Multiplication
Ivor Page[1]

[1]University of Texas at Dallas

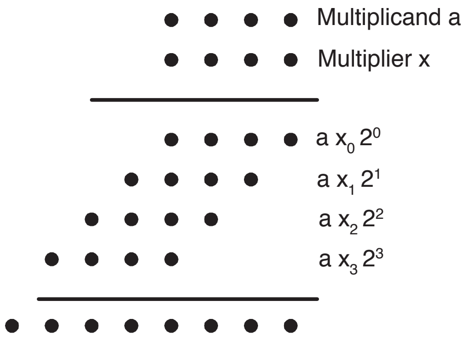
Multiplication

Shift/Add Multiplication Algorithms

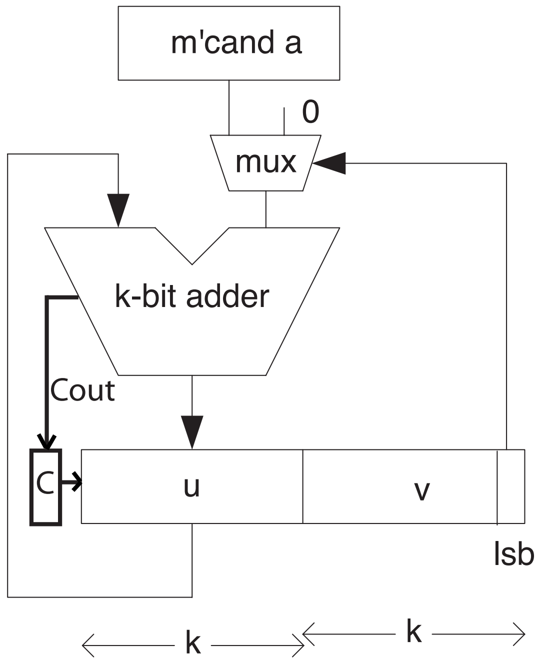
We will adopt the notation,

a	Multiplicand	$\langle a_{k-1}a_{k-2}\cdots a_1a_0\rangle$
b	Multiplier	$\langle x_{k-1}x_{k-2}\cdots x_1x_0\rangle$
p	Product	$\langle p_{2k-1}p_{2k-2}\cdots p_1p_0\rangle$

The following figure shows the arrangement of partial products produced by a row of AND gates, $p^i = x_i.\langle a_{k-1}\cdots a_1, a_0\rangle \times 2^i$.



The shift/add multiplier is a time-iterative system based on a k bit adder and a $2k$ bit result register that is shifted right one place on each iteration. Here is a schematic:



Initially the multiplier, x, is loaded into the k bit register V. The operand y is loaded into U, the upper half of that double register. At the end of the multiply operation, the double length result register contains $y + xa$.

Each iteration comprises an add cycle and a shift cycle. Either the multiplicand or zero is added to the upper half of the result register, depending on the lsb of the multiplier. The result is stored in the upper half of the double register, then the double register is shifted right one place. For this scheme to work, the register U would have to be $k+1$ bits in length. The carry flip-flop, C provides the extra bit.

In practice, the add and shift cycles are usually combined into one operation. If the lsb of the multiplier is 1, a combined add/shift take place. If it is zero, only a shift of the double length $U.V$ result register is needed.

A counter is required in the control logic of all time-iterative implementations to tell when all partial products have been dealt with.

The shifts cause the bits of the multiplier to be presented to the control input of the multiplexer in sequence and cause the places vacated by the multiplier to be taken up by the result as it grows to the right.

