Booth Multiplier
Implementation of Booth’s Algorithm using Verilog RTL

VLSI IP
Comments welcome on avimit@yahoo.com
Web: http://www.vlsiip.com

Abstract:

This paper presents a description of booth’s algorithm for multiplication two binary numbers. Radix-2 booth’s algorithm is explained, it is then identified that the main bottleneck in terms of speed of the multiplier is the addition of partial products. Radix-4 Booth’s algorithm is presented as an alternate solution, which can help reduce the number of partial products by a factor of 2.The booht’s multiplier is then coded in verilog, and area and timing analysis is performed on it. Radix-4 Booth’s multiplier is then changed the way it does the addition of partial products. Carry-Save-Adders are used to add the partial products. Results of timing and area are then shown. The results table contain area and timing results of 3 multipliers i.e ordinary array multiplier, radix-4 booth’s multiplier (without CSA), and radix-4 booth’s multiplier with CSA. Results are then discussed. The technology used is 0.35u MTC45000 form Alcatel

For full verilog code of the radix-4 booth’s multiplier see Appendix

Section 1:Booth Algorithm

1.1 Explanation of Booth Algorithm

First radix 2 booth algorithm is explained, and using the radix-2 booth algorithm, radix-4 will be explained.

One of the ways to multiply signed number was invented by Booth.

Let us consider a Multiplicand M 'n' bits wide represented as M_{n-1} M_{n-2} ..... M_2 M_1 M_0 and a Multiplier 'R' again 'n' bits wide represented as R_{n-1} R_{n-2} .....R_2 R_1 R_0. Both of these are signed(two's compliment) binary numbers. As per Booth's algorithm,

$M \times R = M \times \{(S_{n-1} \times 2^{n-1}) + (S_{n-2} \times 2^{n-2}) .....(S_2 \times 2^2) + (S_1 \times 2^1) + (S_0 \times 2^0)\}$ - equation (1)

where each S_k for, n-1<=k<=0, is a value which depends upon the value of R, and can be found as explained in the following steps

1) Append R by a '0' on LSB, we will called this bit as Z

2) Now make collections of 't' bits, where 't' = 2, for radix 2 booth algorithm, and name each collection C_k , where n-1<=k<=0,

The rule to make each collection C_k is such that $C_k = (R_k R_{k-1})$, if n-1<=k<=1, and $C_k = (R_k Z)$ for k= 0.

This process will result in 'n' collections, such that:

$C_{n-1}= (R_{n-1}R_{n-2})$, ...$C_1=(R_1R_0)$, $C_0=(R_0Z)$

3). Now depending upon the value of C_k, where n-1<=k<=0, find out S_k, where the value of 'S_k' is defined in the following table for all possible combinations of values of a pair C_k.

Table 1:

C_k	S
00	0
01	+1
10	-1
11	0

Now equation(1) on page 3 can be re-written as

$M \times R = (M \times p_{n-1}) + (M \times p_{n-2}) .....+ (M \times p_1) + (M \times p_0)$

where, $p_{n-1}= S_{n-1} \times 2^{n-1}$, $p_{n-2}=S_{n-2} \times 2^{n-2}$.....$p_1=S_1 x 2^1$, $p_0= S_0 x 2^0$

$M \times R = pp_{n-1} \times 2^{n-1} + pp_{n-2} \times 2^{n-2} .....+ pp_1 \times 2^1 + pp_0 \times 2^0$--- equation (2)

where $pp_{n-1}=(M \times p_{n-1})$,$pp_{n-2}=(M \times p_{n-2})$,...$pp_1=(M \times p_1)$, $pp_0=(M \times p_0)$ are called partial products.

4). Add these 'n' partial products as shown in the equation below to get final product.

so $M \times R = pp_{n-1} \times 2^{n-1} + pp_{n-2} \times 2^{n-2}$..... $+pp_1 \times 2^1 + pp_0 \times 2^0$ --- equation (3)

Note that equation(3) is same as equation(2), reproduced for clarity.

