(pl_con, pl_shift) ;
    input [31:0]pl_con;
    output [31:0]pl_shift;
    reg [31:0]pl_shift;

    always@(pl_con)
    begin
    pl_shift = (pl_con >>> 1);
    pl_shift[31] = pl_shift[30];
    end
endmodule

module rippleAdder(x, y, cin, sum, cout);
    input [31:0] x;
    input [15:0] y;
    input cin;
    output cout;
    output [31:0]sum;

    wire [15:0] c;

    begin
    assign sum[0] = x[0];
    assign sum[1] = x[1];
    assign sum[2] = x[2];
    assign sum[3] = x[3];
    assign sum[4] = x[4];
    assign sum[5] = x[5];
    assign sum[6] = x[6];
    assign sum[7] = x[7];
    assign sum[8] = x[8];
    assign sum[9] = x[9];
    assign sum[10] = x[10];
    assign sum[11] = x[11];
    assign sum[12] = x[12];
    assign sum[13] = x[13];
    assign sum[14] = x[14];
    assign sum[15] = x[15];

    fullAdder f1(.m(x[16]), .n(y[0]), .cin(cin), .sum(sum[16]), .cout(c[0]));
    fullAdder f2(.m(x[17]), .n(y[1]), .cin(c[0]), .sum(sum[17]), .cout(c[1]));
    fullAdder f3(.m(x[18]), .n(y[2]), .cin(c[1]), .sum(sum[18]), .cout(c[2]));
    fullAdder f4(.m(x[19]), .n(y[3]), .cin(c[2]), .sum(sum[19]), .cout(c[3]));
    fullAdder f5(.m(x[20]), .n(y[4]), .cin(c[3]), .sum(sum[20]), .cout(c[4]));
    fullAdder f6(.m(x[21]), .n(y[5]), .cin(c[4]), .sum(sum[21]), .cout(c[5]));
    fullAdder f7(.m(x[22]), .n(y[6]), .cin(c[5]), .sum(sum[22]), .cout(c[6]));
    fullAdder f8(.m(x[23]), .n(y[7]), .cin(c[6]), .sum(sum[23]), .cout(c[7]));
    fullAdder f9(.m(x[24]), .n(y[8]), .cin(c[7]), .sum(sum[24]), .cout(c[8]));
    fullAdder f10(.m(x[25]), .n(y[9]), .cin(c[8]), .sum(sum[25]), .cout(c[9]));
```

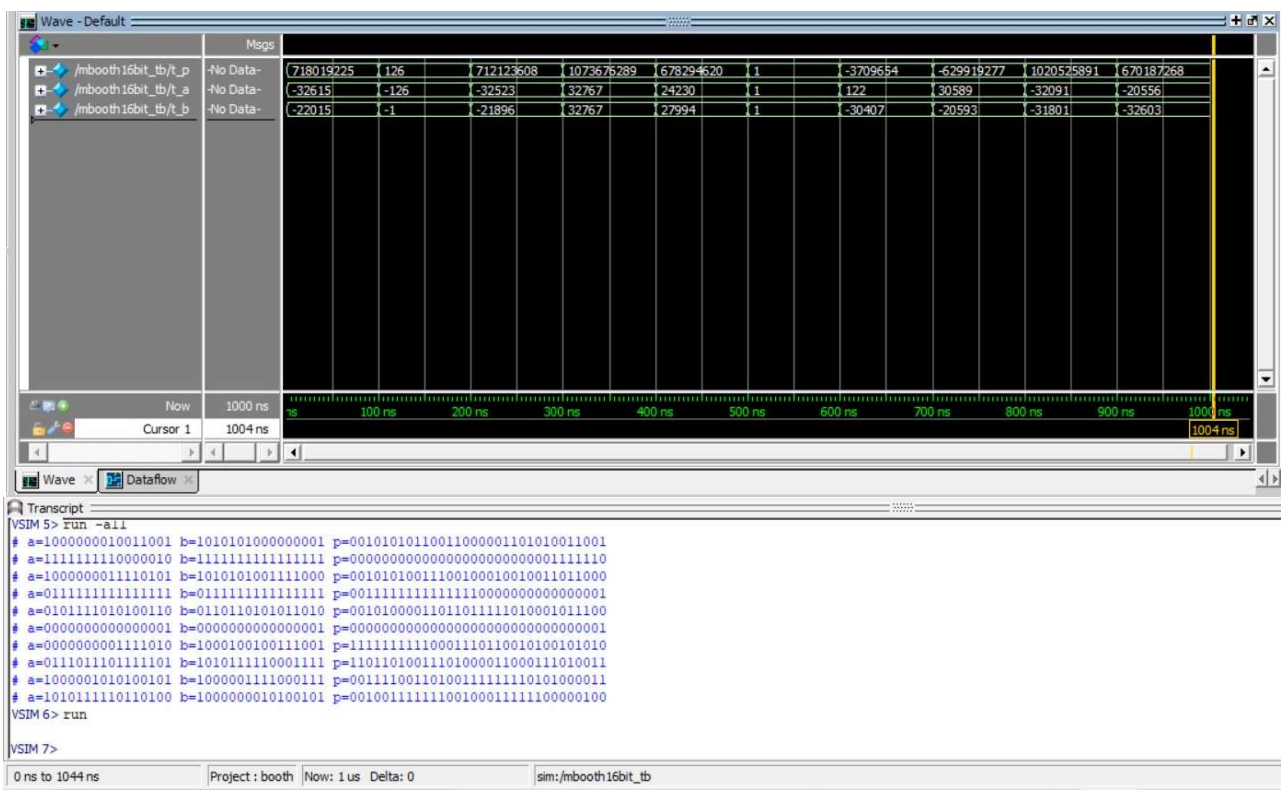
```
    fullAdder f11(.m(x[26]), .n(y[10]), .cin(c[9]), .sum(sum[26]), .cout(c[10]));
    fullAdder f12(.m(x[27]), .n(y[11]), .cin(c[10]), .sum(sum[27]), .cout(c[11]));
    fullAdder f13(.m(x[28]), .n(y[12]), .cin(c[11]), .sum(sum[28]), .cout(c[12]));
    fullAdder f14(.m(x[29]), .n(y[13]), .cin(c[12]), .sum(sum[29]), .cout(c[13]));
    fullAdder f15(.m(x[30]), .n(y[14]), .cin(c[13]), .sum(sum[30]), .cout(c[14]));
    fullAdder f16(.m(x[31]), .n(y[15]), .cin(c[14]), .sum(sum[31]), .cout(c[15]));

    assign cout = c[15];
    end
endmodule

module fullAdder(m, n, cin, sum, cout);
    input cin, m, n;
    output sum, cout;

    assign sum = (m^n^cin);
    assign cout = ((m&n)|(n&cin)|(m&cin));
endmodule
/*
module fullAdder(m, n, cin, sum, cout);
    input cin, m, n;
    output sum, cout;
    wire t1,t2,t3;

    xor x2 (t1, m, n);
    xor x3(sum, t1, cin);

    and a1 (t2, m, n);
    and a2 (t3, t1, cin);
    or  o1(cout, t1, t2);
endmodule
*/

// deciding whether to send a1 or -a1
module compa(a, b, a1);
    input [15:0] a;
    input b;
    output [15:0]a1;

    mux21_16 m16 (.Y(a1), .D0(a), .D1(~a), .S(b));

endmodule

module mux21_16 (Y, D0, D1, S);
    output [15:0] Y;
    input [15:0] D0, D1;
    input S;

    assign Y = (S ? D1 : D0);

/*  wire T1, T2, Sbar;
    and a3 (T1, D1, S);
    and a4 (T2, D0, Sbar);
    not n1 (Sbar, S);
    or o2 (Y, T1, T2);
*/
endmodule

module mux21_32 (Y, D0, D1, S);
    output [31:0] Y;
    input [31:0] D0, D1;
    input S;

    assign Y = (S ? D1 : D0);

/*  wire T1, T2, Sbar;
    and a3 (T1, D1, S);
    and a4 (T2, D0, Sbar);
    not n1 (Sbar, S);
    or o2 (Y, T1, T2);
*/
endmodule

module mux21(Y, D0, D1, S);
    output Y;
    input D0, D1, S;
    wire T1, T2, Sbar;

//  assign Y = (S ? D1 : D0);
    and a3 (T1, D1, S);
    and a4 (T2, D0, Sbar);
    not n1 (Sbar, S);
    or o2 (Y, T1, T2);
endmodule

module condition(p1_add, p1_shift, temp, p1_con);
    input [31:0]p1_add;
    input [31:0]p1_shift;
    input temp;
    output [31:0]p1_con;

    mux21_32 m32 (.Y(p1_con), .D0(p1_shift), .D1(p1_add), .S(temp));


endmodule

```

3.2. Modified Booth Multiplier (16-bit signed numbers)

```
module mbooth16bit(a, b, p);
    input [15:0] a, b;
    output reg [31:0] p;

    wire z;
    wire [32:0] p1_shift;
    wire [32:0] p1_final [7:0];
    wire [16:0] a1;
    wire [16:0] a1_not, a1_double, a1_double_not;

    assign a1[15:0] = a[15:0];
    assign a1[16] = a[15];
    assign p1_shift = 33'b0;
    assign z = 1'b0;

    a1_variants av1 (.a1(a1), .a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not));
    partialProduct pp1 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[1]), .b0(b[0]), .p1_shift(p1_shift), .zin(z), .p1_final(p1_final[0]));
    partialProduct pp2 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[3]), .b0(b[2]), .p1_shift(p1_final[0]), .zin(b[1]), .p1_final(p1_final[1]));
    partialProduct pp3 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[5]), .b0(b[4]), .p1_shift(p1_final[1]), .zin(b[3]), .p1_final(p1_final[2]));
    partialProduct pp4 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[7]), .b0(b[6]), .p1_shift(p1_final[2]), .zin(b[5]), .p1_final(p1_final[3]));
    partialProduct pp5 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[9]), .b0(b[8]), .p1_shift(p1_final[3]), .zin(b[7]), .p1_final(p1_final[4]));
    partialProduct pp6 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[11]), .b0(b[10]), .p1_shift(p1_final[4]), .zin(b[9]), .p1_final(p1_final[5]));
    partialProduct pp7 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[13]), .b0(b[12]), .p1_shift(p1_final[5]), .zin(b[11]), .p1_final(p1_final[6]));
    partialProduct pp8 (.a1(a1),.a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b[15]), .b0(b[14]), .p1_shift(p1_final[6]), .zin(b[13]), .p1_final(p1_final[7]));
    // more modules of partialProduct can be added to multiply numbers with more number of bits

