

快速乘法器设计

摘要:报告阐述了快速乘法器设计。通过采用 *radix-4 Booth* 编码以及 *Wallace* 加法树方法，实现设计 32 位无符号整型快速乘法器。并通过代码改进，实现 32 位有符号乘法功能，使用者可根据实际情况选用。并在此基础上，设计出 4 级流水的 Booth 编码器改进，2 级流水的符合 *IEEE* 标准的 32 位单精度浮点型乘法器，和改进的 *radix-16 Booth* 编码改进三种创新。

关键词: radix-4 Booth; Wallace 加法树; 有符号; 4 级流水; IEEE 标准; 单精度浮点型乘法器; 改进的 radix-16 Booth

一、基本功能实现-32 位有无符号整型快速乘法器

采用 radix4 Booth 编码 ,根据计算公式

$$\begin{aligned} B &= -B_{n-1}2^{n-1} + \left(\sum_{i=0}^{n-2} B_i * 2^i \right) \\ &= -B_{n-1}2^{n-1} + B_{n-2}2^{n-2} + B_{n-3}2^{n-3} + B_{n-4}2^{n-4} + \dots + \\ &\quad B_32^3 + B_22^2 + B_12^1 + B_02^0 + B_{-1} \\ &= (-2B_{n-1} + B_{n-2} + B_{n-3})2^{n-2} + (-2B_{n-2} + B_{n-3} + B_{n-4})2^{n-4} + \dots \\ &\quad (-2B_5 + B_4 + B_3)2^4 + (-2B_3 + B_2 + B_1)2^2 + (-2B_1 + B_0 + B_{-1})2^0 \end{aligned}$$

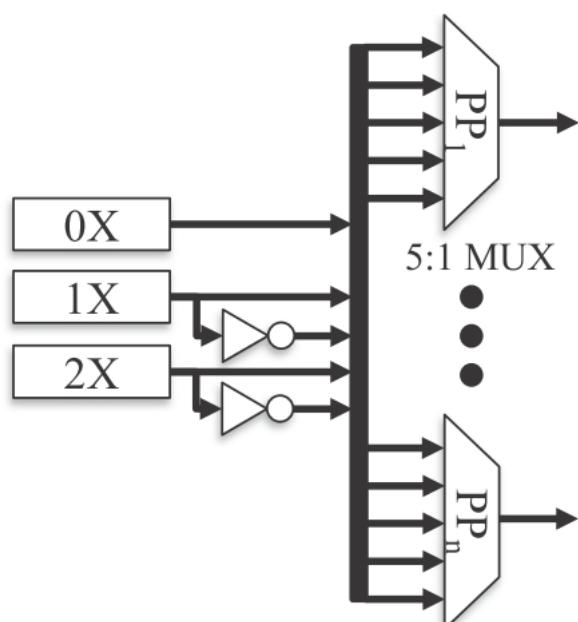
可以得到:

$$\begin{aligned} A * B &= A * (B_{n-2} - B_{n-1})2^{n-1} + A * (B_{n-3} - B_{n-2})2^{n-2} + \\ &\quad A * (B_{n-4} - B_{n-3})2^{n-3} + \dots + A * (B_3 - B_4)2^4 + A * (B_2 - B_3)2^3 + \\ &\quad A * (B_1 - B_2)2^2 + A * (B_0 - B_1)2^1 + A * (B_{-1} - B_0)2^0 \end{aligned}$$

所以可以得到一下查找表:

X2i +1	X 2i	X2 i-1	operat ion	Sing le	Dou ble	Negat ive
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	2Y	0	1	0
1	0	0	-2Y	0	1	1
1	0	1	Y	1	0	1
1	1	0	-Y	1	0	1
1	1	1	0	0	0	1

根据此查找表^[1]，构筑出部分积的多路选择器^[2]：



(a) Radix-4 Booth Encoder

创建一个转 Booth 编码的控制模块

Booth_Ctrl

```

module Booth_Ctrl
#(
    parameter LENGTH = 32
)
(
    input wire [LENGTH-1:0] a_i, //乘数 A
    input wire [2:0] b_i, //被乘数 B 三位选择
    output reg [LENGTH+1:0] bo_o
);

```

1. a_i 为乘数 A 输入
2. b_i 为乘数 B 的三位输入选择
3. bo_o 为 booth 编码后的值

```

`ifndef UNSIGNED_BOOTH
wire [LENGTH:0] a_tr; //无符号乘法, 扩展一位最高位0

assign a_tr = {1'b0, a_i};

always @(*) begin //无符号
    case(b_i)
        3'b000: bo_o = 'b0;
        3'b001: bo_o = {a_tr[LENGTH], a_tr};
        3'b010: bo_o = {a_tr[LENGTH], a_tr};
        3'b011: bo_o = {a_tr, 1'b0};
        3'b100: bo_o = -{a_tr, 1'b0};
        3'b101: bo_o = -{a_tr[LENGTH], a_tr};
        3'b110: bo_o = -{a_tr[LENGTH], a_tr};
        3'b111: bo_o = 'b0;
    endcase
end

```

Booth 转码, 对应上图查找表

由于 Booth 编码基于补码有符号公式, 所以转成无符号需要对其扩展最高位 0。

所以轻易可得以下有符号编码:

```

`else . . .
always @(*) begin . . . //有符号
    case(b_i)
        3'b000: bo_o = 'b0 . . . ;
        3'b001: bo_o = {{2{a_i[LENGTH-1]}}, a_i} . . . ;
        3'b010: bo_o = {{2{a_i[LENGTH-1]}}, a_i} . . . ;
        3'b011: bo_o = {a_i[LENGTH-1], a_i, 1'b0} . . . ;
        3'b100: bo_o = -{a_i[LENGTH-1], a_i, 1'b0} . . . ;
        3'b101: bo_o = -{{2{a_i[LENGTH-1]}}, a_i} . . . ;
        3'b110: bo_o = -{{2{a_i[LENGTH-1]}}, a_i} . . . ;
        3'b111: bo_o = 'b0 . . . ;
    endcase
end

`endif

```

(注：此时 `bo_o` 位宽可以减少两位，但为了方便与无符号对应，故而采取扩展两位符号位策略，结果不变）

转 Booth 编码的控制模块完成

顶层文件例化转 Booth 编码

```

4
5 `include "UnsignedChoose.v"
6 module Booth_mul
7 (
8     input wire [31:0] A, . . . //乘数 A
9     input wire [31:0] B, . . . //被乘数 B
10    output wire [63:0] P . . . //结果 P
11 );
12
13 //*****// parameter-define //*****//
14
15 parameter LENGTH = 32;
16

```

```

//实例化 Booth_Ctrl 模块，实现转码
genvar i;
generate
    for(i=0; i<LENGTH/2+1; i=i+1) begin: Booth_Ctrl_i
        Booth_Ctrl
        #(
            .LENGTH(LENGTH)
        )
        Booth_Ctrl
        (
            .a_i(A),
            .b_i({B_tr[i*2+2], B_tr[i*2+1], B_tr[i*2]}),
            .bo_o(boo_o[i])
        );
        if(i != 16)
            assign pp[i] = {{(LENGTH-2-i*2){boo_o[i][LENGTH+1]}}, boo_o[i], {(i*2){1'b0}}}; //部分积
        else
            assign pp_16th = {boo_o[i], {(i*2){1'b0}}}; //扩展位所产生的第 17 个部分积
    end
endgenerate

```

生成 17 个部分积

第 17 个部分积由于 B 的低 0 位和两位符号位扩展而生成。

```
1 // 扩展最低位置B[-1], 为0
2
3 `ifdef UNSIGNED_BOOTH ..... // 是否选择为无符号
4
5 assign B_tr = {2'b0, B, 1'b0}; ..... // 最高位扩展1位0, 代表无符号, 再扩展一位0组成三位一组数
6
7 `else
8
9 assign B_tr = {{2{B[LENGTH-1]}}, B, 1'b0}; ..... // 扩展两位符号位
10
11 `endif
```