Example, $5 \times 7 + 0,\ k = 4\ bits$:

```
   C    u         v
   -|--------|-------
   0.0 0 0 0   0 1 1 1  m'pr lsb=1 so add:
 +  0 1 0 1
     ----------------
   0.0 1 0 1   0 1 1 1  shift:

   0.0 0 1 0   1 0 1 1  m'pr lsb=1 so add:
 +  0 1 0 1
     ----------------
   0.0 1 1 1   1 0 1 1  shift:

   0.0 0 1 1   1 1 0 1  m'pr lsb=1 so add:
 +  0 1 0 1
     ----------------
   0.1 0 0 0   1 1 0 1  shift:

   0.0 1 0 0   0 1 1 0  m'pr lsb=0 so just shift:

   0.0 0 1 0   0 0 1 1 = 35
```

9.1 Negative Operands

If we multiply $A \times B$, where the multiplicand A is negative and the multiplier B is positive, the result is $(2^k - A) \times B = B2^k - AB$. The required result is $2^{2k} - AB$.

We wish to simulate the effect of having a $2k$ bit multiplicand, a $2k$ bit U register, and a $2k$ bit adder by using the carry-out signal in some way. For positive multiplicands, as in the above example, we simply set the carry-out flag to zero initially and then shift in the carry-out flag on each shift operation. But, if the multiplicand were $2k$ bits long, that carry-out signal would be added to the k'th bit of the multiplicand, which would always be 1 for a negative multiplicand.

If the multiplicand were negative, both the k'th bit of the multiplicand and the k'th bit of the 2k bit U register would be one. If U is initially positive then, on the first add, a negative multiplicand is added to zero. The carry-out flag must be set to one. This is the case, even if there have been previous shift operations.

In the following example, the add and shift are separated into two operations so that the process can be understood more easily. The C flip-flop value is separated from the U register value by a period. On the right is the equivalent operation using $2k$ bit multiplicand, a $2k$ bit U register, and a 2k bit adder:

Example, $-5 \times 7 + 0 = 1011 \times 0111 + 0000$:

```
                                              Double length U register and multiplicand
  C    U         v                                     U             V
  -|--------|-------                          --------------------|-------
  0.0 0 0 0   0 1 1 1  m'pr lsb=1 so add:       0 0 0 0   0 0 0 0   0 1 1 1  add the 2k bit m'cand
 +  1 0 1 1                                   + 1 1 1 1   1 0 1 1
    -----------------                           -----------------
  1.1 0 1 1   0 1 1 1  shift:                   1 1 1 1   1 0 1 1   0 1 1 1  shift
    1 1 0 1   1 0 1 1  m'pr lsb=1 so add:       1 1 1 1   1 1 0 1   1 0 1 1  add
 +  1 0 1 1                                   + 1 1 1 1   1 0 1 1
    -----------------                           -----------------
  1.1 0 0 0   1 0 1 1  shift:                   1 1 1 1   1 0 0 0   1 0 1 1  shift
    1 1 0 0   0 1 0 1  m'pr lsb=1 so add:       1 1 1 1   1 1 0 0   0 1 0 1  add
 +  1 0 1 1                                   + 1 1 1 1   1 0 1 1
    -----------------                           -----------------
  1.0 1 1 1   0 1 0 1  shift:                   1 1 1 1   0 1 1 1   0 1 0 1  shift
    1 0 1 1   1 0 1 0  m'pr lsb=0 so shift:     1 1 1 1   1 0 1 1   1 0 1 0  shift

  1.1 1 0 1   1 1 0 1 = -35                     1 1 1 1   1 1 0 1   1 1 0 1  = -35
```

In the example above, where the multiplicand is negative and the multiplier is positive, a 1 is shifted into the U register on each right shift operation. But, consider the following example where two shifts precede the first add:

Example, $-5 \times 4 + 0 = 1011 \times 0100 + 0000$:

```
                                              Double length U register and multiplicand
  C    U         v                                     U             V
  -|--------|-------                          --------------------|-------
  0.0 0 0 0   0 1 0 0  m'pr lsb=0 so shift:     0 0 0 0   0 0 0 0   0 1 0 0  shift
  0.0 0 0 0   0 0 1 0  shift:                   0 0 0 0   0 0 0 0   0 0 1 0  shift
    0 0 0 0   0 0 0 1  m'pr lsb=1 so add:       0 0 0 0   0 0 0 0   0 0 0 1  add
 +  1 0 1 1                                   + 1 1 1 1   1 0 1 1
    -----------------                           -----------------
  1.1 0 1 1   0 0 0 1  shift:                   1 1 1 1   1 0 1 1   0 0 0 1  shift
  1.1 1 0 1   1 0 0 0  shift:                   1 1 1 1   1 1 0 1   1 0 0 0  shift
    1 1 1 0   1 1 0 0                           1 1 1 1   1 1 1 0   1 1 0 0

  1.1 1 1 0   1 1 0 0 = -20                     1 1 1 1   1 1 1 0   1 1 0 1  = -20
```