    always@(p1_final[7])
    begin
    p[31:0] = p1_final[7];
    end
endmodule

// deciding whether to send a1 or -a1 or 2*a1 or -2*a1
module a1_variants (a1, a1_not, a1_double, a1_double_not);
    input [16:0] a1;
    output [16:0] a1_not, a1_double, a1_double_not;

    assign a1_not = ~a1;
    shift17 s170 (.a1(a1), .a1_double(a1_double));
    assign a1_double_not = ~a1_double;
endmodule

module partialProduct(a1, a1_not, a1_double, a1_double_not, b1, b0, p1_shift, zin, p1_final);
    input [16:0] a1, a1_not, a1_double, a1_double_not;
    input b0, b1;
    input [32:0] p1_shift;
    input zin;
    output [32:0] p1_final;

    wire cin;
    wire cin_con;
    wire [16:0] ain;
    wire [16:0] cout;
    wire [32:0] p1_add;
    wire [32:0] p1_final;

    cin_con cc0 (.b1(b1), .b0(b0), .zin(zin), .cin_con(cin_con));
    mux21 m00 (.Y(cin), .D0(1'b0), .D1(1'b1), .S(cin_con));
    a1_con a10 (.a1(a1), .a1_not(a1_not), .a1_double(a1_double), .a1_double_not(a1_double_not), .b1(b1), .b0(b0), .zin(zin), .ain(ain));
    rippleAdder r00 (.x(p1_shift), .y(ain), .cin(cin), .sum(p1_add), .cout(cout));
    shift9 s90 (.p1_add(p1_add), .p1_final(p1_final));
endmodule

module a1_con (a1, a1_not, a1_double, a1_double_not, b1, b0, zin, ain);
    input [16:0] a1, a1_not, a1_double, a1_double_not;
    input b0, b1, zin;
    output [16:0] ain;

    wire [16:0] t1, t2, t4;
    wire t3, t5, t6, t7;

    mux21_17 m10 (.Y(t1), .D0(a1_double), .D1(a1_double_not), .S(b1));
    mux21_17 m20 (.Y(t2), .D0(a1), .D1(a1_not), .S(b1));
    xor (t3, b0, zin);
    mux21_17 m30 (.Y(t4), .D0(t1), .D1(t2), .S(t3));
    xor (t5, b0, b1);
    xor (t6, b0, zin);
    or (t7, t5, t6);
    mux21_17 m40 (.Y(ain), .D0(17'b0), .D1(t4), .S(t7));
endmodule

// deciding whether cin will be 1 or 0
module cin_con (b1, b0, zin, cin_con);
    input b1, b0, zin;
    output cin_con;

    wire t1, t2, b0not, zinnot;

    not (b0not, b0);
    not (zinnot, zin);
    and (t1, b0, zinnot);
    or (t2, b0not, t1);
    and (cin_con, b1, t2);
endmodule

module shift9 (p1_add, p1_final) ;
    input [32:0]p1_add;
    output [32:0]p1_final;

    assign p1_final[30:0] = p1_add[32:2];
    assign p1_final[31] = p1_add[32];
    assign p1_final[32] = p1_add[32];
endmodule

module shift17 (a1, a1_double);
    input [16:0] a1;
    output [16:0] a1_double;

    assign a1_double[16:1] = a1[15:0];
    assign a1_double[0] = 1'b0;
endmodule

// module of some faster adder can be added here instead of ripple carry adder.
module rippleAdder (x, y, cin, sum, cout);
    input [32:0] x;
    input [16:0] y;
    input cin;
    output [16:0]cout;
    output [32:0]sum;
```

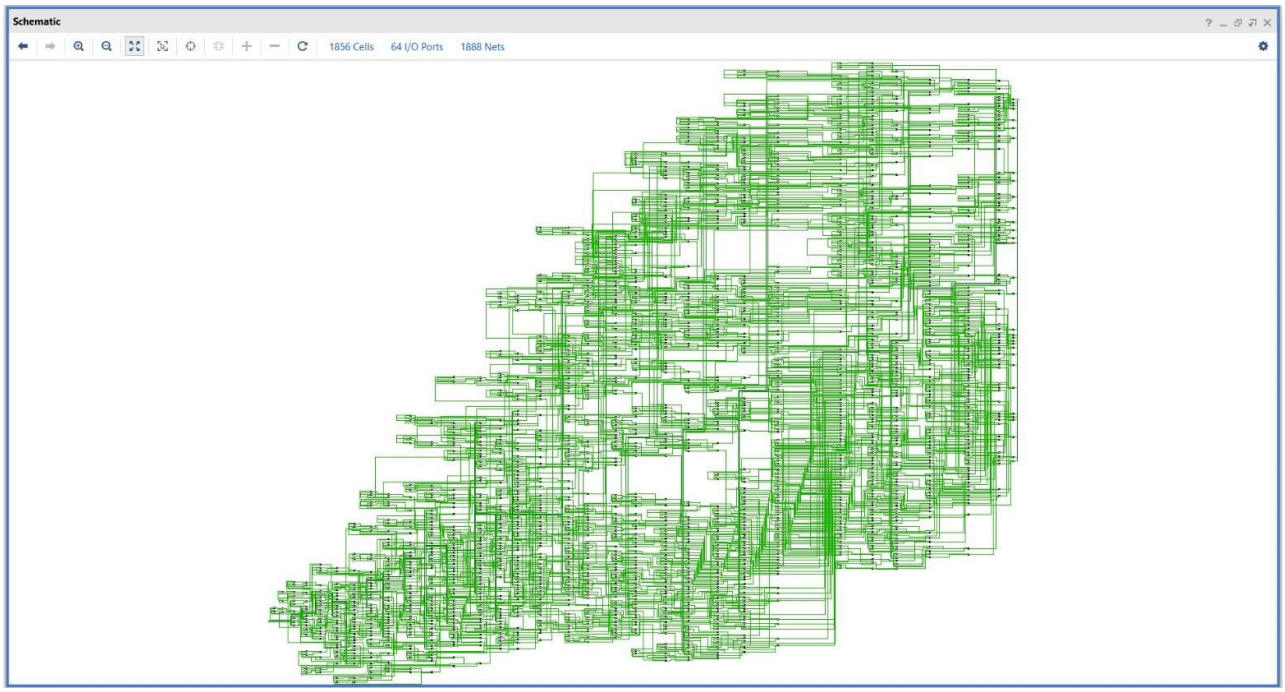
```
    begin
    assign sum[15:0] = x[15:0];
    fullAdder f1(.m(x[16]), .n(y[0]), .cin(cin), .sum(sum[16]), .cout(cout[0]));
    fullAdder f2(.m(x[17]), .n(y[1]), .cin(cout[0]), .sum(sum[17]), .cout(cout[1]));
    fullAdder f3(.m(x[18]), .n(y[2]), .cin(cout[1]), .sum(sum[18]), .cout(cout[2]));
    fullAdder f4(.m(x[19]), .n(y[3]), .cin(cout[2]), .sum(sum[19]), .cout(cout[3]));
    fullAdder f5(.m(x[20]), .n(y[4]), .cin(cout[3]), .sum(sum[20]), .cout(cout[4]));
    fullAdder f6(.m(x[21]), .n(y[5]), .cin(cout[4]), .sum(sum[21]), .cout(cout[5]));
    fullAdder f7(.m(x[22]), .n(y[6]), .cin(cout[5]), .sum(sum[22]), .cout(cout[6]));
    fullAdder f8(.m(x[23]), .n(y[7]), .cin(cout[6]), .sum(sum[23]), .cout(cout[7]));
    fullAdder f9(.m(x[24]), .n(y[8]), .cin(cout[7]), .sum(sum[24]), .cout(cout[8]));
    fullAdder f10(.m(x[25]), .n(y[9]), .cin(cout[8]), .sum(sum[25]), .cout(cout[9]));
    fullAdder f11(.m(x[26]), .n(y[10]), .cin(cout[9]), .sum(sum[26]), .cout(cout[10]));
    fullAdder f12(.m(x[27]), .n(y[11]), .cin(cout[10]), .sum(sum[27]), .cout(cout[11]));
    fullAdder f13(.m(x[28]), .n(y[12]), .cin(cout[11]), .sum(sum[28]), .cout(cout[12]));
    fullAdder f14(.m(x[29]), .n(y[13]), .cin(cout[12]), .sum(sum[29]), .cout(cout[13]));
    fullAdder f15(.m(x[30]), .n(y[14]), .cin(cout[13]), .sum(sum[30]), .cout(cout[14]));
    fullAdder f16(.m(x[31]), .n(y[15]), .cin(cout[14]), .sum(sum[31]), .cout(cout[15]));
    fullAdder f17(.m(x[32]), .n(y[16]), .cin(cout[15]), .sum(sum[32]), .cout(cout[16]));
    end
endmodule

module fullAdder (m, n, cin, sum, cout);
    input cin, m, n;
    output sum, cout;

    assign sum = (m^n^cin);
    assign cout = ((m&n)|(n&cin)|(m&cin));
endmodule
/*
module fullAdder(m, n, cin, sum, cout);
    input cin, m, n;
    output sum, cout;
    wire t1,t2,t3;

    xor x2 (t1, m, n);
    xor x3(sum, t1, cin);

    and a1 (t2, m, n);
    and a2 (t3, t1, cin);
    or  o1(cout, t1, t2);
endmodule
*/

module mux21(Y, D0, D1, S); // when inputs to be multiplexed are single bit long
    output Y;
    input D0, D1, S;
    wire T1, T2, Sbar;