Now let us take an example of two numbers, such that M = 10110(-10), R= 10011(-13), and try to evaluate MxR using the algorithm explained above.
Given that R = 10011,
we append Z to R to make the new R as 10011Z, where Z = 0,
so new R = $\overline{1}\ \underline{0}\ \overline{0}\ \underline{1}\ \overline{1}\ \underline{0}$,

now, clearly

$C_4 = 10$, $S_4 = -1$
$C_3 = 00$, $S_3 = 0$
$C_2 = 01$, $S_2 = +1$
$C_1 = 11$, $S_1 = 0$
$C_0 = 10$, $S_0 = -1$

$pp0 = M*S_0$
$pp1 = M*S_1$
$pp2 = M*S_2$
$pp3 = M*S_3$
$pp4 = M*S_4$

now to obtain the final product, we will use equation(3) on page 4.

final product = $pp_{n-1} * 2^{n-1} + pp_{n-2} * 2^{n-2}..... + pp_1 * 2^2 + pp_0 * 2^0$

```
          0 1 0 1 0        (+10)
        0 0 0 0 0          (0)
1 1 1 1 0 1 1 0            (-40)
    0 0 0 0 0              (0)
  0 1 0 1 0                (+160)
0 0 1 0 0 0 0 0 1 0        (+130)
```

Figure 1: shifting, sign extension and adding of partial products

Remember we will have to sign extend the partial products, to make them equal in width before adding.

So the result we got is +130, that is true because we had M = (-10) and R = (-13)

Radix-4 Booth Algorithm:

Motivation:

The main bottleneck in the speed of multiplication is the addition of partial products. More the number of bits the multiplier/multiplicand is composed of, more are the number of partial products, longer is the delay in calculating the product.The critical path of the multiplier depends upon the number of partial products. In radix-2 booth's algorithm, if we are multiplying 2 'n' bits number, we have 'n' partial products to add.

Radix-4 booth's multiplication is an answer to reducing the number of partial products. Using Radix-4 booth's multiplier, the number of partial products are reduced to 'n/2' if we are multiplying two 'n' bits numbers, if 'n' is even number, or '(n+1)/2' , if 'n' is an odd number. By reducing the number of partial products, one can effectively speed up the multiplier by a factor roughly equal to 2.

Radix-4 Booth Algorithm Explained:

Now, at page no 3, Step 2, if we take 't' = 3, and make collections of 't' bits taken in one C_k, where N-1<=k<=0 where N = n/2, for 'n' is even, N=(n+1)/2 if 'n' is odd. from the multiplier R, such that

$C_k = (R_{2k+1}R_{2k}R_{2k-1})$, for N-2<=k<=1, for

$C_k = (R_{2k+1}R_{2k}Z)$, for k = 0, and Z=0

$C_k = (R_{2k+1}R_{2k}R_{2k-1})$, if 'n' is even, and k = N-1

$C_k = (R_{2k}R_{2k}R_{2k-1})$, if 'n' is odd, and k = N-1,note that the sign bit is repeated i.e R_{2k} can be seen in two of the bits in Ck

We might have to sign-extend 'M' by one bit, if the C_{N-1} contains less than 't' or 3 bits. This will always happen, when 'n' is odd. The number of partial products will also decrease from 'n' to 'N'. All the partial products in equation(3) at page 4, which are multiplied by 2^x, where 'x' is odd will dissapear.That means instead of shifting a partial product (before summing them up for the final product) by '1' bit i.e multiplying it by 2^1 in each partial product, we will shift by 2 bits instead, i.e multiplying it by 2^2.

Table 1 will now change to Table 2 given below:

Table 2:

C_k	S_k
000	0
001	+1
010	+1
011	+2
100	-2
101	-1

Table 2:

C_k	S_k
110	-1
111	0

Final product equation will now become:
$MxR=pp_{N-1} \times 2^{2*(N-1)} + pp_{N-2} \times 2^{2*(N-2)} .....+ pp_1 \times 2^2 + pp_0 \times 2^0$ equation (4),
where
$pp_{N-1}=M*S_{N-1}$, $pp_{N-2}=M*S_{N-2}$..., $pp_1=M*S_1$,$pp_0=M*S_0$
pp_k are called partial products
The full radix-4 booth multiplier equation can be written as
$MxR=M*S_{N-1} \times 2^{2*(N-1)} + M*S_{N-2} \times 2^{2*(N-2)} .....+ M*S_1 \times 2^2 + M*S_0 \times 2^0$ equation (5),
Note that all the terms which contain multiplication by 2 to the power 'x', where 'x' is an odd number have disappeared, suggesting that while addition of the partial products, each partial product will be shifted by 2 bits instead on 1 bit.

