Implement MAC

Yu-Chi Chu

06 Apr 2024

Abstract:

In this design, I implemented a radix-4-Booth's multiplier. I'll illustrate the concept of radix-2 multiplier first, and then explain why radix-2 itself is less efficient than the radix-4 version. Finally, I'll introduce the radix-4 version and give a brief summary of this project.

Concept of radix-2-Booth's multiplier:

1) Basics:

Before diving into Booth's algorithm, let's consider a multiplication with multiplicand M: 01011, and multiplier R: 01110. Booth observed that if a number with a continued 1's sequence, then this number can be calculated as followed: $R = 2^4-2^1$. Therefore, instead of doing M*Q in a naïvely every-bit-multiplication way, M*Q can be obtained through this modified equation:

$$2^4(M) - 2^1(M) = 2^4(M) + 2^1(2\text{'s comp. of } M)$$

With this explained, one can easily conclude that Booth's algorithm is to reduce the computation time by **exploiting continued 1's sequence**, hence the worst-case scenario is when multiplier is a bit-sequence of 01010101…, with 0 and 1 alternating.

2) Generalization:

In Booth's algorithm, considered a n-bit **Multiplicand** '**M**' represented as $M_{n-1}\ M_{n-2}\ldots\ldots\ M_2\ M_1M_0$ and a n-bit **Multiplier** '**R**' represented as $R_{n-1}R_{n-2}\ldots\ldots R_2\ R_1\ R_0$. Both of these are signed (two's compliment) binary numbers. In raidx-2 version, we collect multiplier in every 2 bits, and we do specific operation with respect to the bit-pattern in each collection.
Let's construct a table with C_k and S_k, where C_k is the 2-bit collection term, and S_k is the specific operation corresponding to the given collection.

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

Table 1

The rule to make each collection C_k is such that $C_k = (R_kR_{k-1})$, for k ranging from 1 to n-1, and $C_k = (R_kZ)$ for k= 0, where Z is the appended bit to the LSB of R, its value is 0. This process will result in 'n' collections, such that:

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

As per Booth's algorithm, we have this equation:

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

- equation (1)

Now equation (1) can be rewritten as

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

- equation (2)

where, $p_{n-1} = S_{n-1} * 2^{n-1}$, $p_{n-2} = S_{n-2} \times 2^{n-2}$ ..... $p_1 = S_1 * 2^1$, $p_0 = S_0 \times 2^0$. Eqa. (2) can be further rewritten as

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

- equation (3)

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

3) Example demonstration:

Given that M = 10110(-10), R= 10011(-13). We append Z to R to make the new R as 10011Z, where Z = 0, so new R = $\overline{1}\,0\,0\,\overline{1}\,\overline{1}\,0$, then we have collections as followed:

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

$pp_0 = M*S_0$
$pp_1 = M*S_1$
$pp_2 = M*S_2$
$pp_3 = M*S_3$
$pp_4 = M*S_4$

Final product = $pp_4 * 2^4 + pp_3 * 2^3 + pp_2 * 2^2 + pp_1 * 2^1 + pp_0 * 2^0$. Shown in figure 1.

```
      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

4) Radix-2 Bottleneck:

Please note that there are 5 partial products to add in this example, meaning with n-bit multiplication, there will be a number of n partial products to add. Nominally, the critical path of the multiplier depends upon the number of partial products existing. The radix-4 version is the solution to this problem, which reduces the number of partial products by half.

Radix-4-Booth's multiplier:

For n-by-n-bit multiplication, radix-4 version can reduce the number of partial products to 'n/2', if n is even, '(n+1)/2' if n is odd. The main difference between radix-2 and radix-4 is the bits difference in collection term. Here in radix-4 version, the collection is now in term of 3-bit.

Therefore, we can construct a new table relating C_k and S_k together like the way we've done above. Shown in table 2. Noted that N = n/2, for 'n' is even, N=(n+1)/2 if 'n' is odd.

$C_k = (R_{2k+1}R_{2k}R_{2k-1})$, for k ranging from 1 to N-2.

$C_k = (R_{2k+1}R_{2k}Z)$, 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.

C_k	S_k
000	0
001	+1
010	+1
011	+2
100	-2
101	-1
110	-1
111	0

Table 2

The full radix-4 booth multiplier equation can be written as

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

-equation (4)

Example demonstration:

Given the same example as above, M = 10110(-10), R= 10011(-13), since n is odd, N = (n+1)/2 = 3. New R = $\overline{\text{Sign}}\ 1\ \overline{0\ 0}\ \overline{1\ 1}\ Z$. Visualization is shown in Fig.2.

The collection terms are:

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:

$pp_0 = M*S_0$

$pp_1 = M*S_1$

$pp_2 = M*S_2$

Therefore, M x R is as follows

M x R = $pp_2 * 2^4 + pp_1 * 2^2 + pp_0 * 2^0$ = 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

Please note that each partial product is begin shifted 2 bits leftward at the current position.

Hardware Architecture:

1) Preface:

After understanding the algorithm detail, we shall now dive into the implementation of radix-4-Booth's multiplier. For a 8-by-8-bit multiplication, we can give that N for multiplier(R) is 4, implying we're going to have 4 partial products, and the collection terms are :

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

$C_k = (R_{2k+1}R_{2k}R_{2k-1})$, for k ranging from 1 to N-2.

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

2) Implementation techniques:

a. First encode every collection term from R in parallel→ Booth_enc.
b. Then decode the collection terms to generate partial products→Gen_prod.
c. Shift every partial product to their corresponding position and add them up.

3) Block diagram

In Fig.3, I instantiate 4 booth encoders in parallel, and then send the flags (neg, two, one, zero) to gen_prod units for producing partial products. Note that I realize addition operations by utilizing carry-save adders (CSA) instead of ripple-carry adders (RCA). Nevertheless, RCA is used in the last stage of addition. Therefore, the time-consuming operation of adding partial products can be further accelerated. However, due to the small amount of partial products in this practice, the speed up by CSAs is not significant. The area that CSA occupied has become the overhead of this design **if the number of partial products is larger**.

In conclusion, using CSA is of better performance in both speed and area in this 8-by-8-bit-multiplication practice. Please note that inside CSA and RCA are many full adders (FA). CSA is just another transformation of FA, which has 3 inputs and 2 outputs. The Verilog code is written in a structural way in this practice.

4) Files
   1. Booth_enc.v
   2. Gen_prod.v
   3. FA.v
   4. HA.v
   5. CSA.v
   6. RCA.v
   7. MAC.v

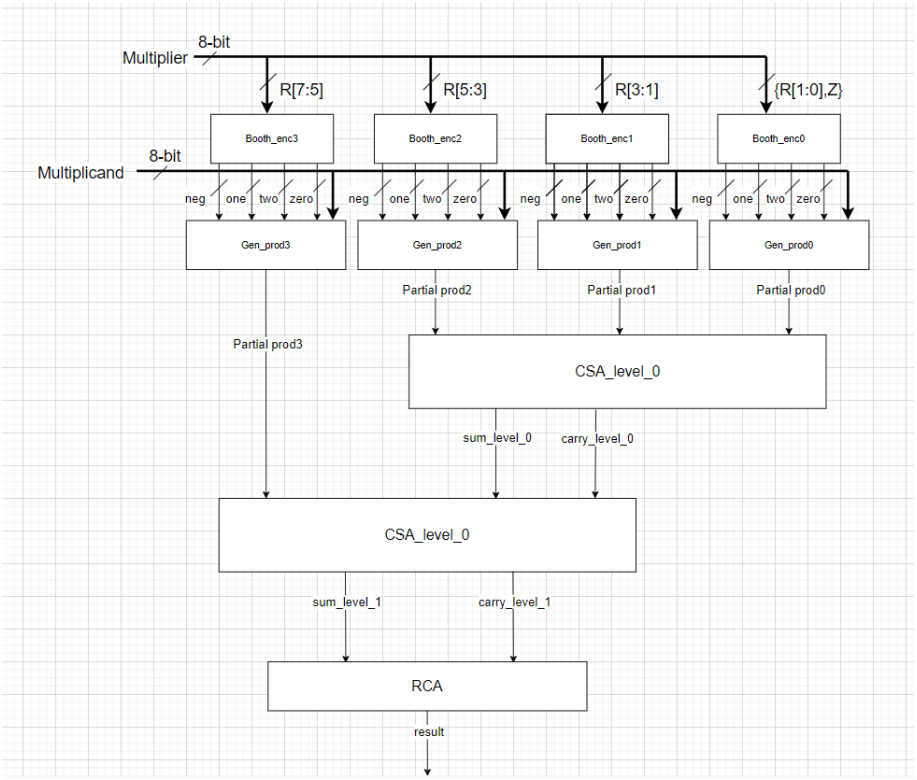

Figure 3

Simulation Results:

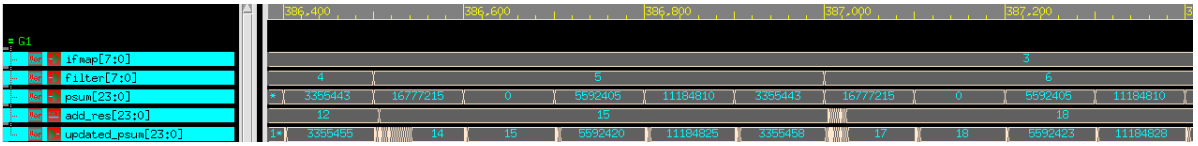

Figure 4

In Fig.4, one can observe that 3*5 = 15, which is shown in add_res, and then psum is added to the 15, which gives the updated_psum. For another example, 3*6 = 18, which is also correct.

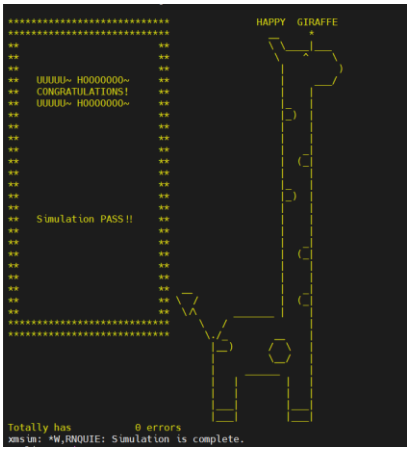
HAPPY GIRAFFE
UUUUU~ HOOOOOOO~
CONGRATULATIONS!
UUUUU~ HOOOOOOO~
Simulation PASS!!
Totally has 0 errors
xmsim: *W,RNQUIE: Simulation is complete.

Figure 5