//  assign Y = (S ? D1 : D0);
    and a3 (T1, D1, S);
    and a4 (T2, D0, Sbar);
    not n1 (Sbar, S);
    or o2 (Y, T1, T2);
endmodule

module mux21_17 (Y, D0, D1, S); // when inputs to be multiplexed are of 17 bits
    output [16:0] Y;
    input [16:0] D0, D1;
    input S;

    assign Y = (S ? D1 : D0);
endmodule
```

** same testbench is used for Booth multiplier and Modified Booth Multiplier.

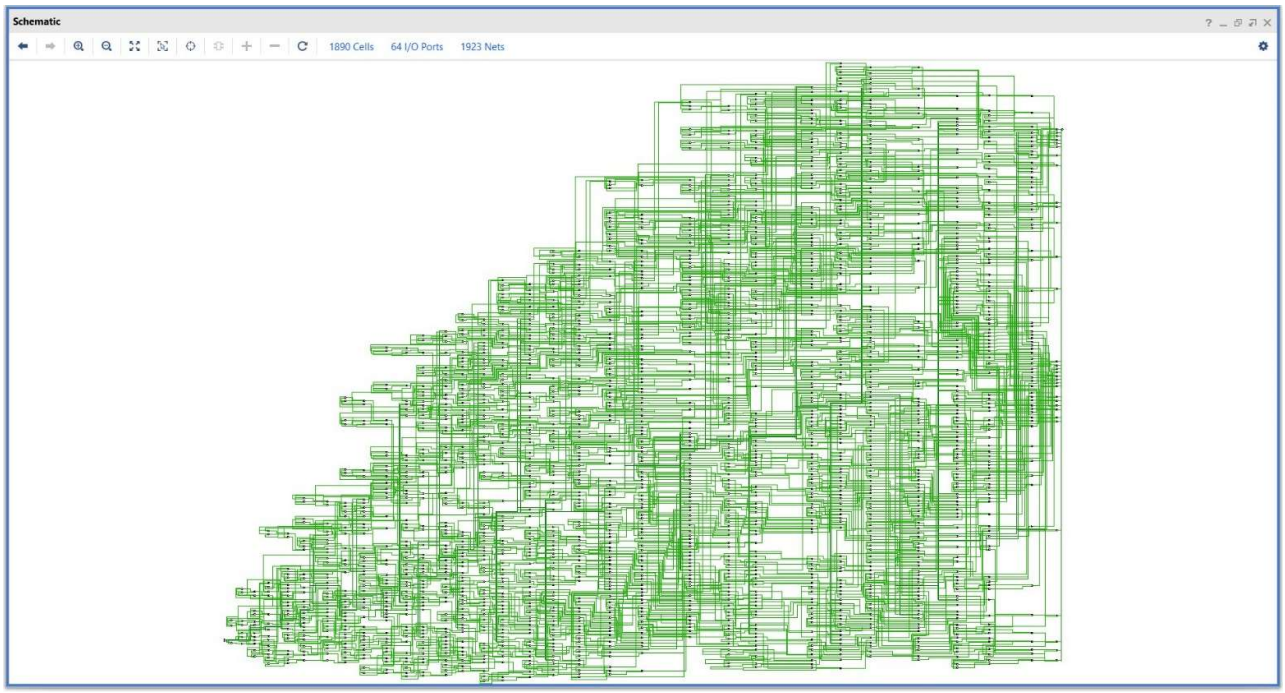


4. RTL Analysis (Elaborated designs post synthesis)

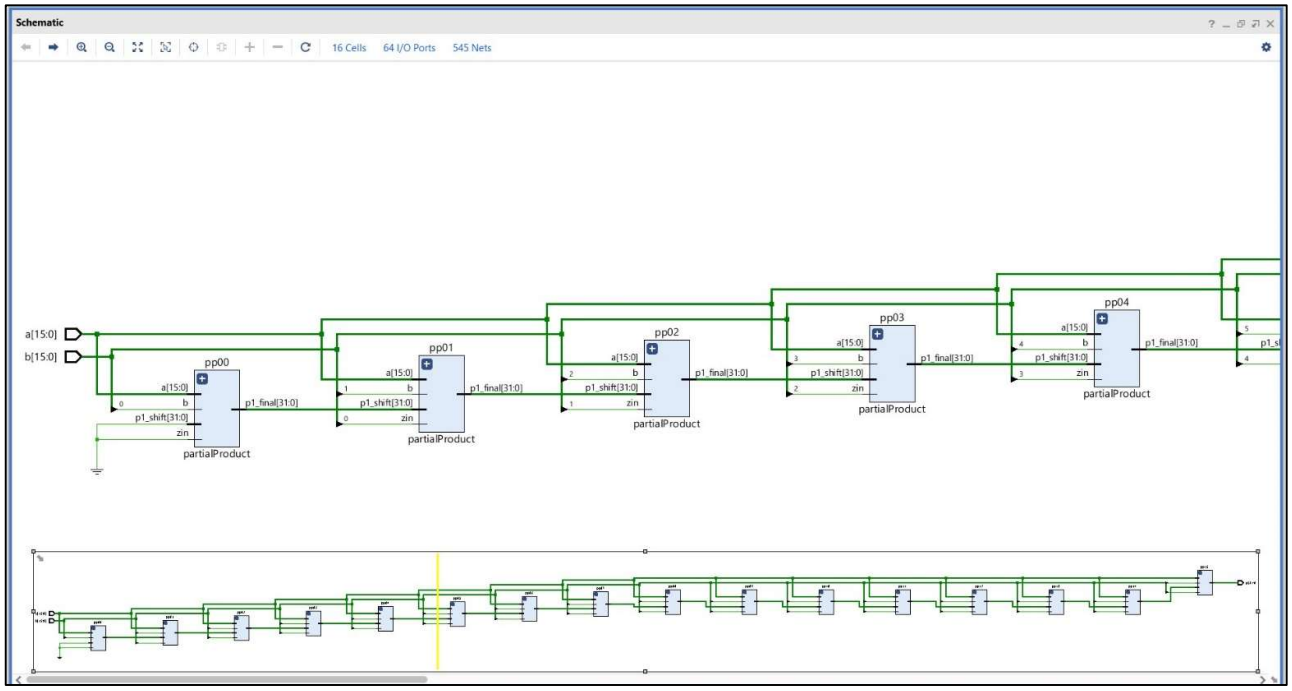
4.1. Array multiplier (16-bit unsigned numbers)

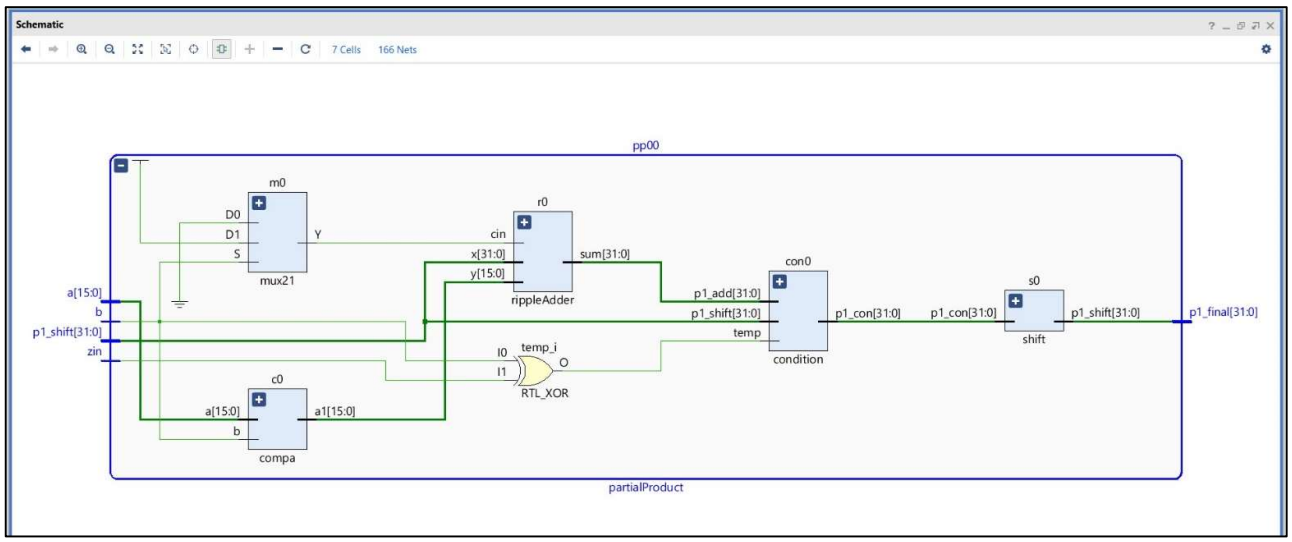


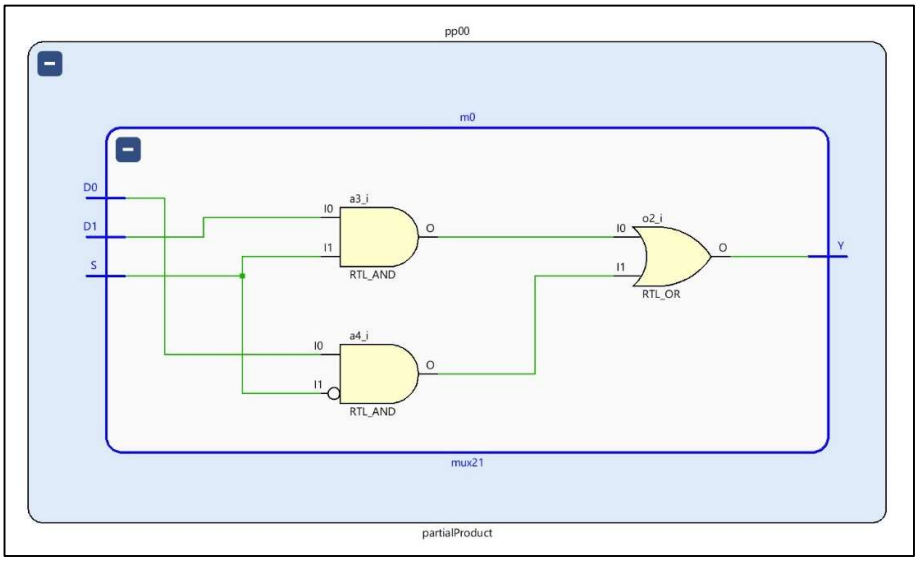
4.2. Array multiplier (16-bit signed numbers)

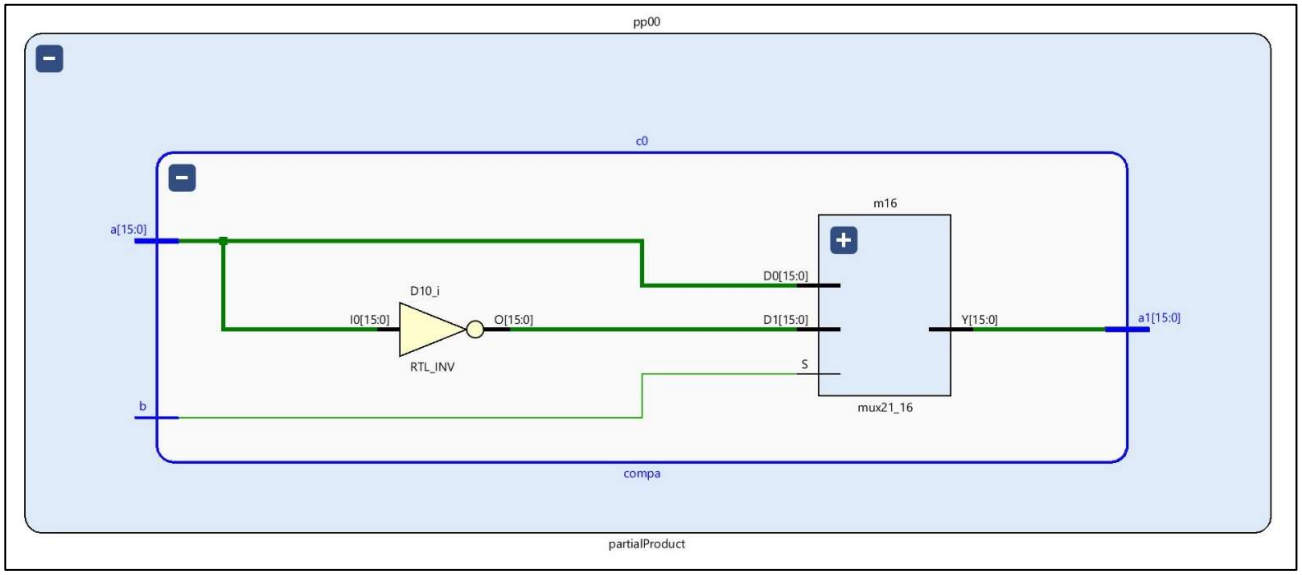


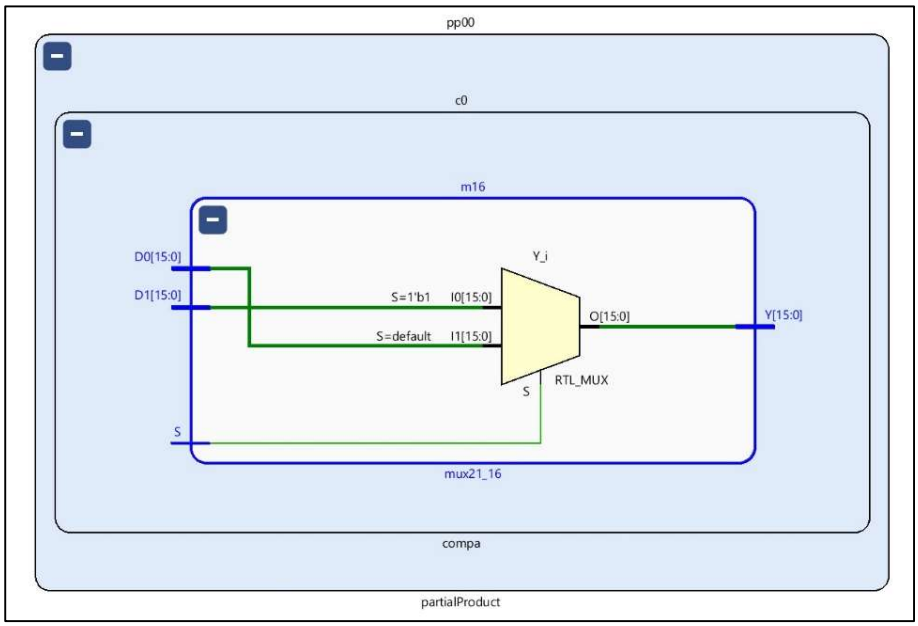
4.3. Booth Multiplier (16-bit signed numbers)

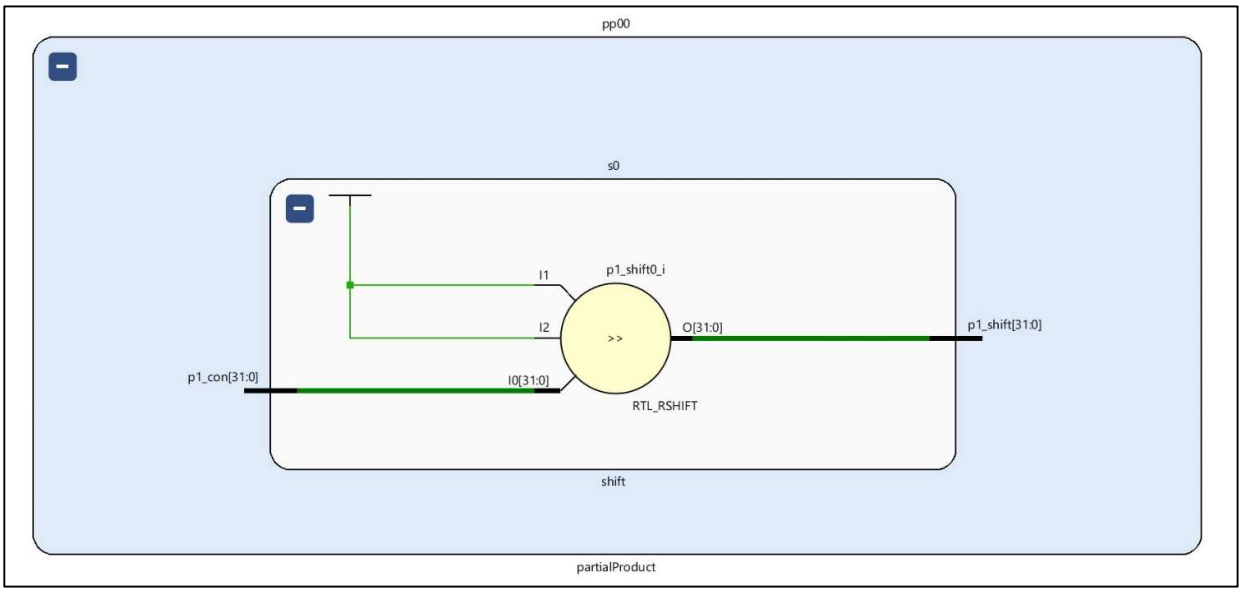


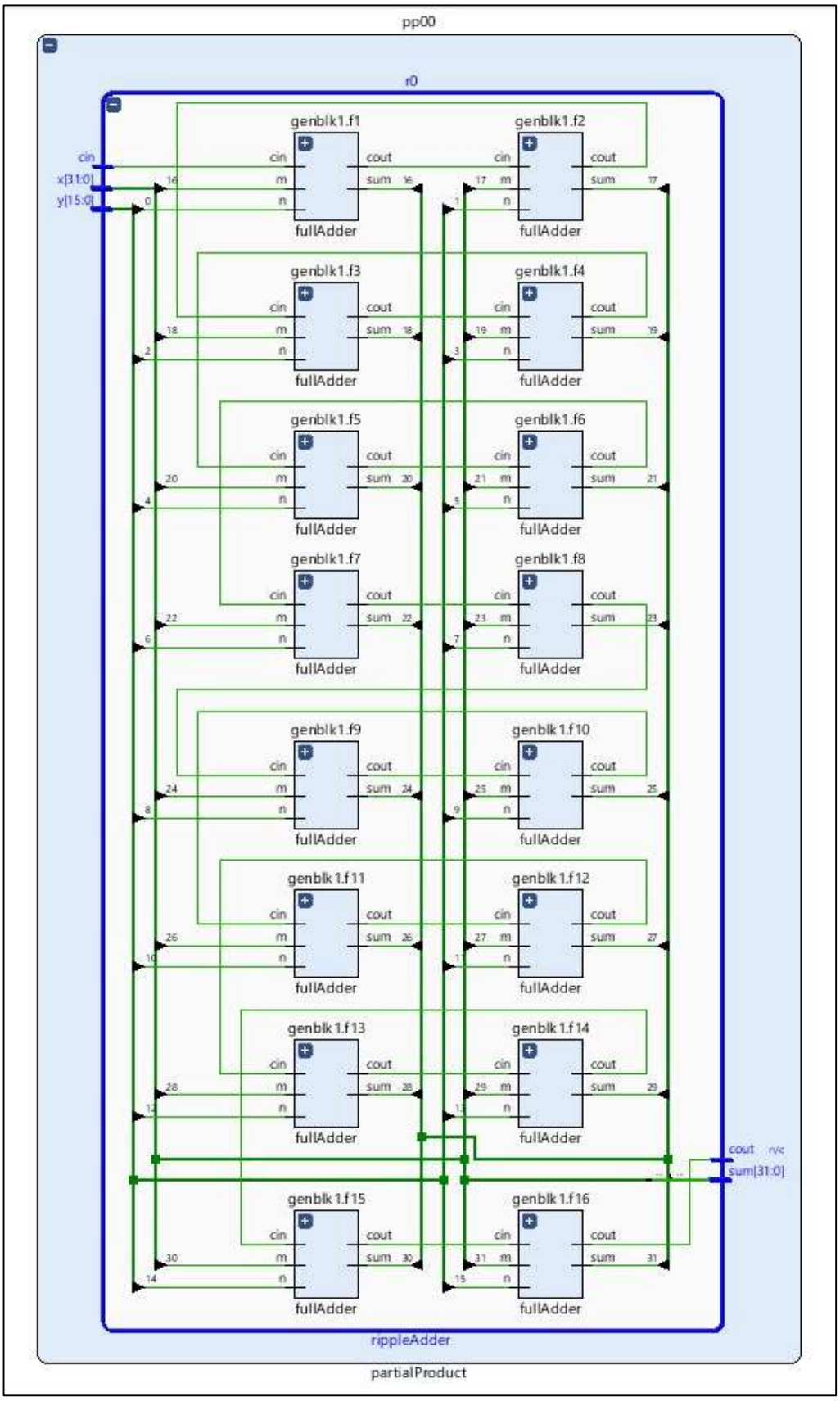


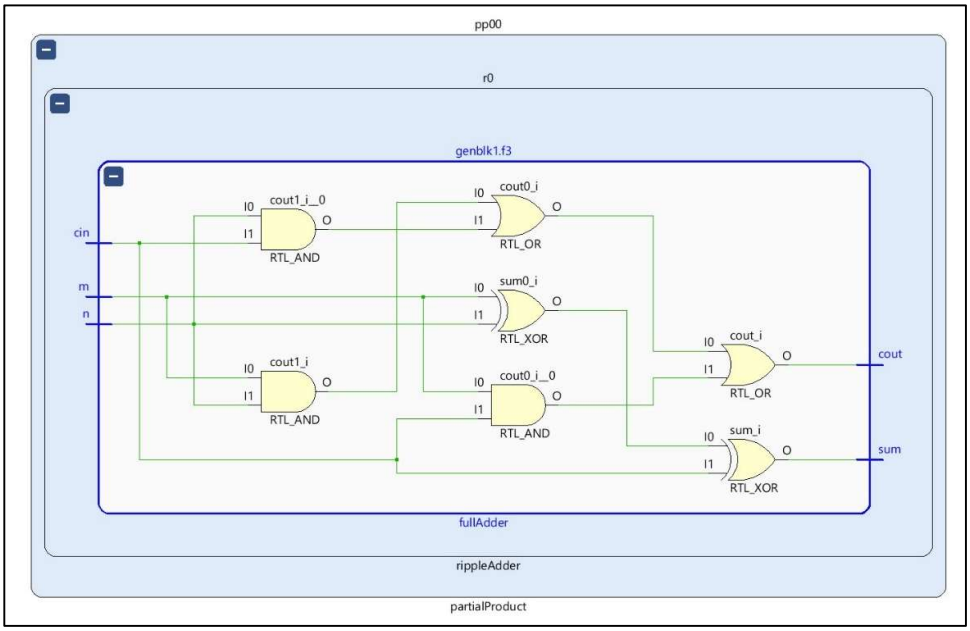


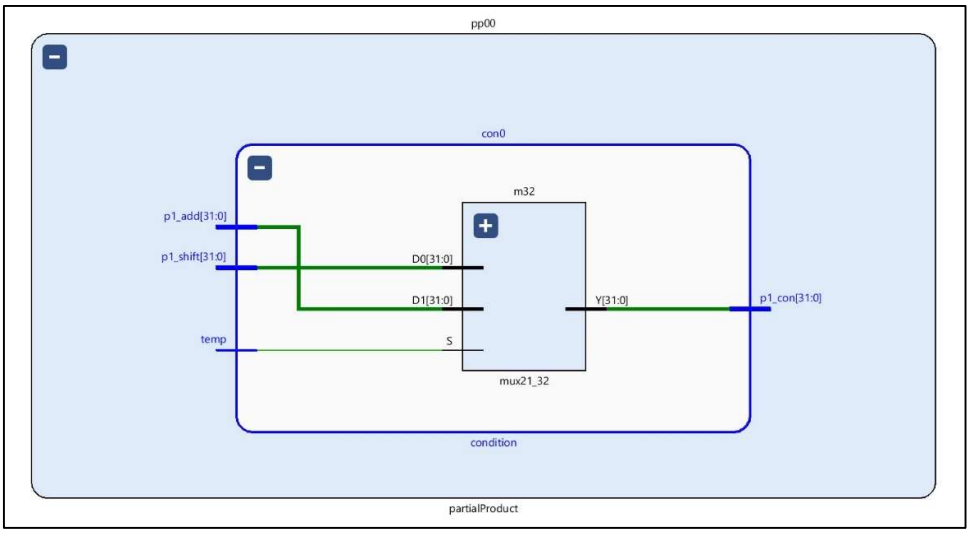

pp00
c0
m16
a[15:0]
D0[15:0]
D10_i
I0[15:0]
O[15:0]
D1[15:0]
Y[15:0]
a1[15:0]
RTL_INV
S
b
mux21_16
compa
partialProduct

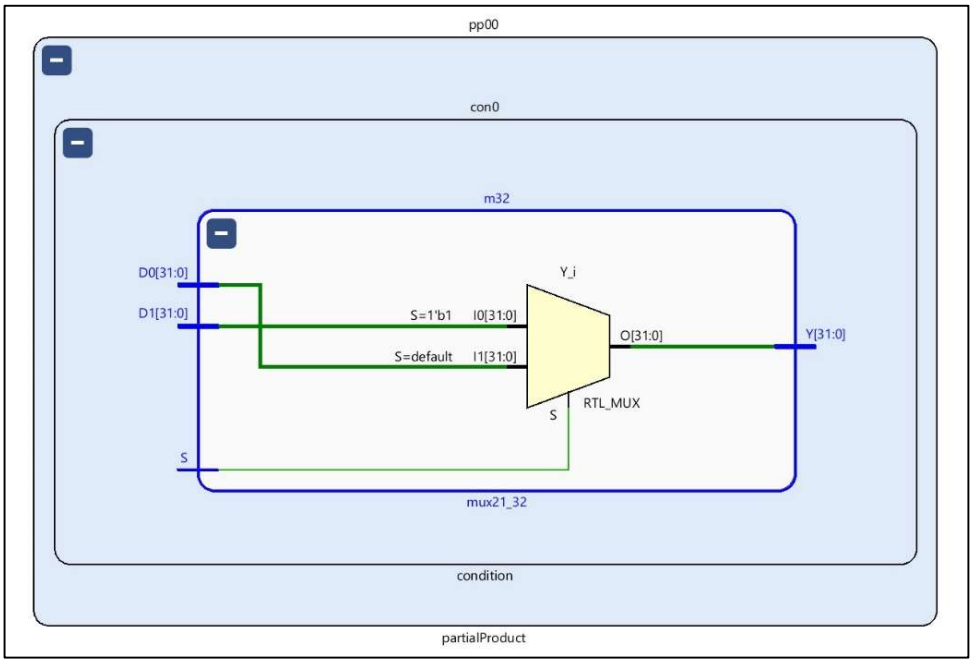

pp00
c0
m16
D0[15:0]
D1[15:0]
Y_i
S=1'b1
I0[15:0]
S=default
I1[15:0]
O[15:0]
Y[15:0]
RTL_MUX
S
S
mux21_16
compa
partialProduct


pp00
s0
I1
p1_shift0_i
I2
>>