The carry flag was initially set to zero. On the first add, the carry flag became 1. This is a special case for negative multiplicands. Normally, the carry flag is the carry-out of the adder.

Notice also that the carry flag only changes on an add (or a subtract) operation.

Now consider a positive multiplicand and a negative multiplier: $(A) \times (2^k - B) = A2^k - AB$. If we sign-extend the multiplier, there will be $2k$ partial products and the time for a multiply will double. There are two other ways to solve this problem. The first is to subtract $A2^k$ from the result. This requires an additional add cycle. The second method is to treat the sign bit of the multiplier differently in the final cycle when that bit is the lsb of the V register. If that bit is 1, we subtract the multiplicand.

First, let's see how the incorrect result is formed and use the extra subtraction cycle to get the correct result.

Example, $5 \times (-7) + 0$:

```
 C.   U         V
 0.0 0 0 0   1 0 0 1  m'pr lsb=1 so add/shift:
+  0 1 0 1
   -----------------
 0.0 1 0 1   1 0 0 1  shift
 0.0 0 1 0   1 1 0 0  m'pr lsb=0 so just shift:
 0.0 0 0 1   0 1 1 0  m'pr lsb=0 so just shift:
 0.0 0 0 0   1 0 1 1  m'pr lsb=1 so add/shift:
+  0 1 0 1
   -----------------
 0.0 1 0 1   1 0 1 1  shift
 0.0 0 1 0   1 1 0 1     = 45. subtract the m'cand from U:
-  0 1 0 1
   -----------------
 1.1 1 0 1   1 1 0 1 = -35
```

Next we will avoid the need for a correction cycle by subtracting the multiplicand in the final cycle if the lsb of V (sign bit of the multiplier) is 1.

Example, $5 \times (-7) + 0$:

```
 C.   U         V
 0.0 0 0 0   1 0 0 1  m'pr lsb=1 so add/shift:
+  0 1 0 1
   -----------------
 0.0 1 0 1   1 0 0 1  shift
 0.0 0 1 0   1 1 0 0  m'pr lsb=0 so just shift:
   0 0 0 1   0 1 1 0  m'pr lsb=0 so just shift:
   0 0 0 0   1 0 1 1  m'pr lsb=1, subtract/shift:
-  0 1 0 1
   -----------------
 1.1 0 1 1   1 0 1 1  shift
   1 1 0 1   1 1 0 1 = -35
```

Finally, we consider an example where both multiplicand and multiplier are negative.

Example, $-5 \times (-7) + 0$:

```
  C.   U         V
  0.0 0 0 0   1 0 0 1  m'pr lsb=1 so add/shift:
+   1 0 1 1
    -----------------
  1.1 0 1 1   1 0 0 1  shift
  1.1 1 0 1   1 1 0 0  m'pr lsb=0 so just shift:
  1.1 1 1 0   1 1 1 0  m'pr lsb=0 so just shift:
  1.1 1 1 1   0 1 1 1  m'pr lsb=1, subtract/shift:
-   1 0 1 1
    -----------------
  0.0 1 0 0   0 1 1 1  shift
    0 0 1 0   0 0 1 1 = 35
```

Here is a summary of how the C register is set on each operation:

Sign of m'cand	Initial C	C after first add	C after subsequent adds	C after subtract
$m'cand < 0$	0	1	carry-out	0
$m'cand > 0$	0	0	carry-out	carry-out

The C register is shifted into the U.V register pair on each shift, including those after an add or a subtract cycle. It does not change during any shift operation.