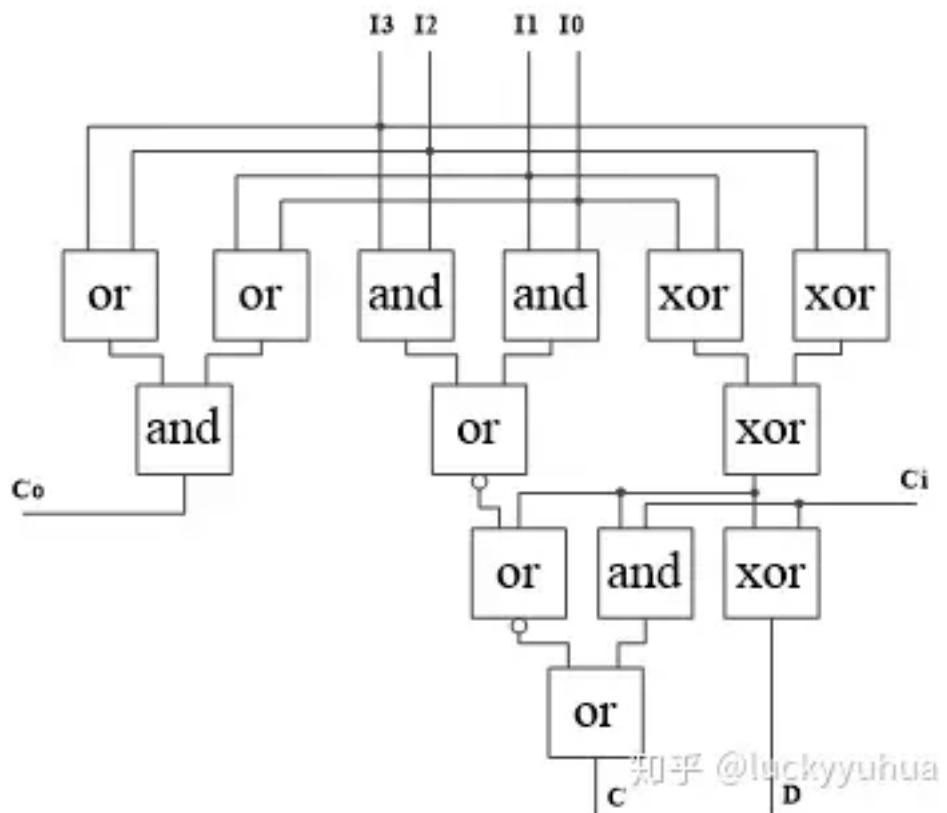
生成部分积后，相加则可得到结果，但由于行波进位加法器进位传递延时对电路性能存在非常大的影响，此时采用多级 4-2 压缩器组成 Wallace 加法树（也可以使用 CSA），并行计算，实现优化运算。

4-2 压缩器模块：

compressor4_2

```
3 module compressor4_2
4 #(
5     parameter LENGTH = 32 ..... // 位宽参数
6 )
7 (
8     input wire [LENGTH*2+1:0] I0 , ..... // 加数 0
9     input wire [LENGTH*2+1:0] I1 , ..... // 加数 1
10    input wire [LENGTH*2+1:0] I2 , ..... // 加数 2
11    input wire [LENGTH*2+1:0] I3 , ..... // 加数 3
12    input wire ..... Ci , ..... // 进位输入
13
14    output wire [LENGTH*2+1:0] C , ..... // 高位数
15    output wire [LENGTH*2+1:0] D , ..... // 低位数
16    output wire ..... Co ..... // 进位输出
17 );
```

输入输出定义



根据此逻辑图，得到以下代码：

```

19 //*****// wire-define //*****//
20 wire [LENGTH*2+1:0] T ;
21 wire [LENGTH*2+1:0] T_1 ;
22 wire [LENGTH*2+1:0] C_o ;
23
24
25 //*****// main-code //*****//
26
27 //运算式
28
29 assign T = I0 ^ I1 ^ I2 ^ I3 ;
30 assign T_1 = (I0 & I1) | (I2 & I3) ;
31 assign C_o = (I0 | I1) & (I2 | I3) ;
32
33 assign C = (T & {C_o[LENGTH*2:0], Ci}) | ~ (T | ~T_1) ;
34 assign D = T ^ {C_o[LENGTH*2:0], Ci} ;
35 assign Co = C_o[LENGTH*2+1] ;
36

```

4-2 压缩器模块完成

最后结果仍然需要一个全加器模块生成

创建一个 32 位全加器：

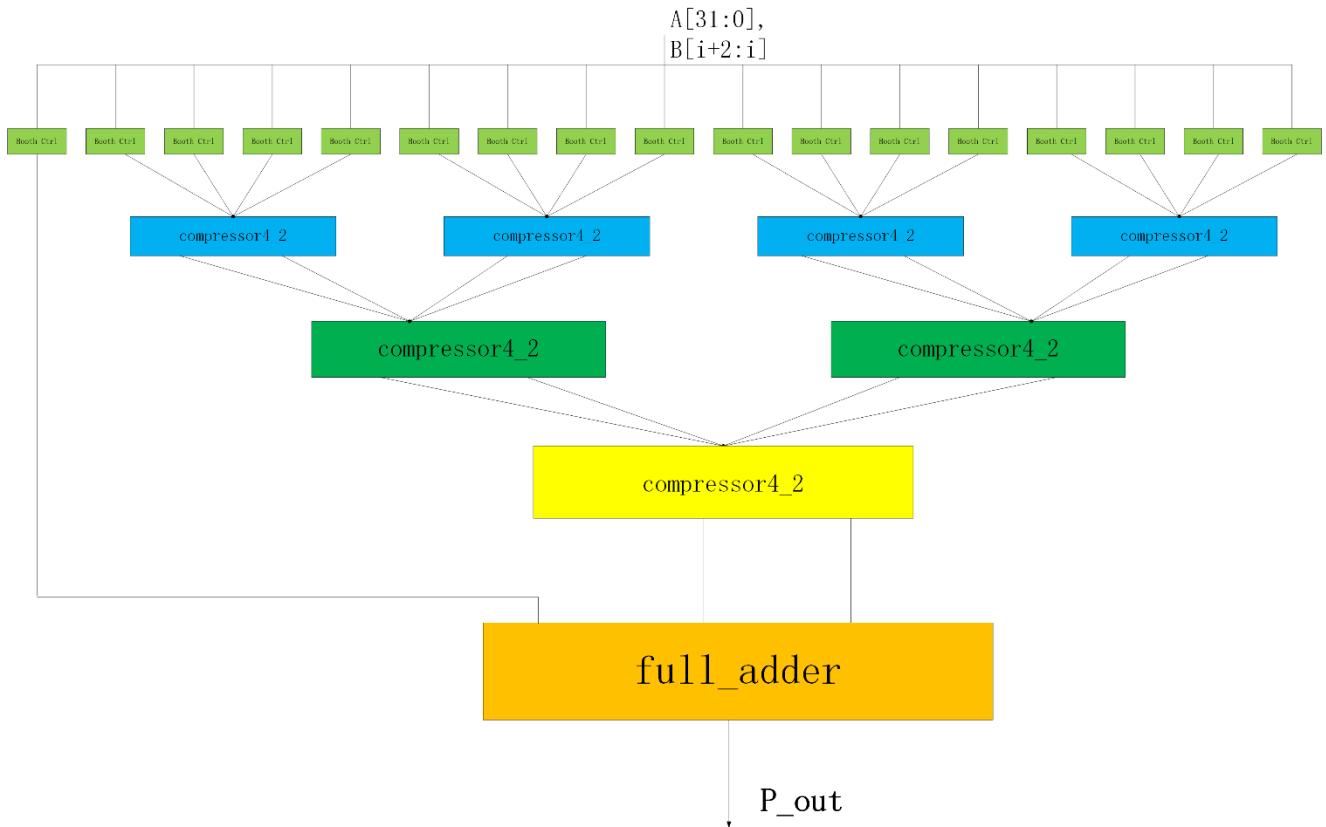
```

3 module full_adder
4 #(
5     parameter LENGTH = 32 //位宽参数
6 )
7 (
8     input wire [LENGTH*2+1:0] a , //加数 a
9     input wire [LENGTH*2+1:0] b , //加数 b
10    input wire [LENGTH*2+1:0] c , //加数 c
11    input wire ci , //进位输入
12
13    output wire [LENGTH*2+1:0] s , //和
14    output wire co //进位输出
15 );
16
17
18 //***** main-code *****
19 assign {co, s} = a + b + c + ci ; //行为级建模
20

```

这里有 3 个输入，原因是前面 4-2 压缩器处理了 16 个部分积，还有因扩展生成的第 17 个部分积未处理，在这里加上。

所以得到以下流程图：



总共采用 17 个 Booth_Ctrl 模块生成部分积，4 个一级 4-2 压缩器，2 个二级 4-2 压缩器，1 个三级 4-2 压缩器，和一个全加器。

最终顶层文件：

```
4 `include "UnsignedChoose.v"
5 module Booth_mul
6 (
7     input wire [31:0] A, //乘数 A
8     input wire [31:0] B, //被乘数 B
9     output wire [63:0] P //结果 P
10 );
11
12 //***** parameter-define *****
13 parameter LENGTH = 32;
14
15
```

```
16
17 //***** wire-define *****
18
19 wire [LENGTH+1:0] boo_o [0:LENGTH/2]; //booth转码输出值
20 wire [LENGTH*2-1:0] pp [0:LENGTH/2-1]; //部分积
21 wire [LENGTH*2+1:0] pp_16th; //扩展位产生的第17个部分积分, 全加器输入值
22
23 wire [3:0] pp_stg1_co; //第一级压缩器进位输出进位
24 wire [3:0] co; //二三级压缩器以及全加器输出进位
25
26 wire [LENGTH*2-1:0] pp_stg2_m [0:3]; //第一级压缩器输出高位值, 第二级压缩器输入值
27 wire [LENGTH*2-1:0] pp_stg2_l [0:3]; //第一级压缩器输出低位值, 第二级压缩器输入值
28
29 wire [LENGTH*2-1:0] pp_stg3_m [0:1]; //第二级压缩器输出高位值, 第三级压缩器输入值
30 wire [LENGTH*2-1:0] pp_stg3_l [0:1]; //第二级压缩器输出低位值, 第三级压缩器输入值
31
32 wire [LENGTH*2-1:0] pp_stg_m_end; //第三级压缩器输出高位值, 全加器输入值
33 wire [LENGTH*2-1:0] pp_stg_l_end; //第三级压缩器输出低位值, 全加器输入值
34
35 wire [LENGTH+2:0] B_tr; //输入数 B 扩展
36
37 wire [LENGTH*2+1:0] P_out; //全加器输出
38
39 //***** main-code *****