O[31:0]
p1_shift[31:0]
p1_con[31:0]
I0[31:0]
RTL_RSHIFT
shift
partialProduct

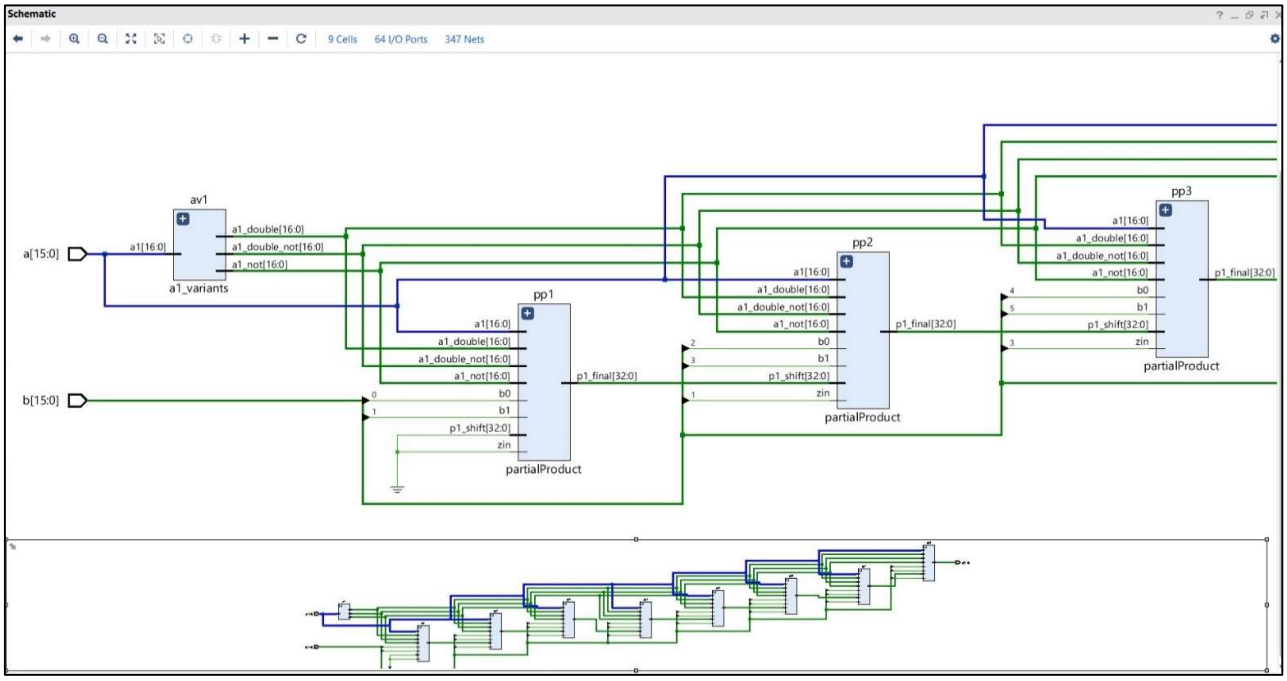

pp00
r0
cin
x[31:0]
y[15:0]
genblk1.f1
genblk1.f2
genblk1.f3
genblk1.f4
genblk1.f5
genblk1.f6
genblk1.f7
genblk1.f8
genblk1.f9
genblk1.f10
genblk1.f11
genblk1.f12
genblk1.f13
genblk1.f14
genblk1.f15
genblk1.f16
fullAdder
cin
cout
m
n
sum
cout
sum[31:0]
rippleAdder
partialProduct

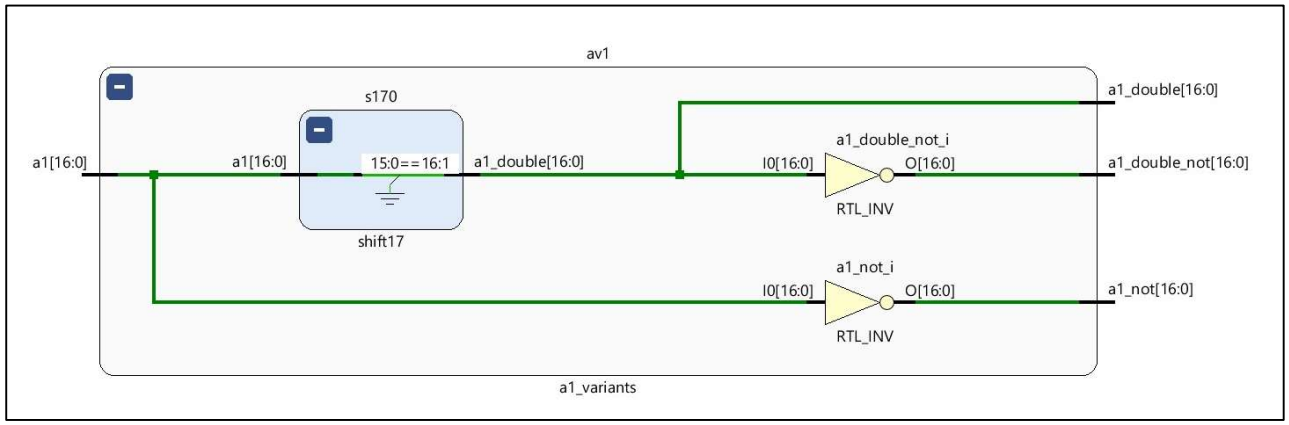

pp00
r0
genblk1.f3
cin
m
n
cout1_i__0
RTL_AND
cout0_i
RTL_OR
sum0_i
RTL_XOR
cout1_i
RTL_AND
cout0_i__0
RTL_AND
cout_i
RTL_OR
sum_i
RTL_XOR
cout
sum
fullAdder
rippleAdder
partialProduct

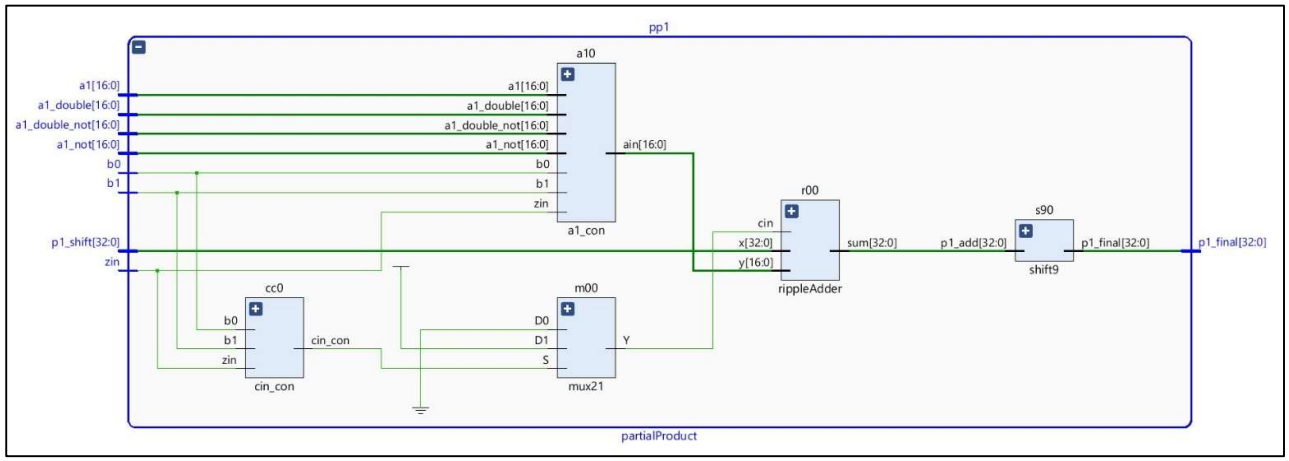

pp00
con0
m32
p1_add[31:0]
p1_shift[31:0]
temp
D0[31:0]
D1[31:0]
S
Y[31:0]
p1_con[31:0]
mux21_32
condition
partialProduct

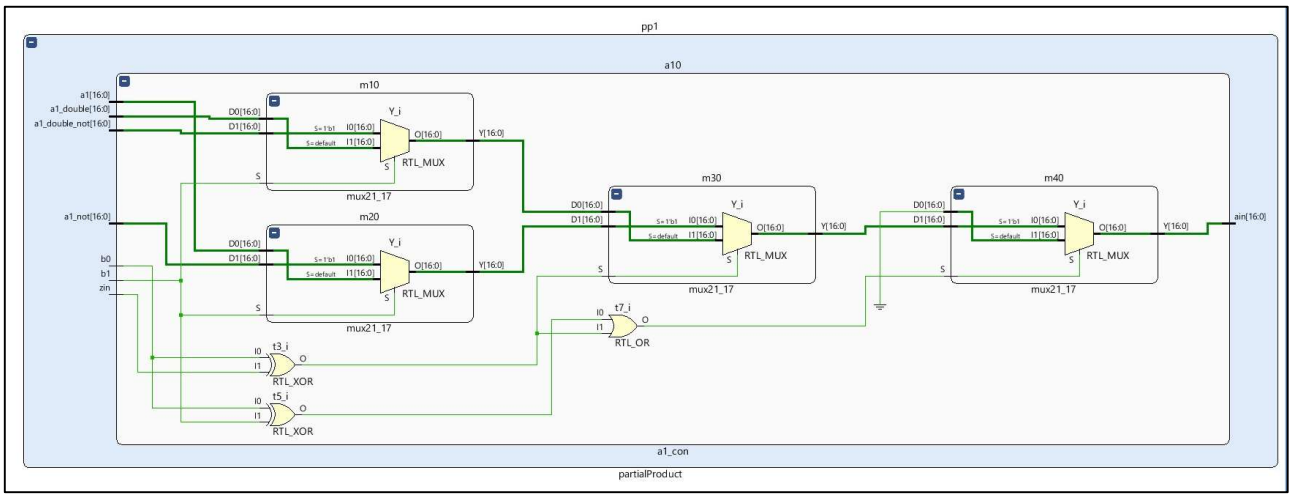

pp00
con0
m32
D0[31:0]
D1[31:0]
S
Y_i
S=1'b1
I0[31:0]
S=default
I1[31:0]
O[31:0]
Y[31:0]
RTL_MUX
mux21_32
condition
partialProduct

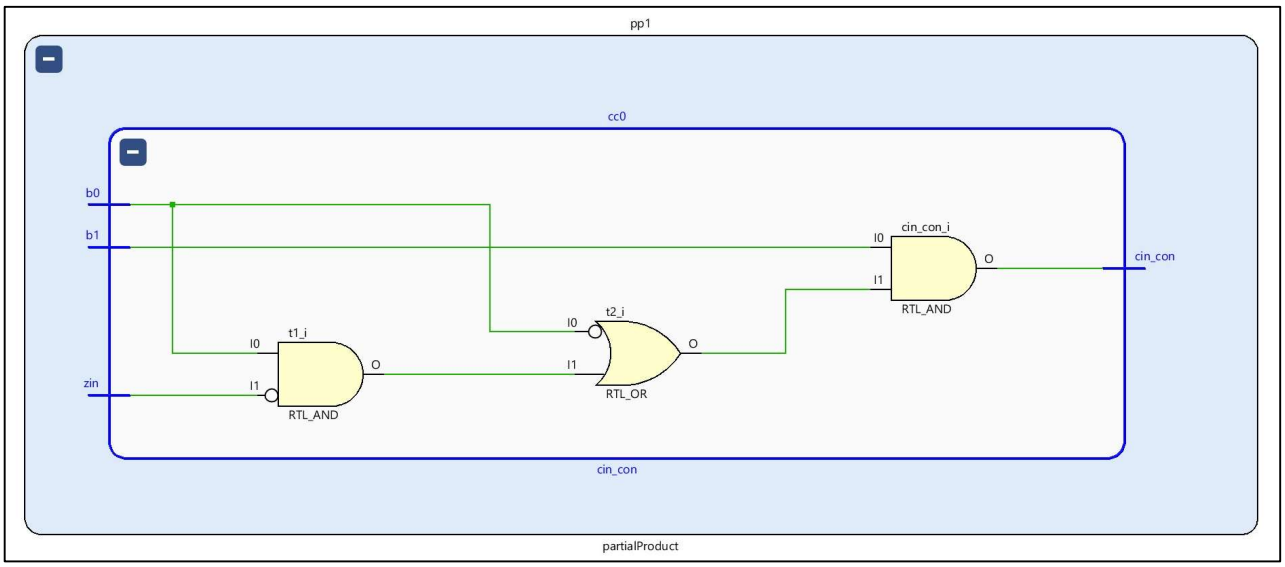
4.4. Modified Booth Multiplier (16-bit signed numbers)

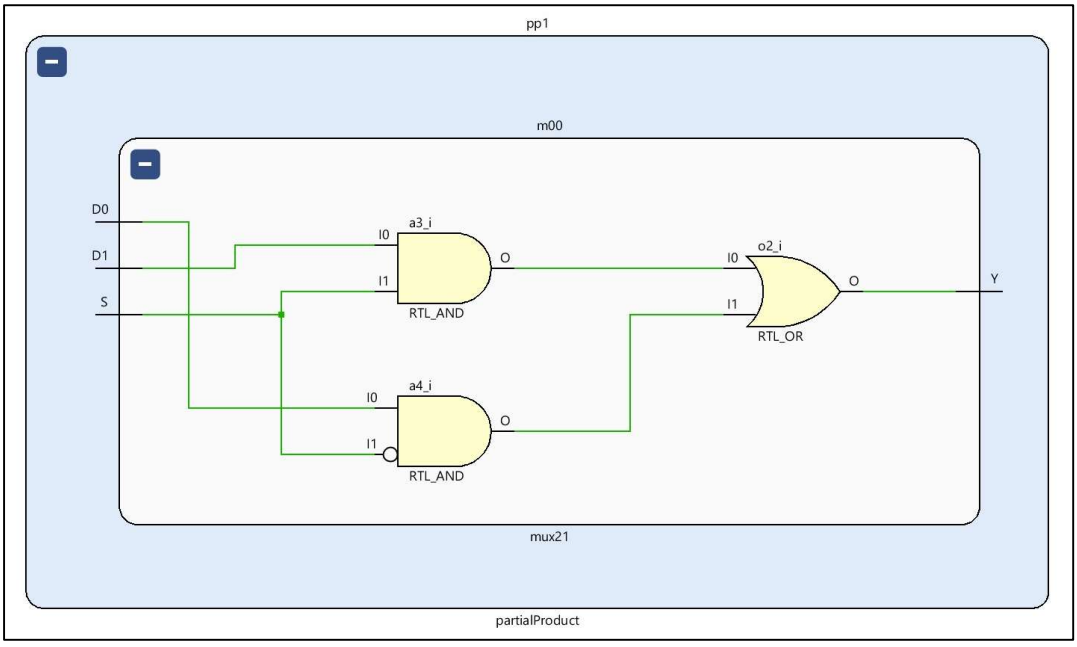


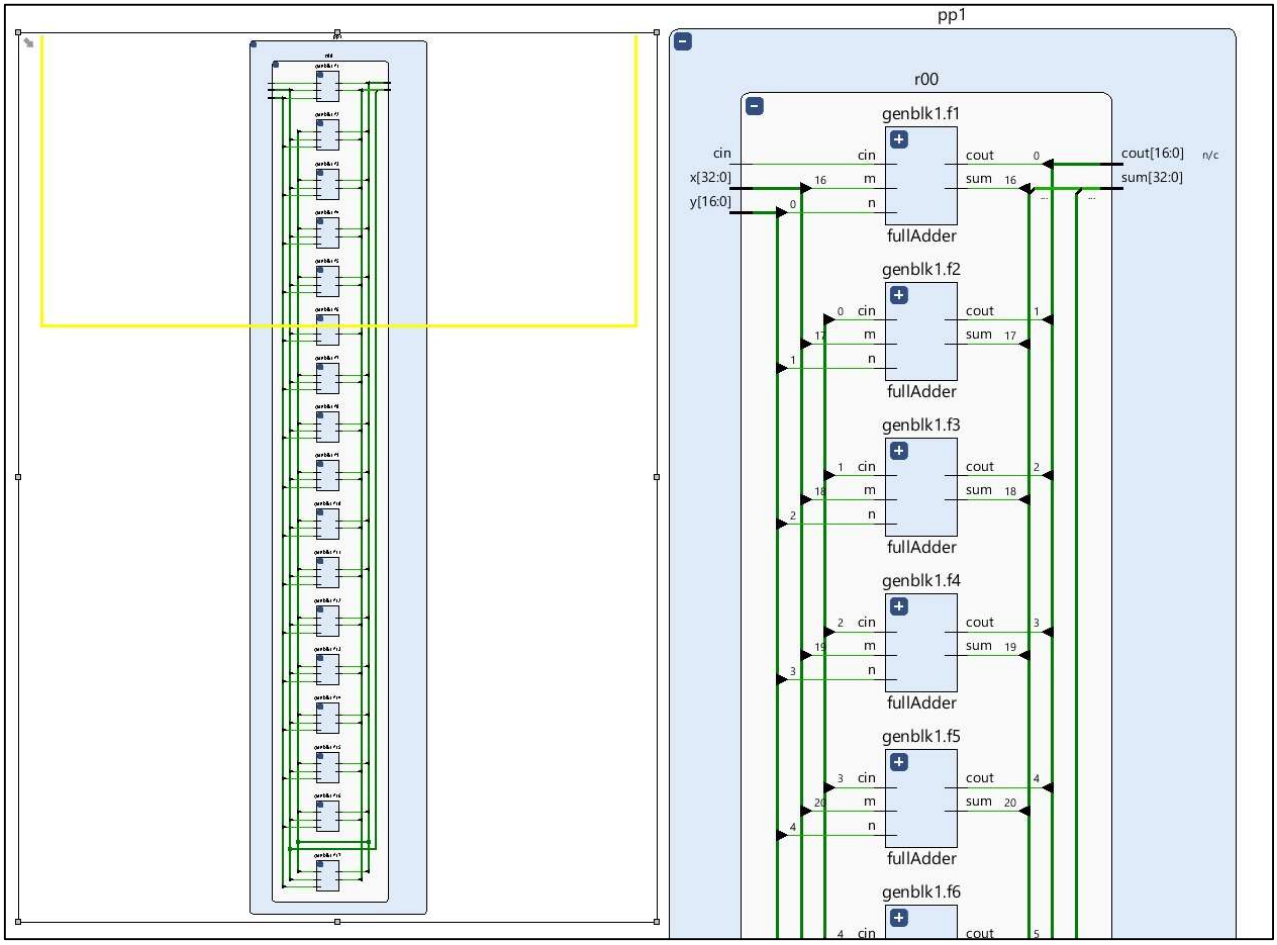


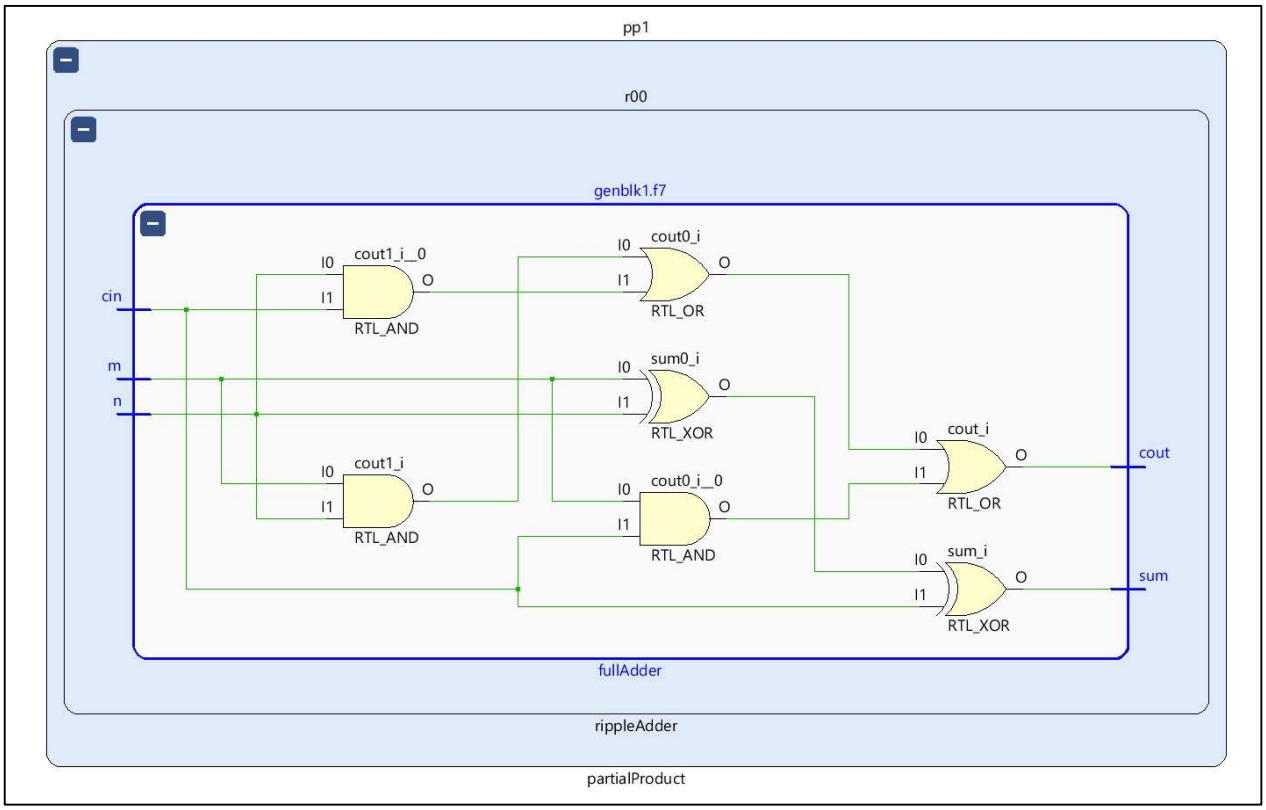



pp1
a10
m10
m20
m30
m40
mux21_17
RTL_MUX
RTL_XOR
RTL_OR
a1[16:0]
a1_double[16:0]
a1_double_not[16:0]
a1_not[16:0]
b0
b1
zin
ain[16:0]
a1_con
partialProduct