We will take the same example as we took in radix-2 booth's multiplier to show the working of the algorithm.
M = 10110(-10), R= 10011(-13)
'n' = 5 bits.
since n is odd, N = (n+1)/2 = 3

R = $\overline{\text{Sign 1 0 0 1 1 Z}}$, substituting 'Sign' bit for '1', and Z for '0', R = $\overline{1\ 1}\ \underline{0\ 0\ 1}\ \overline{1\ 0}$

C_0 = 110, so S_0 = -1
C_1 = 001, so S_1 = +1
C_2 = 110, so S_2 = -1

so our partial products are
$pp_0 = M*S_0$
$pp_1 = M*S_1$
$pp_2 = M*S_2$
$M \times R = pp_2 * 2^4 + pp_1 * 2^2 * pp_0 * 2^0$
$MxR = 11010(-10) * (-1) * 2^4 + 11010(-10) * (+1) * 2^2 + 11010(-10) * 2^0 * (-1)$

```
          0 1 0 1 0      (+10)
1 1 1 1 0 1 1 0          (-40)
  0 1 0 1 0              (+160)
0 0 1 0 0 0 0 0 1 0      (+130)
```

Figure 2: shifting, sign extension and adding of partial products

Again, the answer was found to be +130, which is corret because M=(-10) and R=(-13)
Note
1) Each partial product is being sifted 2 places
2) That the number of partial products have been reduced in radix-4 algorithm to half

Section 1.2 Design of a Radix-4 Booth Multiplier using verilog.

Booth's Multiplier can be either a sequential circuit, where each partial product is generated and accumulated in one clock cycle, or it can be purely combinational, where all the partial products are generated in parallel.
Our objective is to do a combinational multiplier.
The analytical expression of radix-4 booth's multiplier is given in equation(5), which is reproduced here.

$MxR=pp_{N-1} \times 2^{2*(N-1)} + pp_{N-2} \times 2^{2*(N-2)} .....+ pp_1 \times 2^2 + pp_0 \times 2^0$ equation (4),

where

$pp_{N-1}=M*S_{N-1}$, $pp_{N-2}=M*S_{N-2}$..., $pp_1=M*S_1$,$pp_0=M*S_0$

pp_k are called partial products

$MxR=M*S_{N-1} \times 2^{2*(N-1)} + M*S_{N-2} \times 2^{2*(N-2)} .....+ M*S_1 \times 2^2 + M*S_0 \times 2^0$

where N = n/2 if 'n' is even, N = (n+1)/2, if 'n' is odd. Note 'n' is the total number of bits in a vector, not the index of the MSB of a vector. Index value or MSB of the vector will therefore be 'n-1'
All S_k can be found by looking into Table 2, values of C_k, and method to find C_k are described at page 5
Clearly, our objective is to find all the partial products and add them together, after shifting them by appropriate number of bits.

$S_k = f(R_{2k+1}R_{2k}R_{2k-1})$, if k = 0, then instead of R_{2k-1} use 0, if 2k+1 > 'n-1' then use R_{2k} instead of R_{2k+1} Expression --(1)
For sake of simplicity, 'n' is assumed to be an even number, as this would be the case in majority of the designs done. For the assignment 'n' is already given to be '16'.

The design:

What we need:
1). We need to generate values : -1*M, -2*M, M, 2*M, where M is the multiplicand.
2). We need to generate C_k,S_k, for each k
3). Partial Products
4). A way to add partial products.Taking into account the shifting of the partial products, such that their sign bits are preserved, and each shifting is basically shifting by 2 bits instead of 1 bit in radix-4 booth's algorithm.

1). Generating M is not required, it is already present as input, 2*M is simply M shifted left by one bit, i.e appending a '0' to LSB of M.(remember to extend M by at least one bit, so that 2*M can be contained in the number of bits
Generating -M, and -2*M: (-1*M) is twos compliment of M, -2*M is the two's compliment of -1*M shifted by one bit i.e an '0' bit added to the LSB of (-1*M)

2). Generating C_k, S_k.
Use table 2 for C_ks and directly the Expression (1) shown above to generate these S_ks
3). partial products. Select one of (0, -M, -2M, M, 2M) depending upon S_k, and these will be the partial products(unshifted)

4). A way to add partial products:
Shift each partial product by 2 bits, as shown in Figure 2 on page(7), to get the final product.

The following compact verilog code is self-explanatory, and implements all the 4 steps described above, to generate the final product 'prod' using 'x' in place of M and 'y' in place of 'R'.

```
assign inv_x = {~x[width-1],~x}+1; //generate two's compiment of multiplicand x(M)
always @ (x or y or inv_x)
begin
  cc[0] = {y[1],y[0],1'b0}; //generate Ck for k=0(special case)
  for(kk=1;kk<N;kk=kk+1)
    cc[kk] = {y[2*kk+1],y[2*kk],y[2*kk-1]}; //generate Ck for each k, for k is not 0
  for(kk=0;kk<N;kk=kk+1)
  begin
    case(cc[kk]) //Depending upon Ck, select M,2M,-M,-2M, or 0 as the partial product
      3'b001 , 3'b010 : pp[kk] = {x[width-1],x};
      3'b011 : pp[kk] = {x,1'b0};
      3'b100 : pp[kk] = {inv_x[width-1:0],1'b0};
      3'b101 , 3'b110 : pp[kk] = inv_x;
      default : pp[kk] = 0;
    endcase
    spp[kk] = $signed(pp[kk]);//sign extend
    for(ii=0;ii<kk;ii=ii+1)
      spp[kk] = {spp[kk],2'b00}; //multiply by 2 to the power x or shifting operation
  end //for(kk=0;kk<N;kk=kk+1)
  prod = spp[0];
  for(kk=1;kk<N;kk=kk+1)
    prod = prod + spp[kk]; //add partial products to get result
end
assign p = prod;
```