```

```
//扩展最低位置B[-1], 为0
`ifdef UNSIGNED_BOOTH //是否选择为无符号
assign B_tr = {2'b0, B, 1'b0}; //最高位扩展1位0, 代表无符号, 再扩展一位0组成三位一组数
`else
assign B_tr = {{2{B[LENGTH-1]}}, B, 1'b0}; //扩展两位符号位
`endif
```

```

3 //实例化 Booth_Ctrl 模块, 实现转码
4 genvar i;
5 generate
6   for(i=0; i<LENGTH/2+1; i=i+1) begin: Booth_Ctrl_i
7     Booth_Ctrl
8     #(
9       .LENGTH(LENGTH)
10    )
11     Booth_Ctrl
12     (
13       .a_i      (A),
14       .b_i      ({B_tr[i*2+2], B_tr[i*2+1], B_tr[i*2]}),
15       .bo_o     (boo_o[i])
16     );
17     if(i != 16)
18       assign pp[i] = {{(LENGTH-2-i*2){boo_o[i][LENGTH+1]}}, boo_o[i], {(i*2){1'b0}}}; //部分积
19     else
20       assign pp_16th = {boo_o[i], {(i*2){1'b0}}}; //扩展位所产生的第 17 个部分积
21   end
22 endgenerate

```

```

5 //实例化 4-2 压缩器, 实现部分积求和
6 genvar j;
7 generate
8   for(j=0; j<LENGTH/2/4; j=j+1) begin: compressor4_2_level1 //第一级压缩器
9     compressor4_2
10    #(
11      .LENGTH(LENGTH)
12    )
13     compressor4_2_level1_inst
14     (
15       .I0(pp[j*4+0]),
16       .I1(pp[j*4+1]),
17       .I2(pp[j*4+2]),
18       .I3(pp[j*4+3]),
19       .Ci(1'b0),
20       .Co(pp_stg1_co[j]),
21       .C (pp_stg2_m[j]),
22       .D (pp_stg2_l[j])
23     );
24   end
25 endgenerate

```

第一级, 4 个压缩器 (上图)

```

3 compressor4_2
4 #(
5   .LENGTH(LENGTH)
6 )
7 compressor4_2_level2_inst_0
8 (
9   .I0(pp_stg2_l[0]),
10  .I1({pp_stg2_m[0][LENGTH*2-2:0], 1'b0}),
11  .I2(pp_stg2_l[1]),
12  .I3({pp_stg2_m[1][LENGTH*2-2:0], 1'b0}),
13  .Ci(1'b0),
14  .Co(co[0]),
15  .C (pp_stg3_m[0]),
16  .D (pp_stg3_l[0])
17 );

```

```

compressor4_2
#(
  .LENGTH(LENGTH)
)
compressor4_2_level2_inst_1
(
  .I0(pp_stg2_l[2]),
  .I1({pp_stg2_m[2][LENGTH*2-2:0], 1'b0}),
  .I2(pp_stg2_l[3]),
  .I3({pp_stg2_m[3][LENGTH*2-2:0], 1'b0}),
  .Ci(1'b0),
  .Co(co[1]),
  .C (pp_stg3_m[1]),
  .D (pp_stg3_l[1])
);

```

第二级, 2 个压缩器 (上图)

```

9 compressor4_2
10 #( 
11     .LENGTH(LENGTH)
12 )
13 compressor4_2_level3_inst
14 (
15     .I0(pp_stg3_l[0]          ) ,
16     .I1({pp_stg3_m[0][LENGTH*2-2:0], 1'b0} ) ,
17     .I2(pp_stg3_l[1]          ) ,
18     .I3({pp_stg3_m[1][LENGTH*2-2:0], 1'b0} ) ,
19     .Ci(1'b0                  ) ,
20     .Co(co[2]                 ) ,
21     .C (pp_stg_m_end         ) ,
22     .D (pp_stg_l_end         ) ,
23 );

```

第 3 级，1 个压缩器（上图）

```

full_adder
#(
    .LENGTH(LENGTH)
)
full_adder_inst
(
    .a ({pp_stg_m_end[LENGTH*2-1], pp_stg_m_end, 1'b0} ) ,
    .b ({2{pp_stg_l_end[LENGTH*2-1]}}, pp_stg_l_end) ,
    .c (pp_16th           ) ,
    .ci(1'b0              ) ,
    .s (P_out             ) ,
    .co(co[3]             )
);

assign P = P_out[LENGTH*2-1:0];

```

最后的全加器以及输出结果（上图）

TB 文件编写:

```
25  
26 assign P1 = A * B;  
27  
28 initial begin  
29     for (i = 0; i < 2000; i = i + 1) begin  
30         // 设置输入数据  
31         `ifdef UNSIGNED_BOOTH  
32             A = {$random} ;  
33             B = {$random} ;  
34  
35         `else  
36             A = $random ;  
37             B = $random ;  
38         `endif  
39         // 等待一些时间  
40         #100;  
41  
42         // 打印输出结果  
43         $display("Input A = %b", A);  
44         $display("Input B = %b", B);  
45         $display("Output P = %b", P);  
46         $display("Output P1 = %b", P1);  
47
```

根据条件编译, 选择有符号还是无符号, 对应前面有无符号 Booth 编码模块。
随机生成 2000 个数相乘, 得到结果与 P1 对比 (P1 为调用软件自带乘法运算结果)

使用 quartus ii 18.0 创建工程, 并联合 modelsim 仿真

仿真结果以及波形如下:

```
# Input A = 10110110001101110001010001101100  
# Input B = 11001001011011011111011010010010  
# Output P = 1000111101011111100000111111110101010111000001010110110110011000  
# Output P1 = 1000111101011111100000111111110101010111000001010110110110011000  
# Test Passed: Output matches expected result  
# Total Tests:      2000  
# Passed:          2000  
# Failed:           0  
# //=====Random test all Pass!!=====//
```

msgs					
+ A	32'd3057063020	617026889	483929913	2874739286	2102130682
+ B	32'd3379426962	2736712774	1091553154	3296576136	634129739
+ P1	64'd10331121194321145240	1688625369027780086	528235222850095602	9476796927449278896	1333023580720551998
+ P	64'd10331121194321145240	1688625369027780086	528235222850095602	9476796927449278896	1333023580720551998
+ i	32'd2000	1385	1386	1387	1388
+ pass_count	32'd2000	1385	1386	1387	1388
+ fail_count	32'd0	0			

有符号乘法仿真结果以及波形:

```

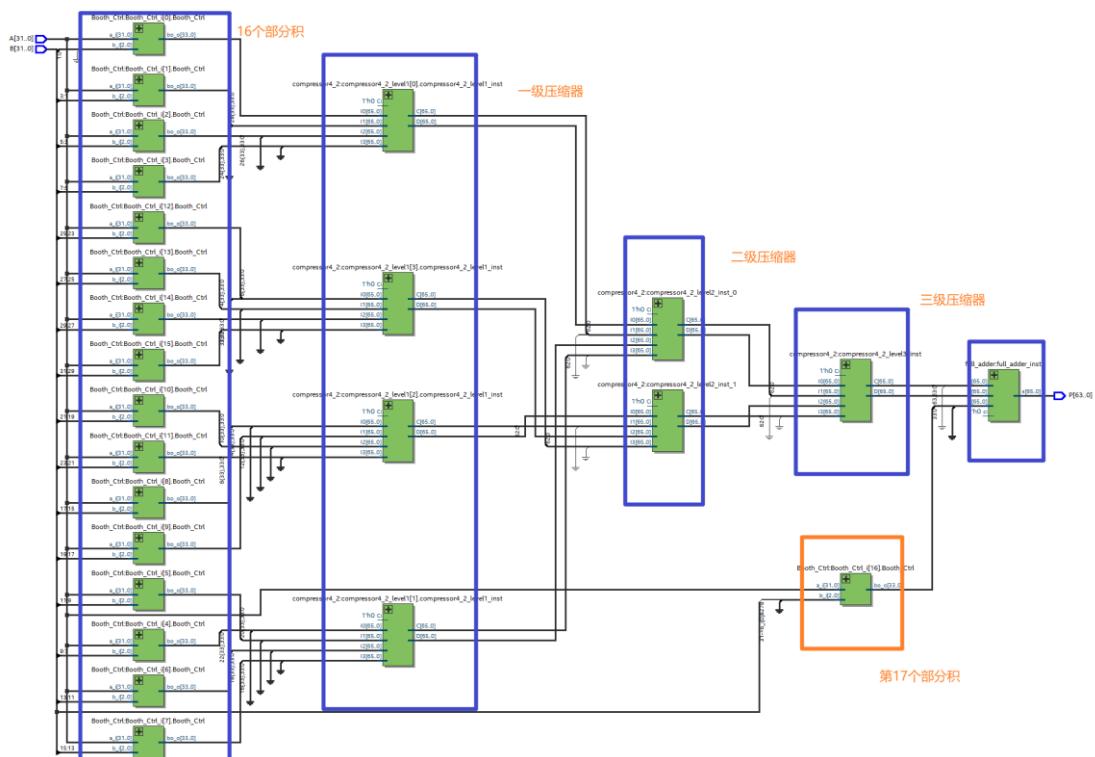
# ----- Output message -----
# Input A = 10110110001101110001010001101100
# Input B = 1100100101101101111011010010010
# Output P = 000011111011101001111000111111110101011100000101011011
# Output P1 = 000011111011101001111000111111110101011100000101011011
# Test Passed: Output matches expected result
# Total Tests:      2000
# Passed:          2000
# Failed:           0
# //=====Random test all Pass!!=====//