pp1
cc0
b0
b1
zin
cin_con_i
t1_i
t2_i
RTL_AND
RTL_OR
cin_con
partialProduct

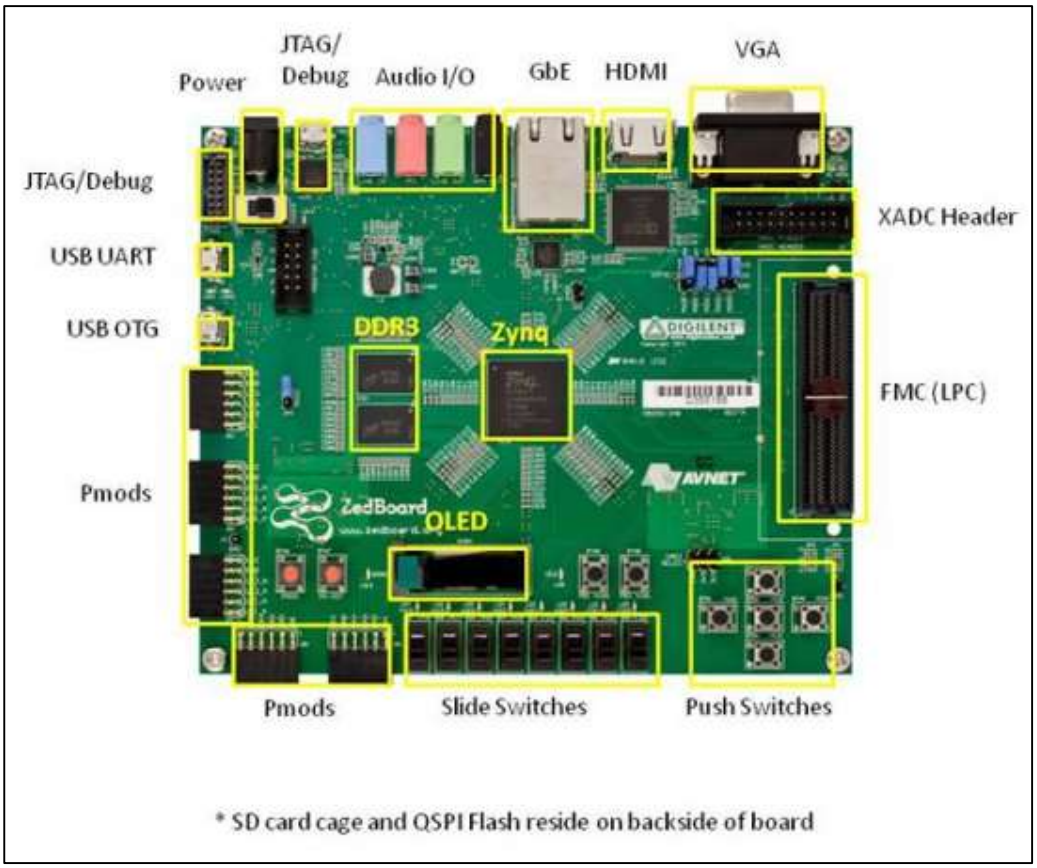

pp1
m00
D0
D1
S
a3_i
a4_i
o2_i
RTL_AND
RTL_OR
Y
mux21
partialProduct


pp1
r00
genblk1.f1
cin
x[32:0]
y[16:0]
cin
m
n
cout
sum
fullAdder
cout[16:0]
sum[32:0]
genblk1.f2
genblk1.f3
genblk1.f4
genblk1.f5
genblk1.f6


pp1
r00
genblk1.f7
cin
m
n
cout1_i__0
cout0_i
sum0_i
cout1_i
cout0_i__0
cout_i
sum_i
RTL_AND
RTL_OR
RTL_XOR
cout
sum
fullAdder
rippleAdder
partialProduct

5. Summary of Timing/ Power/ Utilization

The board used for implementation is ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1).



As the architectures are combinational in nature, a virtual clock is used for the timing constraints. Specifications of the virtual clock are as follows:

Name	Waveform	Period (ns)	Frequency (MHz)
virtual_clock	{0.000 250.000}	500.000	2.000

Array Multiplier (16 bit unsigned numbers)

Project Summary

Overview | Dashboard

Display name: ZedBoard Zynq Evaluation and Development Kit
Board part name: em.avnet.com:zed:part0:1.4
Board revision: d
Connectors: No connections
Repository path: D:/Xilinx/Vivado/2019.2/data/boards/board_files
URL: http://www.zedboard.org
Board overview: ZedBoard Zynq Evaluation and Development Kit
Changes

Synthesis

Status: ✓ Complete
Messages: 4 warnings
Part: xc7z020clg484-1