The above RTL code successfully implements the radix-4 booth's algorithm. The simulation of this booth's multiplier gave correct results. This also proves that the alorithm given in this report is correct. The design is parameterized, and just changing the value of 'width' a new booth's multiplier is ready. That is it is designed for re-use.
However, this booth's multiplier uses '+' for addition. Which when synthesized can produce adders with long delays, and therefore it will affect the overall performance of the multiplier. In order to cut down on 'delays', we might choose carry-save-adders or carry-look-ahead address, to minimise the critical path of the adder, and speed up the multiplier.

Following is the code for a booth multiplier using carry save adders to add the partial products. Note that the number of carry save adders will be quite large and it will greatly impact the area of the design.
Also note that in this design, only partial product needs addition, there is no 'carry' associated with each partial product as it is in case of the code given in the assignment template. This is so because the partial product is a full 2's compliment (whenever required), and the carry has already been added, unlike the one given in the assignment notes where partial product is only 1's compliment, and the addition of '1' to one's compliment to make it 2's compliment, has been propagated as a 'carry'.

The following RTL implements the booth's multiplier using carry save adders.

```
assign inv_x = {~x[width-1],~x}+1; //generate two's compiment of multiplicand x(M)
always @ (x or y or inv_x)
begin
  cc[0] = {y[1],y[0],1'b0}; //generate Ck for k=0(special case)
  for(kk=1;kk<N;kk=kk+1)
    cc[kk] = {y[2*kk+1],y[2*kk],y[2*kk-1]}; //generate Ck for each k, for k is not 0
  for(kk=0;kk<N;kk=kk+1)
  begin
    case(cc[kk]) //Depending upon Ck, select M,2M,-M,-2M, or 0 as the partial product
      3'b001 , 3'b010 : pp[kk] = {x[width-1],x};
      3'b011 : pp[kk] = {x,1'b0};
      3'b100 : pp[kk] = {inv_x[width-1:0],1'b0};
      3'b101 , 3'b110 : pp[kk] = inv_x;
      default : pp[kk] = 0;
    endcase
    spp[kk] = $signed(pp[kk]);//sign extend
    for(ii=0;ii<kk;ii=ii+1)
      spp[kk] = {spp[kk],2'b00}; //multiply by 2 to the power x or shifting operation
  end //for(kk=0;kk<N;kk=kk+1)
end

assign sum0 = $signed(spp[0]);
assign sum1 = $signed(spp[1]);
assign sum2 = $signed(spp[2]);
assign sum3 = $signed(spp[3]);
```

```
assign sum4 = $signed(spp[4]);
assign sum5 = $signed(spp[5]);
assign sum6 = $signed(spp[6]);
assign sum7 = $signed(spp[7]);

//add all partial produces using carry save adders.
csa_32 csa_32_0 ( .s1(sum8), .s2(sum9), .p1(sum0), .p2(sum1), .p3(sum2), .cin(1'b0));
csa_32 csa_32_1 ( .s1(sum10), .s2(sum11), .p1(sum3), .p2(sum4), .p3(sum5), .cin(1'b0));
csa_32 csa_32_2 ( .s1(sum12), .s2(sum13), .p1(sum6), .p2(sum7), .p3(32'b0), .cin(1'b0));
csa_32 csa_32_3 ( .s1(sum14), .s2(sum15), .p1(sum8), .p2(sum9[31:0]), .p3(sum10), .cin(1'b0));
csa_32 csa_32_4 ( .s1(sum16), .s2(sum17), .p1(sum11[31:0]), .p2(sum12), .p3(sum13[31:0]),
.cin(1'b0));
csa_32 csa_32_5 ( .s1(sum18), .s2(sum19), .p1(sum14), .p2(sum15[31:0]), .p3(sum16),
.cin(1'b0));
csa_32 csa_32_6 ( .s1(sum20), .s2(sum21), .p1(sum18), .p2(sum19[31:0]), .p3(sum17[31:0]),
.cin(1'b0));
assign p = sum20+sum21;
```

Section 1.4 Results discussion

It is required to perform a timing and area analysis on 2 types of multipliers i.e an ordinary array multiplier and a booth's multiplier, so that the need of a booth's multiplier can be appreciated. For this purpose, the ordinary multiplier, and two types of booth's multiplier were designed, and synthesized, on target library MTC45000. Each of these multipliers were optimised for timing and area separately. Timing being the main criteria and constraint.
The process followed for optimisation:
All the multipliers were first synthesized and constrained for area, and minimum area was found irrespective of the timing. All areas were recorded.