```

msgs					
+ A	32'd675449168	-862835559	-881412458	-1207261072	-864584040
+ B	32'd612428617	1731182542	-1454121390	-1846557661	1258345366
+ P1	64'd4136643998...	-1493725856357610978	1281680708590276620	2229277181328672592	-1087945320251558640
+ P	64'd4136643998...	-1493725856357610978	1281680708590276620	2229277181328672592	-1087945320251558640
+ i	32'h000004e7	0000057a	0000057b	0000057c	0000057d
+ pass_count	32'h000004e7	0000057a	0000057b	0000057c	0000057d
+ fail_count	32'h00000000	00000000			

成功验证乘法器。

查看 RTL 视图:



32 位有无符号快速乘法器设计成功。

二、创新拓展

(一) 4 级流水的 Booth 编码器改进

为了增加 Booth 乘法器性能，提高其吞吐率，现给乘法器加装 4 级流水。可根据实际情况修改确定流水级数

```
//first-pipeline
reg [LENGTH*2-1:0] pp_reg [0:LENGTH/2-1];
reg [LENGTH*2+1:0] pp_16th_reg;

//second-pipeline
reg [LENGTH*2-1:0] pp_stg2_m_reg [0:3];
reg [LENGTH*2-1:0] pp_stg2_l_reg [0:3];

//third-pipeline
reg [LENGTH*2-1:0] pp_stg3_m_reg [0:1];
reg [LENGTH*2-1:0] pp_stg3_l_reg [0:1];

//forth-pipeline
reg [LENGTH*2-1:0] pp_stg_m_end_reg;
reg [LENGTH*2-1:0] pp_stg_l_end_reg;
```

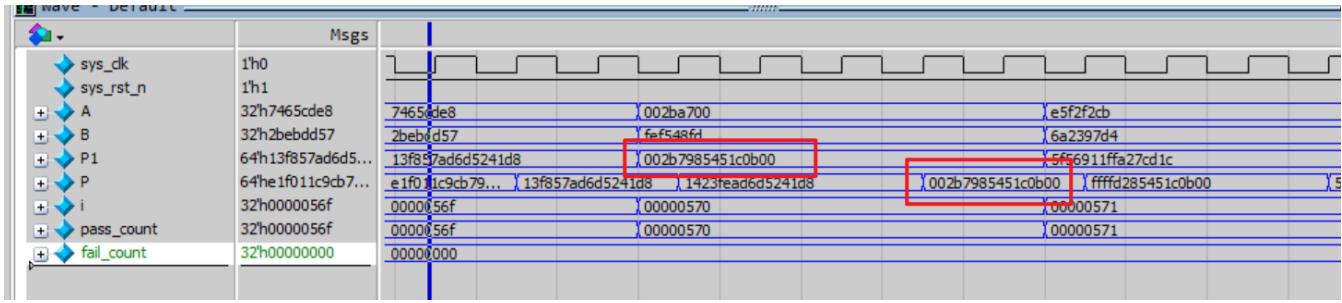
```
-----// first-pipeline //-----
always @(posedge sys_clk or negedge sys_rst_n) begin
    if(sys_rst_n == 1'b0) begin
        pp_16th_reg <= 'b0;
        for(i=0; i<LENGTH/2; i=i+1)
            pp_reg[i] <= 'b0;
    end
    else begin
        pp_16th_reg <= pp_16th;
        for(i=0; i<LENGTH/2; i=i+1)
            pp_reg[i] <= pp[i];
    end
end
```

```

32 //-----// second-pipeline //-----//
33 always @(posedge sys_clk or negedge sys_rst_n) begin
34   if(sys_rst_n == 1'b0)
35     for(i=0; i<LENGTH/2/4; i=i+1)
36       pp_stg2_m_reg[i] <= 'b0;
37   else
38     for(i=0; i<LENGTH/2/4; i=i+1)
39       pp_stg2_m_reg[i] <= pp_stg2_m[i];
40 end
41
42 always @(posedge sys_clk or negedge sys_rst_n) begin
43   if(sys_rst_n == 1'b0)
44     for(i=0; i<LENGTH/2/4; i=i+1)
45       pp_stg2_l_reg[i] <= 'b0;
46   else
47     for(i=0; i<LENGTH/2/4; i=i+1)
48       pp_stg2_l_reg[i] <= pp_stg2_l[i];
49 end
50
51 //-----// third-pipeline //-----//
52 always @(posedge sys_clk or negedge sys_rst_n) begin
53   if(sys_rst_n == 1'b0) begin
54     pp_stg3_l_reg[0] <= 'b0;
55     pp_stg3_l_reg[1] <= 'b0;
56   end
57   else begin
58     pp_stg3_l_reg[0] <= pp_stg3_l[0];
59     pp_stg3_l_reg[1] <= pp_stg3_l[1];
60   end
61 end
62
63 always @(posedge sys_clk or negedge sys_rst_n) begin
64   if(sys_rst_n == 1'b0) begin
65     pp_stg3_m_reg[0] <= 'b0;
66     pp_stg3_m_reg[1] <= 'b0;
67   end
68   else begin
69     pp_stg3_m_reg[0] <= pp_stg3_m[0];
70     pp_stg3_m_reg[1] <= pp_stg3_m[1];
71   end
72 end
73
74 //-----// forth-pipeline //-----//
75 always @(posedge sys_clk or negedge sys_rst_n) begin
76   if(sys_rst_n == 1'b0)
77     pp_stg_m_end_reg <= 'b0;
78   else
79     pp_stg_m_end_reg <= pp_stg_m_end;
80 end
81
82 always @(posedge sys_clk or negedge sys_rst_n) begin
83   if(sys_rst_n == 1'b0)
84     pp_stg_l_end_reg <= 'b0;
85   else
86     pp_stg_l_end_reg <= pp_stg_l_end;
87 end

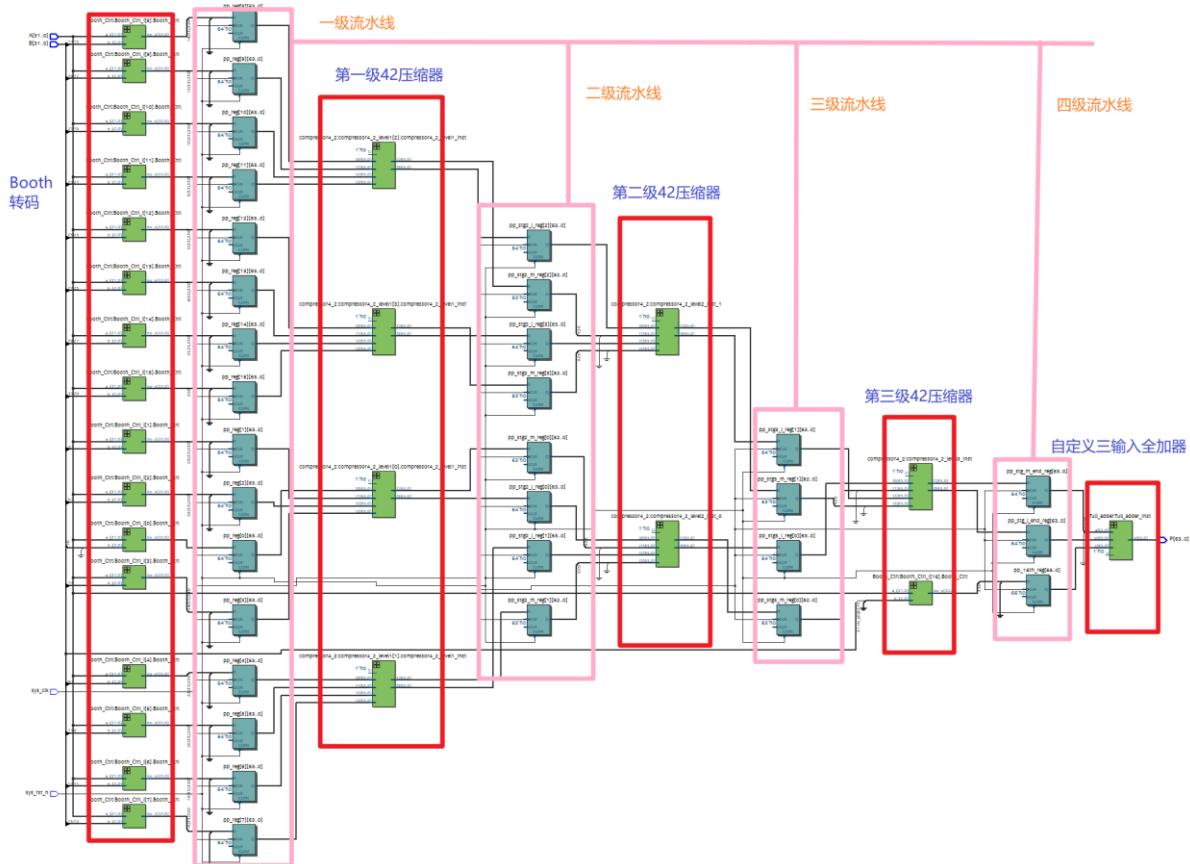
```