Strategy: Vivado Synthesis Defaults
Report Strategy: Vivado Synthesis Default Reports
Incremental synthesis: None

Implementation — **Summary** | Route Status

Status: ✓ Complete
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Implementation Defaults
Report Strategy: Vivado Implementation Default Reports
Incremental implementation: None

DRC Violations

Summary: 2 critical warnings
1 warning
Implemented DRC Report

Timing — **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS): 455.603 ns
Total Negative Slack (TNS): 0 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 32
Implemented Timing Report

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	455.603 ns	Worst Hold Slack (WHS):	5.167 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	32	Total Number of Endpoints:	32

All user specified timing constraints are met.

Utilization — Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	360	53200	0.68
IO	64	200	32.00

Power — **Summary** | On-Chip

Total On-Chip Power: 0.105 W
Junction Temperature: 26.2 °C
Thermal Margin: 58.8 °C (4.9 W)
Effective θJA: 11.5 °C/W
Power supplied to off-chip devices: 0 W

Array Multiplier (16 bit signed numbers)

Project Summary

Overview | Dashboard

Display name: ZedBoard Zynq Evaluation and Development Kit
Board part name: em.avnet.com:zed:part0:1.4
Board revision: d
Connectors: No connections
Repository path: D:/Xilinx/Vivado/2019.2/data/boards/board_files
URL: http://www.zedboard.org
Board overview: ZedBoard Zynq Evaluation and Development Kit
Changes

Synthesis

Status: ✓ Complete
Messages: 4 warnings
Part: xc7z020clg484-1
Strategy: Vivado Synthesis Defaults
Report Strategy: Vivado Synthesis Default Reports
Incremental synthesis: None

Implementation — **Summary** | Route Status

Status: ✓ Complete
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Implementation Defaults
Report Strategy: Vivado Implementation Default Reports
Incremental implementation: None

DRC Violations

Summary: 2 critical warnings
1 warning
Implemented DRC Report

Timing — **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS): 455.168 ns
Total Negative Slack (TNS): 0 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 32
Implemented Timing Report

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	455.168 ns	Worst Hold Slack (WHS):	4.993 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	32	Total Number of Endpoints:	32

All user specified timing constraints are met.

Utilization — Post-Synthesis | **Post-Implementation**

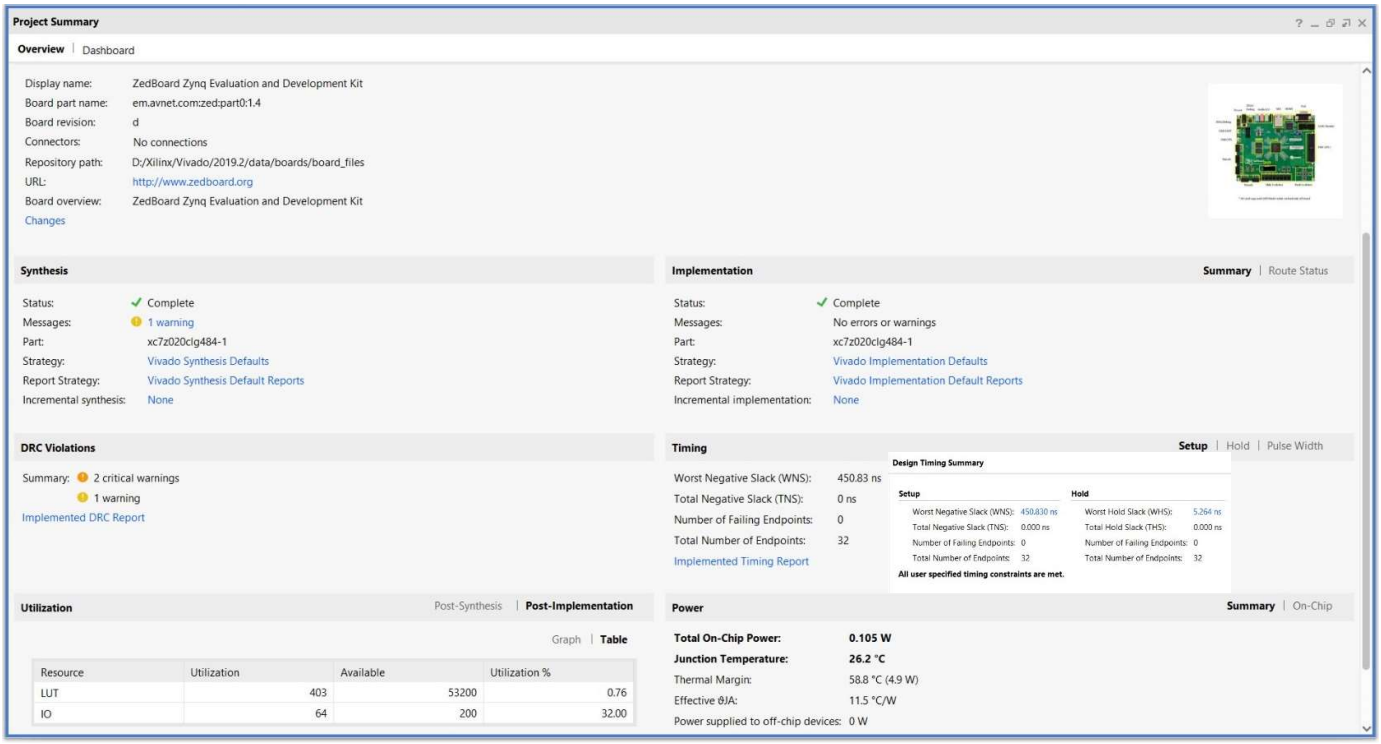
Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	359	53200	0.67
IO	64	200	32.00

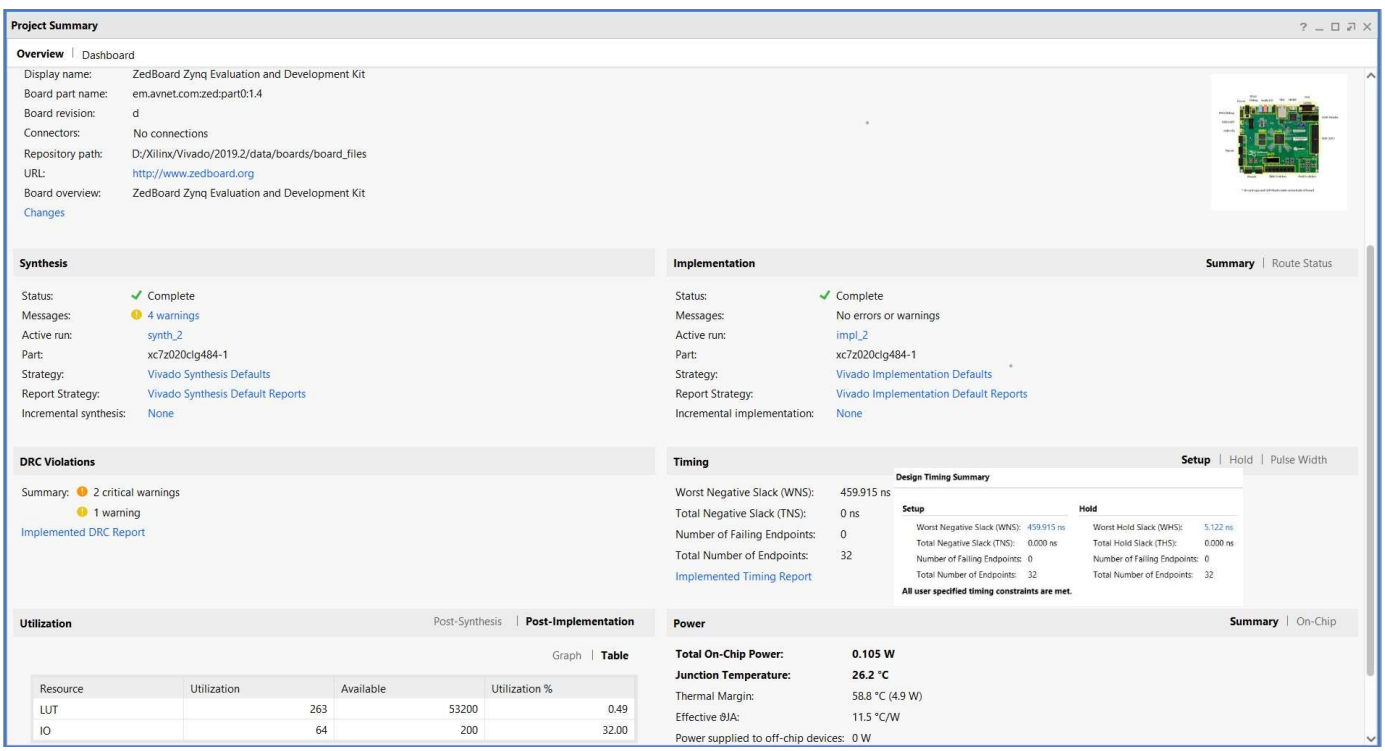