9.2 Booth Recoding

In the final example above, there is a string of 2 zeros in the multiplier. The corresponding cycles of the multiplication are just shifts. In Booth radix-2 recoding, the cycles of the multiplication algorithm corresponding to strings of consecutive zeros or ones in the multiplier require only shifts. The simplest way to understand the technique is to consider recoding of the bits of the multiplier into a redundant number system with digit range [-1,1] (that is digit set $\{-1, 0, 1\}$). Here is an example:

```
  0  0  0  1  1  1  1  1  1  1  0  0  1  1  1  m'pr x in radix-2
  0  0  1  0  0  0  0  0  0 -1  0  1  0  0 -1  recoded m'per
```

A zero followed by a string of w ones is replaced by 1, $w - 1$ zeros, then -1. The complete algorithm for conversion is to follow.

```
  1  0  0  1  1  1  0  1  1  0  1  0  1  1  1  0  m'pr x in radix-2
 -1  0  1  0  0 -1  1  0 -1  1 -1  1  0  0 -1  0  recoded m'per
```

In the first example, the number of add cycles is reduced from 10 to 4 but, if each bit of the multiplier has an even chance of being a one or a zero, the average number of adds remains the same.

The modification to the hardware suggests that both the multiplicand and its negative value be available in registers. Alternatively, logic could be provided to generate the 1's complement of the multiplicand and, when the complement is used, the carry-in to the adder would be set to 1.

The multiplier register is extended by one bit to include x_{-1}. The operation in each cycle depends upon the two least significant bits of the multiplier register, x_0 and x_{-1}. Initially $x_{-1} = 0$.

Booth Radix-2 Recoding

x_i	x_{i-1}	Operation	Explanation
0	0	shift	within sequence of zeros
0	1	$+a$/shift	end of a sequence of 1's
1	0	$-a$/shift	beginning of a sequence of 1's
1	1	shift	within sequence of 1's

Consider some values in 2's complement notation and their Booth recoding:

```
2's complement   Booth recoding     Value
-------------    --------------     -----
1  0  0  0  0    -1  0  0  0  0      -16
1  1  0  0  0     0 -1  0  0  0       -8
1  1  1  0  0     0  0 -1  0  0       -4
1  1  1  1  1     0  0  0  0 -1       -1
0  0  0  0  0     0  0  0  0  0        0
0  0  0  0  1     0  0  0  1 -1        1
0  0  0  1  1     0  0  1  0 -1        3
0  0  1  1  1     0  1  0  0 -1        7
0  1  1  1  1     1  0  0  0 -1       15
0  1  1  1  0     1  0  0 -1  0       14
0  1  1  0  0     1  0 -1  0  0       12
```

The recoded version is not a complement system, but a redundant positional system with radix-2 and digit set $[-1, 1]$. This implies that the Booth multiplier correctly handles negative multiplier values.

An example follows in which the multiplier is negative. The result register's sign bit propagates to the right during shifts. Note the extra bit position to the right of the lsb of the multiplier, following the point in the example.

Example, $5 \times (-7) + 0$:

```
    0 0 0 0 1 0 0 1.0  m'pr lsbs=10 so subtract/shift:
-   0 1 0 1
    ---------------
    1 1 0 1 1 1 0 0.1  m'pr lsbs=01 so add/shift:
    ---------------
+   0 1 0 1
    ---------------
    0 0 0 1 0 1 1 0.0  m'pr lsbs=00 so just shift:
    ---------------
    0 0 0 0 1 0 1 1.0  m'pr lsbs=10, subtract/shift:
-   0 1 0 1
    ---------------
    1 1 0 1 1 1 0 1 = -35
```

Radix-2 Booth recoding will not, on average, reduce the number of additions. Radix-4 Booth recoding is, however, widely used to reduce the number of partial products to be added.