TB 仿真文件修改后，仿真波形如下：



输出结果与软件自带乘法运算对比相同，落后 4 个时钟周期，仿真成功

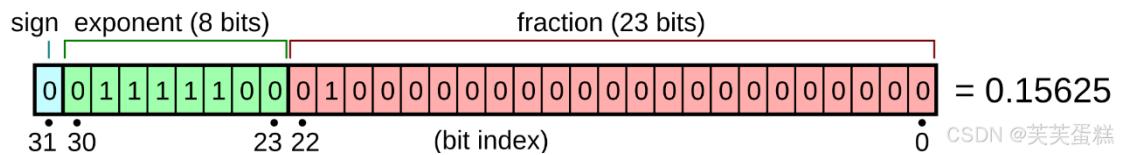
RTL 视图如下：



Booth 乘法器 4 级流水线改进成功。

(二) 在前面 Booth 乘法器基础上，设计出 2 级流水的符合 IEEE754 标准的 32 位单精度浮点型乘法器

32 位单精度浮点型乘法器需要对数字进行解析，最高位为符号位，后面 8 位为阶码，以 2 为底，最后 23 位为浮点数尾数，如图：



公式为：

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$$

代码：

输入输出端口配置：

```
module float_mul_top
(
    input          sys_clk      ,      // 时钟信号
    input          sys_rst_n   ,      // 复位信号
    input          round_config ,      // 决定舍入的方法, 0采用截断, 1采用就近舍入
    input wire [31:0] float_a     ,      // 输入的被乘数
    input wire [31:0] float_b     ,      // 输入的乘数
    output reg [31:0] float_p    ,      // 输出运算结果
    output reg [1:0] overflow    // 输出溢出标志
);
```

提取乘数 float_a 和 float_b 的符号位，阶码以及尾数：

```
/*
-----提取float_a 的符号, 阶码, 尾数-----
always @(posedge sys_clk or negedge sys_rst_n) begin
    if(sys_rst_n == 1'b0) begin //复位, 初始化
        s1 <= 1'b0 ;
        exp1 <= 8'b0 ;
        man1 <= 24'b0 ;
    end
    else begin
        s1 <= float_a[31] ;
        exp1 <= float_a[30:23] ;
        man1 <= {1'b1, float_a[22:0]} ;
    end
end

/*
-----提取float_b 的符号, 阶码, 尾数-----
always @(posedge sys_clk or negedge sys_rst_n) begin
    if(sys_rst_n == 1'b0) begin //复位, 初始化
        s2 <= 1'b0 ;
        exp2 <= 8'b0 ;
        man2 <= 24'b0 ;
    end
    else begin
        s2 <= float_b[31] ;
        exp2 <= float_b[30:23] ;
        man2 <= {1'b1, float_b[22:0]} ;
    end
end
```

判断乘法结果符号：异或判断

```
83 //符号位  
84 assign one_s_out = s1 ^ s2 ; //输入符号异或  
85
```

尾数相乘：使用已设计 Booth 乘法器

```
85  
86 //尾数相乘  
87 Booth_mul Booth_mul_inst  
(  
88     .A({8'b0, man1}),  
89     .B({8'b0, man2}),  
90     .P(one_m_Tem))  
91 );  
92  
93  
94 always @(*) begin  
95     if(man1 == 24'h800_000)  
96         one_m_out = 48'b0;  
97     else if(man2 == 24'h800_000)  
98         one_m_out = 48'b0;  
99     else  
100        one_m_out = one_m_Tem[47:0]; //48位  
101        // one_m_out = man1 * man2;  
102 end  
103
```

将阶码转成补码形式，并使用双符号位，判断是否溢出

```
//阶码相加, 阶码是移码, 移码是符号位取反的补码  
always @(*) begin  
    //把阶码的移码形式变为补码形式, 并且转成双符号位格式, 00为正, 11为负  
    if(exp1[7] == 1)  
        temp1 = {2'b00, 1'b0, exp1[6:0]};  
    else  
        temp1 = {2'b11, 1'b1, exp1[6:0]};  
end  
  
always @(*) begin  
    if(exp2[7] == 1)  
        temp2 = {2'b00, 1'b0, exp2[6:0]};  
    else  
        temp2 = {2'b11, 1'b1, exp2[6:0]};  
end
```

阶码相加:

```
0 //阶码以双符号补码的形式相加计算
1 assign one_e_out[9:0] = temp1[9:0] + temp2[9:0];
```

输出的符号位，阶码以及尾数输入第一级流水寄存

```
/*-----第一级流水寄存-----*/
always @(posedge sys_clk or negedge sys_rst_n) begin
    if(sys_rst_n == 0) begin
        one_s_reg <= 1'b0;
        one_e_reg <= 10'b0;
        one_m_reg <= 48'b0;
    end
    else begin
        one_s_reg <= one_s_out;
        one_e_reg <= one_e_out;
        one_m_reg <= one_m_out;
    end
end
```

进行尾数规范化，以及舍入处理和溢出判断

```
else begin
    if(one_m_reg[47] == 1) begin
        n = 1'b1; // 左归码为1
        mul_out_p = one_m_reg >> 1; // 右移一位
    end
    else begin
        n = 1'b0;
        mul_out_p = one_m_reg;
    end
    if(round_config == 1) begin // 0采用截断，1采用就近舍入
        if(mul_out_p[22] == 1)
            two_m_out[22:0] = mul_out_p[45:23] + 1'b1;
        else
            two_m_out[22:0] = mul_out_p[45:23];
    end
    else
        two_m_out[22:0] = mul_out_p[45:23];
end
end
```

```

166      // 双符号的表示，即符号位为 1 时为正数，为 0 时为负数
167      temp3 = one_e_reg[9:0] + n + 1'b1;
168      // 双符号的定义，01为上溢，10为下溢，符号相同
169      if(temp3[9:8] == 2'b01)
170          two_f_out = 2'b01; // 阶码上溢
171      else if(temp3[9:8] == 2'b10)
172          two_f_out = 2'b10; // 阶码下溢
173      else
174          two_f_out = 2'b00; // 无溢出
175 end

```

阶码转回移码：

```

always @(*) begin // 输出补码转回移码
    case(temp3[7])
        1'b1: two_e_out = {1'b0, temp3[6:0]};
        1'b0: two_e_out = {1'b1, temp3[6:0]};
    endcase
end

```

进行第二级流水寄存：

```

92      else if((two_m_out == 0) && (two_e_out == 0)) begin
93          // 特殊值处理
94          two_s_reg <= 1'b0;
95          two_e_reg <= 8'b0;
96          two_m_reg <= 23'b0;
97          two_f_reg <= 2'b0;
98      end
99      else begin
100          two_s_reg <= one_s_reg;
101          two_f_reg <= two_f_out;
102          two_e_reg <= two_e_out;
103          two_m_reg <= two_m_out;
104      end
105 end

```

输出结果：

```

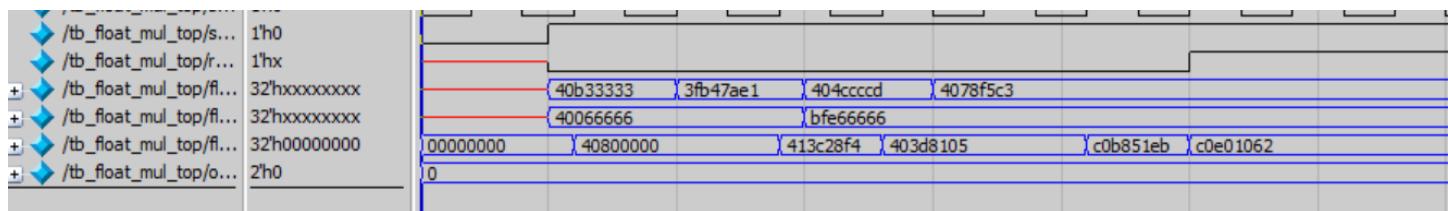
// 输出结果
always @(*) begin
    float_p = {two_s_reg, two_e_reg[7:0], two_m_reg[22:0]};
    overflow = two_f_reg;
end

```

TB 文件测试:

```
11 sys_rst_n = 1;
12 round_cofig = 0;
13 float_a = 32'b0_10000001_01100110011001100110011; //5.6
14 float_b = 32'b0_10000000_00001100110011001100110; //2.1
15 #10
16 //-----
17 sys_rst_n = 1;
18 round_cofig = 0;
19 float_a = 32'b0_01111111_0110100011110101110001; //1.41
20 float_b = 32'b0_10000000_00001100110011001100110; //2.1
21 #10
22 //-----
23 sys_rst_n = 1;
24 round_cofig = 0;
25 float_a = 32'b0_10000000_10011001100110011001101; //3.2
26 float_b = 32'b1_01111111_11001100110011001100110; //-1.8
27 #10
28 //-----
29 sys_rst_n = 1;
30 round_cofig = 0;
31 float_a = 32'b0_10000000_11110001111010111000011; //3.89
32 float_b = 32'b1_01111111_11001100110011001100110; //-1.8
```