Power — **Summary** | On-Chip

Total On-Chip Power: 0.105 W
Junction Temperature: 26.2 °C
Thermal Margin: 58.8 °C (4.9 W)
Effective θJA: 11.5 °C/W
Power supplied to off-chip devices: 0 W

Booth Multiplier (16-bit signed numbers)



Modified Booth Multiplier (16-bit signed numbers)



4. FUTURE WORK

All these architectures can now be designed in a sequential manner to save up on the area and the hardware used. After that similar procedure can be carried out on them and they can be compared with their combinational counterparts. Effect of different adder architectures (like CLA, Carry Save etc) on multiplier can also be studied.

REFERENCES

1. Analysis of Multipliers Based on Various Performance Measures, (IJERT ISSN: 2278-0181)
2. Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition by Samir Palnitkar
3. Implementation of Booth Multiplier and Modified Booth Multiplier
4. (IJRTER ISSN: 2455-1457)
5. Power, Area and Delay Comparision Of Different Multipliers (IJSETR ISSN: 2278 – 7798)
6. Hardware Implementation of Configurable Booth Multiplier on FPGA, Kishore Shinde, A.K. Kureshi
7. Vivado Design Suite User Guide, link: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf
8. Vivado QuickTake Tutorials, link: https://www.youtube.com/watch?v=U_16tKynK7Y&list=PL35626FEF3D5CB8F2
9. ZedBoard : http://zedboard.org/product/zedboard