9.3 High Radix Modified Booth Recoding

Notice that, in the radix-2 Booth recoded string, the non-zero elements alternate between -1 and 1. We cannot have 11 or 101 or 1001, etc. within these strings of digits. The same applies to negative 1's. This simple observation leads to very valuable limits on the digit set in higher radix recoding systems. The radix-4 recoding system is obtained by combining pairs of neighboring digits in the radix-2 recoding:

1	0	0	1	1	1	0	1	1	0	1	0	1	1	1	0	radix-2
$\bar{1}$	0	1	0	0	$\bar{1}$	1	0	$\bar{1}$	1	$\bar{1}$	1	0	0	$\bar{1}$	0	Booth radix-2
	-2		2		-1		2		-1		-1		0		-2	Booth radix-4

The digit set is $[-2, +2]$. We don't need the values 3 or -3. The variations in the multiplicand needed in the multiplication process can be got from the multiplicand and its complement by shifts.

Similarly, in the radix-8 recoding, the digit set is $[-4, 4]$ and is got by combining three neighboring digits in the radix-2 recoding string:

1	0	0	1	1	1	0	1	1	0	1	0	1	1	1	0	radix-2
$\bar{1}$	0	1	0	0	$\bar{1}$	1	0	$\bar{1}$	1	$\bar{1}$	1	0	0	$\bar{1}$	0	Booth radix-2
-1		2			-1			-1			-2			-2		Booth radix-8

For radix-16 recoding the digit set is $[-8, 8]$:

1	0	0	1	1	1	0	1	1	0	1	0	1	1	1	0	radix-2
$\bar{1}$	0	1	0	0	$\bar{1}$	1	0	$\bar{1}$	1	$\bar{1}$	1	0	0	$\bar{1}$	0	Booth radix-2
		-6				-2				-5				-2		Booth radix-16

In each case, the redundancy is cleverly used to limit the digit set.

If we call the multiplicand a, we see that in radix-8 recoding, $3a$ and $-3a$ are needed. The remaining versions of the multiplicand are obtained from a and $-a$ by shifts. In radix-16 recoding, $\pm 5a$, $\pm 6a$ and $\pm 7a$ are also needed. We can obtain $\pm 6a$ from $\pm 3a$ by left shifts. The odd values are obtained from a by adding and shifting: $7a = 8a - a$, $5a = 4a + a$, $3a = 2a + a$ the complements of these are also needed.

We have two choices in designing the multiplier. One is to precompute all versions of the multiplicand before beginning the multiply process. The results would then be held in a table. The second option is to generate each version of the multiplicand as it is needed, using shifters and CSA adders. Each CSA adder would generate two vectors of bits, the sum vector and the carry vector. These could be added in the main adder of the multiplier. For speed purposes, the main adder would also be implemented using the CSA technique.

9.4 Generating the recoded string

The radix q recoding string is got by an *on the fly* algorithm that examines $1 + log_2\, q$ bits of the multiplier at each iteration.

Here is an example of the algorithm at work in a radix-4 system. An extra bit is added at the right hand end of the binary multiplier value (after the point) and initialized to zero. Three bits of the multiplier are examined at each step.

1	0	0	1	1	1	0	1	1	0	1	0	1	1	1	0 . 0	radix-2
	-2		2		-1		2		-1		-1		0		-2	Booth radix-4

Each radix-4 recoded digit is got by examining the three binary digits above it. When each recoded bit has been generated, and used in the multiplication operation, the result register is right shifted two places. Only $k/2$ partial products are generated.

The following table shows how the Booth radix-4 digits are generated from three consecutive bits in the binary bit pattern of the multiplier.