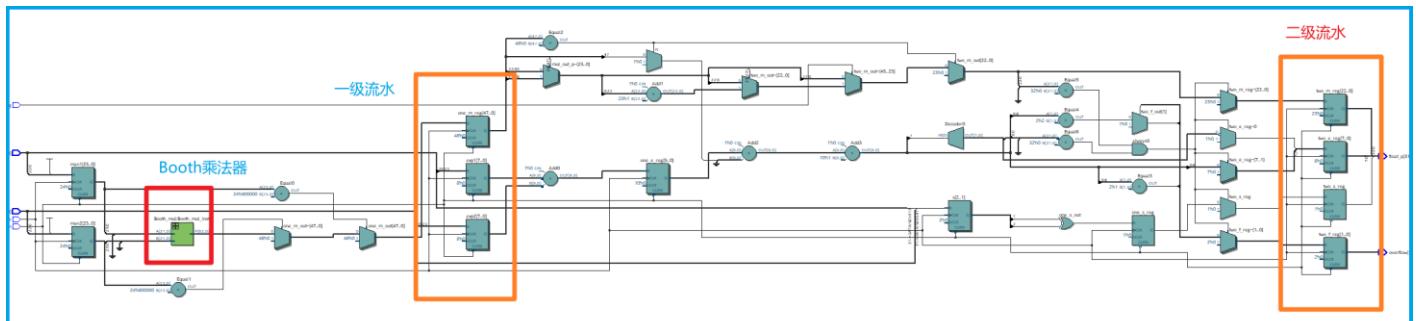
输入乘数, modelsim 仿真如下:



由于 2 级流水寄存，输出落后输入两个时钟周期。

通过计算，仿真结果正确。

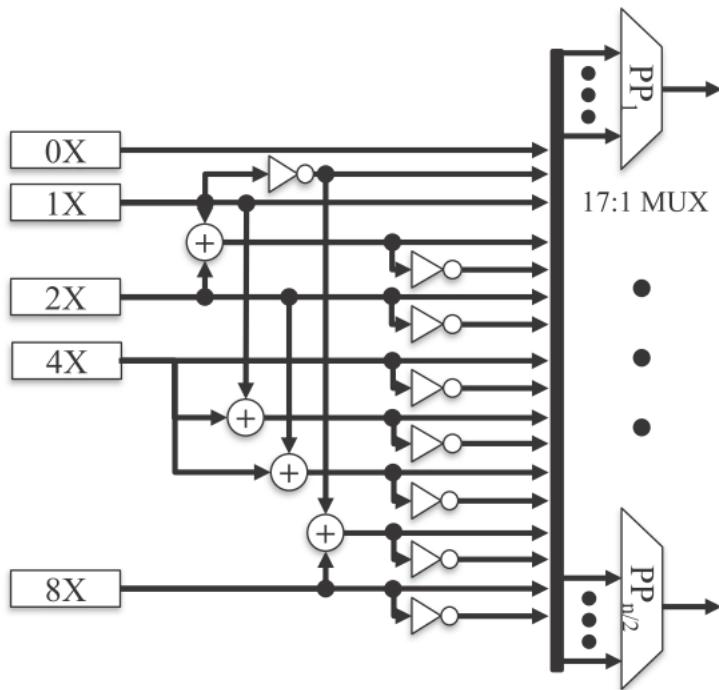
RTL 视图如下:



32 位单精度浮点型乘法器设计成功。

(三) 采用改进的 radix-16 编码改进 radix-4 编码乘法器

采用基 4 布斯编码的乘法相较于传统乘法运算，优化效果已经很明显且易于实现，可以满足大部分应用要求，32 位乘法器，甚至 64 位乘法器都可以采用，是比较常用的一种方式。本次创新采用基 16 布斯编码，部分积相对于基 4 又减少一半，加法操作优化效果更加明显。但由于部分积产生逻辑无法单纯通过移位实现，需要引入加法器等其它运算部件，从这方面来看又削弱了优化效果。如果采用基 16，需要用以下选择器^[1]：



(c) Radix-16 Booth Encoder

17-to-1 选择器会加大实现难度并且所需硬件更多，无法实现优化。

根据算式公式^[1]改进：

$$[[x_{i+1}, x_i, x_{i-1}]] = -2^1 x_{i+1} + 2^0 x_i + 2^0 x_{i-1}, \quad (2)$$

$$[[x_{i+2}, x_{i+1}, x_i, x_{i-1}]] = -2^2 x_{i+2} + 2^1 x_{i+1} + 2^0 x_i + 2^0 x_{i-1}, \quad (3)$$

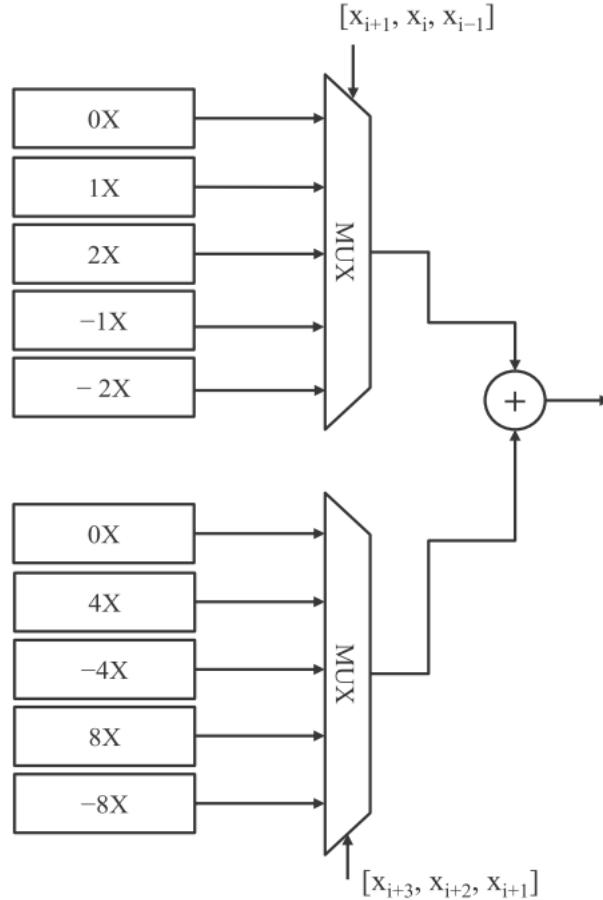
$$[[x_{i+3}, x_{i+2}, x_{i+1}, x_i, x_{i-1}]] = -2^3 x_{i+3} + 2^2 x_{i+2} + 2^1 x_{i+1} + 2^0 x_i + 2^0 x_{i-1}. \quad (4)$$

$$\begin{aligned} & [[x_{i+3}, x_{i+2}, x_{i+1}]] + [[x_{i+1}, x_i, x_{i-1}]] \\ &= -2^3 x_{i+3} + 2^2 x_{i+2} + 2^1 x_{i+1} + (-2^1 x_{i+1} + 2^0 x_i + 2^0 x_{i-1}) \\ &= -2^3 x_{i+3} + 2^2 x_{i+2} + 2^1 x_{i+1} + 2^0 x_i + 2^0 x_{i-1}, \end{aligned} \quad (5)$$

$$[[x_{i+3}, x_{i+2}, x_{i+1}, x_i, x_{i-1}]] = [[x_{i+3}, x_{i+2}, x_{i+1}]] + [[x_{i+1}, x_i, x_{i-1}]]. \quad (6)$$

公式 (2) 是 radix-4 编码, 公式 (4) 是 radix-16 编码, 而根据公式 (5) (6) 可轻易得到: 基 16 可由两个基 4 通过加法操作得到, 即,

17-to-1 mux 可有 2 个 5-to-1 mux 得来:



代码实现:

X _{i+3}	X _{i+2}	X _{i+1}	X _i	X _{i-1}	PP	X _{i+1}	X _i	X _{i-1}	Base PP	X _{i+3}	X _{i+2}	X _{i+1}	Weighted PP	2-stage PP (B.PP + W.PP)
0	0	0	0	0	0Y	0	0	0	0F	0	0	0	0F	0F
0	0	0	0	1	1Y	0	0	1	1F					
0	0	0	1	0	1Y	0	1	0	1F					
0	0	0	1	1	2Y	0	1	1	2F					
0	0	1	0	0	2Y	1	0	0	-2F	0	0	1	4F	2F
0	0	1	0	1	3Y	1	0	1	-1F					
0	0	1	1	0	3Y	1	1	0	-1F					
0	0	1	1	1	4Y	1	1	1	0F					
0	1	0	0	0	4Y	0	0	0	0F	0	1	0	4F	4F
0	1	0	0	1	5Y	0	0	1	1F					
0	1	0	1	0	5Y	0	1	0	1F					
0	1	0	1	1	6Y	0	1	1	2F					
0	1	1	0	0	6Y	1	0	0	-2F	0	1	1	8F	6F
0	1	1	0	1	7Y	1	0	1	-1F					
0	1	1	1	0	7Y	1	1	0	-1F					
0	1	1	1	1	8Y	1	1	1	0F					
1	0	0	0	0	-8Y	0	0	0	0F	1	0	0	-8F	-8F
1	0	0	0	1	-7Y	0	0	1	1F					
1	0	0	1	0	-7Y	0	1	0	1F					
1	0	0	1	1	-6Y	0	1	1	2F					
1	0	1	0	0	-6Y	1	0	0	-2F	1	0	1	-4F	-6F
1	0	1	0	1	-5Y	1	0	1	-1F					
1	0	1	1	0	-5Y	1	1	0	-1F					
1	0	1	1	1	-4Y	1	1	1	0F					
1	1	0	0	0	-4Y	0	0	0	0F	1	1	0	-4F	-4F
1	1	0	0	1	-3Y	0	0	1	1F					
1	1	0	1	0	-3Y	0	1	0	1F					
1	1	0	1	1	-2Y	0	1	1	2F					
1	1	1	0	0	-2Y	1	0	0	-2F	1	1	1	0F	-2F
1	1	1	0	1	-1Y	1	0	1	-1F					
1	1	1	1	0	-1Y	1	1	0	-1F					
1	1	1	1	1	0Y	1	1	1	0F					