All multipliers were then optimized for timing, and the delay of the critical path was recorded. All the values from above two steps are recorded and tabulated for reference below.

Table 3:

Multiplier type	ordinary multiplier	booths multiplier **without** csa	booths multiplier **with** csa
delay of the critical path in ns	23.40	16.61	14.56
area at the time of min timing	2474.83	3181.78	4541.12
min area sq microns	1833.92	1375.06	2095.02

From the above table, it is gathered that while the design was optimized for timing, that is for the most important constraint in the exercise, booth multiplier with csa has a critical path delay of 14.56 ns, booth multiplier(without csa) has a critical path delay of 16.16 ns as compared to the critical path of ordinary multiplier which was reported to be 23.4 ns. So booth’s multiplier with csa is best as far as the timing is concerned.
clearly booth multiplier (with or without csa) outperformed the ordinary multiplier when optimized for timing.

As far as area is concerned, booths multiplier with csa has the maximum area of 4541 units, among all, which is obvious, because we used so many adders for parallel processing.
Booth multiplier without csa have even less area than the ordinary multiplier,(when optimized only fro area) because the number of adders are reduced to half of the number of adders is ordinary multiplier because there are only half of partial products in a radix-4 booth’s multiplier.

Conclusion:
1). Use booth’s multiplier with csa if area is not critical.
2). Use booth’s multiplier without csa if area is critical and a bit of compromise on timing can be made. In the above example, 1360 units of area are used to improve the timing by ~2ns. In percentages, 42% area is increased to get 13.7 % reduction in timing. So the best choice is booth’s multiplier without csa. This design is also parameterized, giving a high degree of re-use.

APPENDIX

A1 Booth Multiplier Verilog Code(without csa)

```
////////////////////////////////////////////////////////////////////////////////
//                    Coypright (C) Aviral Mittal.
////////////////////////////////////////////////////////////////////////////////
//   All rights reserved. Reproducion in whole or in part is prohibited without
//   written consent of copyright Owner.The professional use of this material
//   is subject to the copy right owners approval in written.
////////////////////////////////////////////////////////////////////////////////
//   Comments welcome on aviral.mittal@sli-institute.ac.uk or avimit@yahoo.com
////////////////////////////////////////////////////////////////////////////////
// It is a radix-4 booth's multiplier. It will work ONLY if width is even Number
‘define width16
‘timescale 1ns/1ps
module booth_mult (p, x, y);
parameter width=‘width;
parameter N = ‘width/2;
input[width-1:0]x, y;
output[width+width-1:0]p;
  reg [2:0] cc[N-1:0];
  reg [width:0] pp[N-1:0];
  reg [width+width-1:0] spp[N-1:0];
  reg [width+width-1:0] prod;
  wire [width:0] inv_x;
  integer kk,ii;

assign inv_x = {~x[width-1],~x}+1;
always @ (x or y or inv_x)
begin
  cc[0] = {y[1],y[0],1’b0};
  for(kk=1;kk<N;kk=kk+1)
    cc[kk] = {y[2*kk+1],y[2*kk],y[2*kk-1]};
  for(kk=0;kk<N;kk=kk+1)
  begin
    case(cc[kk])
      3’b001 , 3’b010 : pp[kk] = {x[width-1],x};
      3’b011 : pp[kk] = {x,1’b0};
      3’b100 : pp[kk] = {inv_x[width-1:0],1’b0};
      3’b101 , 3’b110 : pp[kk] = inv_x;
      default : pp[kk] = 0;
    endcase
    spp[kk] = $signed(pp[kk]);
    for(ii=0;ii<kk;ii=ii+1)
      spp[kk] = {spp[kk],2’b00}; //multiply by 2 to the power x or shifting operation
  end //for(kk=0;kk<N;kk=kk+1)
  prod = spp[0];
```

```
  for(kk=1;kk<N;kk=kk+1)
    prod = prod + spp[kk];
end
assign p = prod;
endmodule
```