Booth Radix-4 Recoding

x_{i+1}	x_i	x_{i-1}	Operation	Explanation
0	0	0	shift	within sequence of zeros
0	0	1	$+a$/shift	end of a sequence of 1's
0	1	0	$+a$/shift	isolated 1
0	1	1	$+2a$/shift	end of a sequence of 1's
1	0	0	$-2a$/shift	beginning of a sequence of 1's
1	0	1	$-a$/shift	end of a sequence of 1's,beginning of another
1	1	0	$-a$/shift	beginning of sequence of 1's
1	1	1	shift	within sequence of 1's

Example, $22 \times (-6) + 0$, $k = 6$ bits:

```
   0 0 0 0 0 0 1 1 1 0 1 0.0  m'pr lsbs=100 so -2a/shift:
-  1 0 1 1 0 0
   ---------------
   1 1 0 1 0 1 0 0 1 1 1 0.1  m'pr lsbs=101 so subtract/shift:
   ---------------
-  0 1 0 1 1 0
   ---------------
   1 1 0 1 1 1 1 1 0 0 1 1.1  m'pr lsbs=00 so just shift:
   ---------------
   1 1 1 1 0 1 1 1 1 1 0 0.0  = -132
```

The adder must generate $k+2$ output bits, the left-most being the sign bit. The right-shifts of the result register are 2-place arithmetic shifts (the sign bit is propagated to the right).

Here is the conversion table for radix-8 recoding. Four consecutive bits of the multiplier are examined on each step:

Booth Radix-4 Recoding				
x_{i+2}	x_{i+1}	x_i	x_{i-1}	
0	0	0	0	shift
0	0	0	1	$+a$/shift
0	0	1	0	$+a$/shift
0	0	1	1	$+2a$/shift
0	1	0	0	$+2a$/shift
0	1	0	1	$+3a$/shift
0	1	1	0	$+3a$/shift
0	1	1	1	$+4a$/shift
1	0	0	0	$-4a$/shift
1	0	0	1	$-3a$/shift
1	0	1	0	$-3a$/shift
1	0	1	1	$-2a$/shift
1	1	0	0	$-2a$/shift
1	1	0	1	$-a$/shift
1	1	1	0	$-a$/shift
1	1	1	1	shift

It is simple to generate these tables once the conversion from Booth radix-2 to the higher radix scheme is understood. Consider the above radix-4 table entry for the binary input pattern $< 1\ 0\ 1\ 0 >$. The equivalent Booth radix-2 pattern (for the leftmost 3 bits) is $< -1\ 1\ -1 >$. Its value as a single radix-8 digit is -3. Therefore, $-3a$ appears in the table.

A similar scheme is used to develop the radix-16 recoding. The radix-16 multiplier generates $k/4$ partial products (or requires $k/4$ add/shift cycles). Five bits of the multiplier are examined on each iteration and a shift of four places takes place in each cycle.

9.5 Using a Carry Save Adder

In the above implementations, the nature of the main adder was not specified. If a non-redundant carry-propagate adder is used, the best time for an add cycle is $O(log\ k)$. Since repeated additions are required, a carry-save adder is appropriate. In this design, the upper half of the result register comprises two parts, k bits for the incomplete sum, and k bits for the carries from the previous add cycle.

Figure 1 illustrates the design:

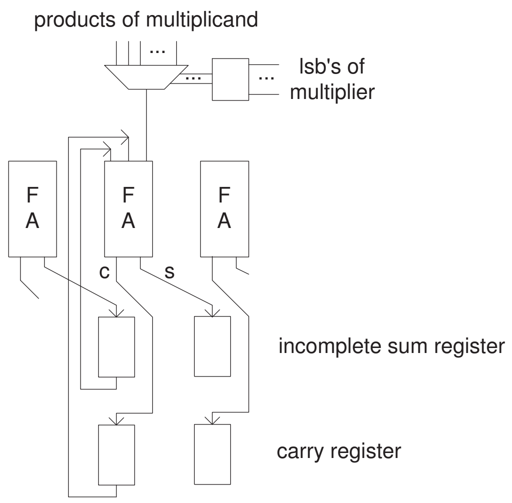


Figure 1: Single CSA Adder in Booth Multiplier

In the diagram, the products of the multiplicand are a, $-a, \cdots$. The outputs of the full-adders are skewed to the right to represent the combination of an add and a single bit shift. If Booth radix-4 or greater recoding is used, the skew would be over more bits.