根据查找表实现 Booth_Ctrl 模块：

```
7 module Booth_Ctrl
8 #(
9     parameter LENGTH = 32
10 )
11 (
12     input wire [LENGTH-1:0] a_i, //乘数 A
13     input wire [4:0] b_i, //被乘数 B 五位选择
14     output wire [LENGTH+3:0] bo_o
15 );
16
```

观察到，编码模块输出相对于 radix-4 位宽多 2 位

```
5
6 always @(*) begin //无符号低位
7     case(b_i[2:0])
8         3'b000: bo_o_l = 'b0;
9         3'b001: bo_o_l = {{3{a_tr[LENGTH]}}, a_tr};
10        3'b010: bo_o_l = {{3{a_tr[LENGTH]}}, a_tr};
11        3'b011: bo_o_l = {{2{a_tr[LENGTH]}}, a_tr, 1'b0};
12        3'b100: bo_o_l = -{{2{a_tr[LENGTH]}}, a_tr, 1'b0};
13        3'b101: bo_o_l = -{{3{a_tr[LENGTH]}}, a_tr};
14        3'b110: bo_o_l = -{{3{a_tr[LENGTH]}}, a_tr};
15        3'b111: bo_o_l = 'b0;
16    endcase
17 end
18
19 always @(*) begin //无符号高位
20     case(b_i[4:2])
21         3'b000: bo_o_m = 'b0;
22         3'b001: bo_o_m = {a_tr[LENGTH], a_tr, 2'b0};
23         3'b010: bo_o_m = {a_tr[LENGTH], a_tr, 2'b0};
24         3'b011: bo_o_m = {a_tr, 3'b0};
25         3'b100: bo_o_m = -{a_tr, 3'b0};
26         3'b101: bo_o_m = -{a_tr[LENGTH], a_tr, 2'b0};
27         3'b110: bo_o_m = -{a_tr[LENGTH], a_tr, 2'b0};
28         3'b111: bo_o_m = 'b0;
29    endcase
30 end
```

高位编码结果需左移若干位。

无符号实现，使用两个 5-to-1mux

有符号同理，下图：

```

3    else
4    always @(*) begin //有符号低位
5      case(b_i[2:0])
6        3'b000: bo_o_l = 'b0 ;
7        3'b001: bo_o_l = {{4{a_i[LENGTH-1]}}, a_i } ;
8        3'b010: bo_o_l = {{4{a_i[LENGTH-1]}}, a_i } ;
9        3'b011: bo_o_l = {{3{a_i[LENGTH-1]}}, a_i, 1'b0 } ;
0        3'b100: bo_o_l = -{{3{a_i[LENGTH-1]}}, a_i, 1'b0 } ;
1        3'b101: bo_o_l = -{{4{a_i[LENGTH-1]}}, a_i } ;
2        3'b110: bo_o_l = -{{4{a_i[LENGTH-1]}}, a_i } ;
3        3'b111: bo_o_l = 'b0 ;
4      endcase
5    end
6
7    always @(*) begin //有符号高位
8      case(b_i[4:2])
9        3'b000: bo_o_m = 'b0 ;
0        3'b001: bo_o_m = {{2{a_i[LENGTH-1]}}, a_i, 2'b0 } ;
1        3'b010: bo_o_m = {{2{a_i[LENGTH-1]}}, a_i, 2'b0 } ;
2        3'b011: bo_o_m = {a_i[LENGTH-1], a_i, 3'b0 } ;
3        3'b100: bo_o_m = -{a_i[LENGTH-1], a_i, 3'b0 } ;
4        3'b101: bo_o_m = -{{2{a_i[LENGTH-1]}}, a_i, 2'b0 } ;
5        3'b110: bo_o_m = -{{2{a_i[LENGTH-1]}}, a_i, 2'b0 } ;
6        3'b111: bo_o_m = 'b0 ;
7      endcase
8    end

```

最终通过加法器得到编码输出：

```
assign bo_o = bo_o_m + bo_o_l ;
```

顶层模块改进：

```

38 `ifdef UNSIGNED_BOOTH //是否选择为无符号
39 assign B_tr = {4'b0, B, 1'b0}; //最高位扩展1位0, 代表无符号, 再扩展三位0组成五位一组数
40
41 `else
42 assign B_tr = {{4{B[LENGTH-1]}}, B, 1'b0}; //扩展四位符号位
43
44
45
46

```

符号位扩展增加

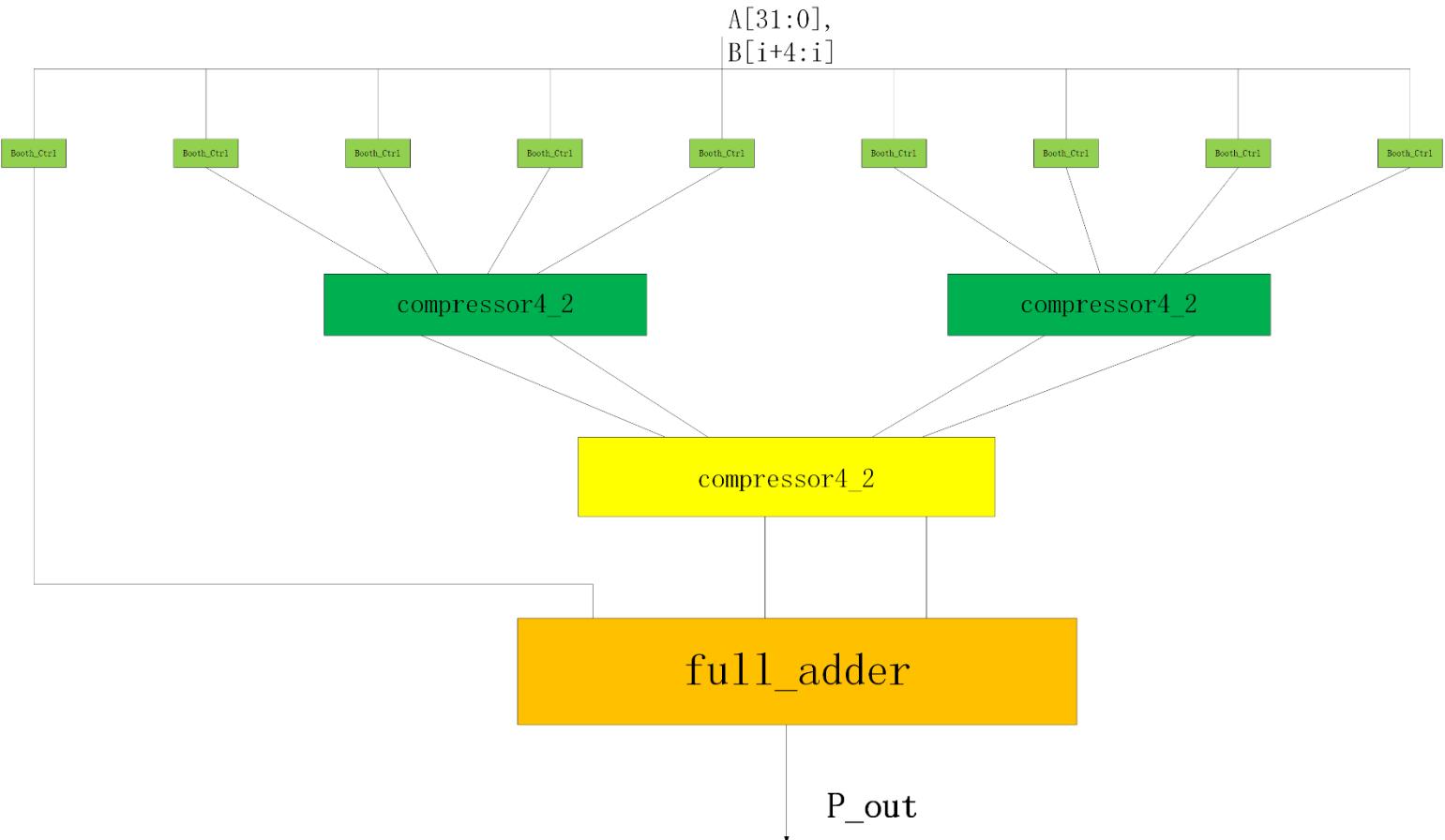
```

0 genvar i;
1 generate
2   for(i=0; i<LENGTH/4+1; i=i+1) begin: Booth_Ctrl_i
3     Booth_Ctrl
4     #(
5       .LENGTH(LENGTH)
6     )
7     Booth_Ctrl
8     (
9       .a_i      (A),
10      .b_i      ({B_tr[i*4+5]}),
11      .bo_o     (boo_o[i])
12    );
13   if(i != 8)
14     assign pp[i] = {{(LENGTH-4-i*4){boo_o[i][LENGTH+3]}}, boo_o[i], {(i*4){1'b0}}}; //部分积
15   else
16     assign pp_8th = {boo_o[i], {(i*4){1'b0}}}; //扩展位所产生的第 9 个部分积
17   end
18 endgenerate