In each iteration a CSA addition takes constant time. After the final iteration, a carry-propagate addition is required to combine the incomplete sums with the carries. This can be done in $T_{RC} = O(k)$ time using a ripple-carry adder, or $T_{CLA} = O(log\ k)$ time using a CLA adder. The overall time is then $c_1k/b + T_{RC}$ or $c_1k/b + T_{CLA}$ where c_1 is the delay through the CSA and $b = 1, 2, 3, 4$ corresponds to the advantage of using Booth recoding. Since the first term is linear in k, it seems sensible to use a ripple-carry adder for the final stage. Minimizing the absolute value of the total multiplication time is, however, usually the goal. Any reduction in the total time may be worth the extra hardware.

Consider, for example, a 32×32 bit multiplier. We can reasonably guess $c_1 = 5D$. If we use a radix-16 recoding system, the first term in the equation for the overall time is $5D \times 32/4 = 40D$. If a ripple-carry adder is used in the final stage, the second term is approximately $2 \times 32D = 64D$. If a CLA adder is used with CLA units of size 4, the second term becomes about $15D$. The total times are $104D$ and $55D$, a difference of almost a factor of 2.

9.6 Direct Generation of Partial Products Without Recoding

Consider a design in which three digits of the multiplier are examined at a time and a shift of three places takes place on each iteration. We call this a radix-8 multiplier. Say we examine x_{i+2}, x_{i+1}, x_i, beginning with $i = 0$. We would add $ax_{i+2}2^{i+2} + ax_{i+1}2^{i+1} + ax_i2^i$ to the result register. Since scaling of the partial products takes place automatically, we would actually add one of $\{a, 2a, \cdots, 7a\}$ to the upper half of the result register. A shift network would generate $2a$ and $4a$ and a CSA tree with five input operands would generate the next partial result. The five operands would be $4ax_{i+2}$, $2ax_{x+1}$, ax_1, and the incomplete sum and carry outputs from the previous iteration. Note that the digit set for this system is $[0, 7]$. Compare this to radix-8 Booth recoding which has a digit set $[-4, 4]$.

For a radix-16 system of this kind, the digit set is $[0, 15]$, three CSAs are needed and the depth of the CSA tree is three (to generate partial products such as $11a$, for example).

The use of a CSA tree significantly adds to the area of the multiplier in comparison with the equivalent Booth Radix-8 scheme. The delay through the CSA tree is $O(log\ b)$ for radix b. This must be compared with the delay through the recoding unit in the Booth design. That unit examines 4 bits of the multiplier and generates signals for the multiplexer control inputs. The CSA tree design uses three bits of the multiplier directly as the multiplexer inputs.

Radix-16 CSA tree-multipliers are also possible, providing an almost $4\times$ speedup over the single CSA design.

The optimal area$\times$time product for any k bit multiplier is proportional to $\sqrt{k}$ and the optimal area$\times$time^2 product is proportional to k^2. The overall time complexity of a radix b CSA tree multiplier is $O((k/b)log\ b + log\ k)$ if a CLA adder is used for the final iteration. The area of the array of AND gates to produce the b partial products is $O(kb)$.

So, for a radix b CSA tree multiplier, the two products are:

$$AT = O(k^2 log\ b + bk\ log\ k)$$

$$AT^2 = O((k^3/b) log^2\ b)$$

These designs are sub-optimal for any radix b.

Pipelining of the CSA tree can be used to enable a further speedup. For example, while the first level of the tree is adding values for iteration I, the second level is adding values for iteration $I - 1$ and so on. This improvement reduces the time complexity to $O((k/b) + log\ k)$

Radix b Booth multipliers follow the same asymptotic results for area since we have not included the area of the CSA units in our estimates. The time complexity of the Booth radix b scheme is $O((k/b) + log\ k)$, assuming that the recoding logic adds insignificantly to the delay.

The pipelining idea is taken to the limit in the next module where we consider array multipliers.