```

部分积减少为 9 个，第 9 个由于扩展而生成。

Wallace 加法树减少一级，即只用到两级压缩器：



仿真结果如下：

```

# Output P = 100011110101111110000011111111010101011100000101011011011001100
# Output P1 = 100011110101111110000011111111010101011100000101011011011001100
# Test Passed: Output matches expected result
# Total Tests:      2000
# Passed:          2000
# Failed:          0
# //=====Random test all Pass!!=====//

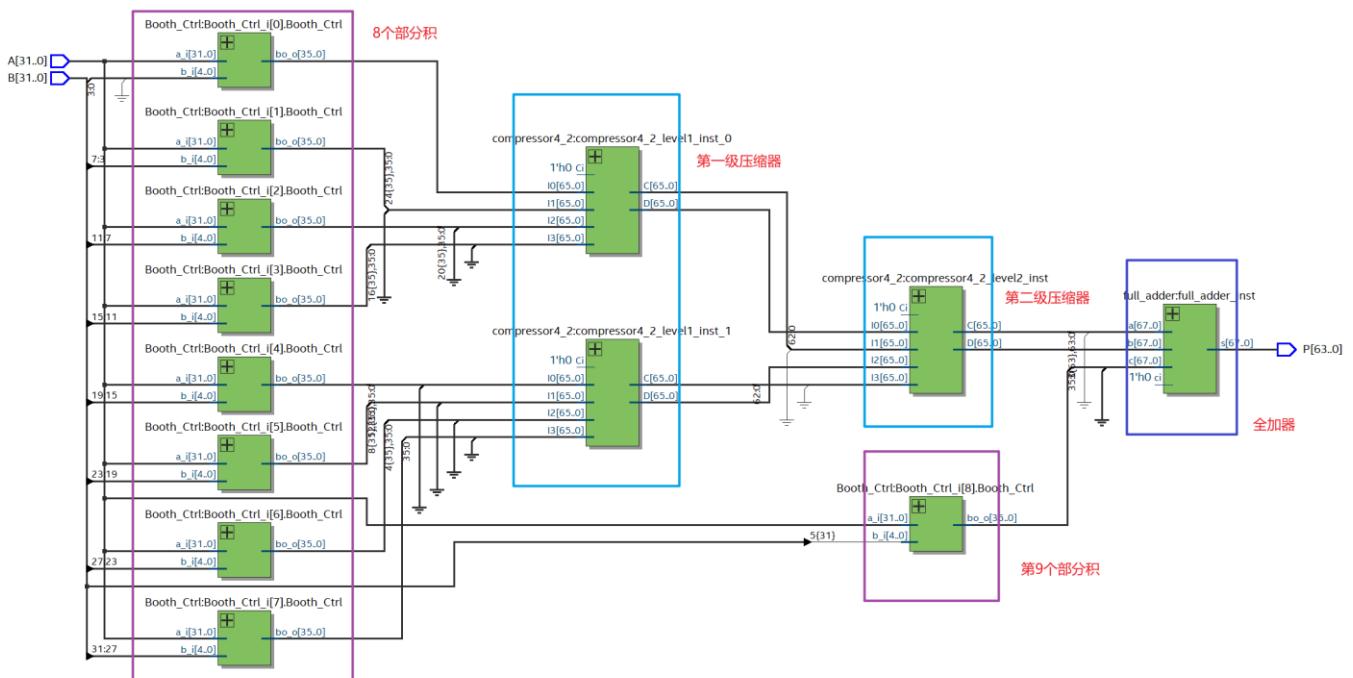
```

	Msgs
+ A	32'hfaa0f5
+ B	32'h47a9a38f
+ P1	64h462cd3bf105...
+ P	64h462cd3bf105...
+ i	32'h0000037a
+ pass_count	32'h0000037a
+ fail_count	32'h00000000
	00000000

有符号同理：

	Msgs
+ /tb_Booth_mul/A	32'ha538004a
+ /tb_Booth_mul/B	32'd621099338
+ /tb_Booth_mul/P1	-64d9459694248...
+ /tb_Booth_mul/P	-64d9459694248...
+ /tb_Booth_mul/i	32'd571
+ /tb_Booth_mul/pass...	32'h0000029f
+ /tb_Booth_mul/fail...	32'h00000000
	00000000

RTL 视图：



仿真成功，改进成功。

资源消耗对比：

il	Flow Status	Successful - MON OCT 07 16:19:58 2024
il	Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Standard E
il	Revision Name	Booth_mul
il	Top-level Entity Name	Booth_mul
il	Family	Cyclone IV E
s	Total logic elements	2,568 / 6,272 (41 %)
s	Total registers	0
s	Total pins	128 / 180 (71 %)
s	Total virtual pins	0
s	Total memory bits	0 / 276,480 (0 %)
s	Embedded Multiplier 9-bit elements	0 / 30 (0 %)
le	Total PLLs	0 / 2 (0 %)
le	Device	EP4CE6F17C6
le	Timing Models	Final

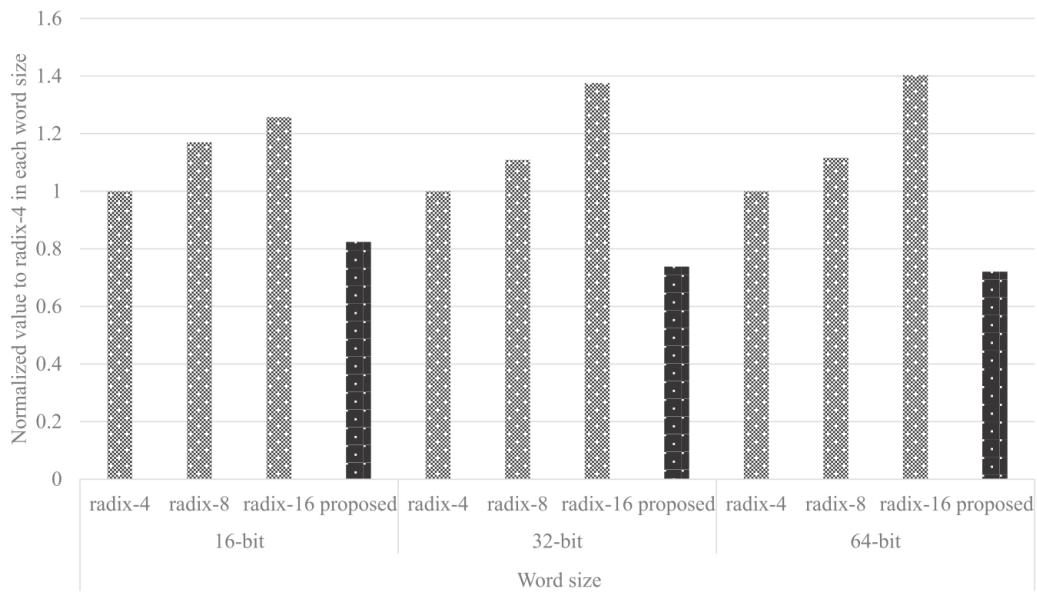
改进的 Radix-16

e	Revision Name	Booth_mul
y	Top-level Entity Name	Booth_mul
y	Family	Cyclone IV E
ysis	Total logic elements	3,043 / 6,272 (49 %)
ysis	Total registers	0
r	Total pins	128 / 180 (71 %)
r	Total virtual pins	0
r	Total memory bits	0 / 276,480 (0 %)
r	Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Me	Total PLLs	0 / 2 (0 %)
Me	Device	EP4CE6F17C6
Me	Timing Models	Final

Radix-4

改进的基 16 逻辑资源明显低于基 4.

PD 对比：



Column proposed 为设计成功。优化效果较 radix-4 明显

三、结论

采用 radix-4 Booth 编码以及 Wallace 加法树方法，成功实现设计 32 位无符号整型快速乘法器，同时可以实现有无符号功能的选择。TB 仿真自动测试输出结果对比。并在此基础上，成功实现改进 4 级流水的 Booth 编码器，2 级流水的符合 IEEE754 标准的 32 位单精度浮点型乘法器，与改进的 radix-16 Booth 编码改进三种创新。

参考文献

- [1] Ch V S Chaitanya, PSathish Kumar. Design and Analysis of Booth Multiplier with Optimised Power Delay Product[J]. International Conference on Computer Communication and Informatics, Jan. 04 - 06, 2018.
- [2] Hyunpil Kim, Sangook Moon, and Yongsurk Lee. Radix-16 Booth multiplier using novel weighted 2-stage Booth algorithm[J]. IEICE Electronics Express, Vol. 11, No. 13, 1-8