Cairo University
Electronics and Communications Department

Computer Arithmetic: Go forth and multiply

Hossam A. H. Fahmy

What is $X \times Y$?

- For integers

$$\text{Product} = \underbrace{X + X + \cdots + X}_{Y \text{ times}}$$

- All the computer numbers are represented as integers.
- Depending on the time and resources allowed for this operation, several implementations are possible.

Loop on Y

```
while(Y>0){product = product + X; Y=Y-1; }
```

- This is the simplest implementation.
- Feasible in both hardware and software.
- If Y has n bits this algorithm takes up to an $\mathcal{O}(2^n)$ steps.

$\Rightarrow$ *Slow and with variable latency*

Loop on the bits of Y

The *add and shift* method examines the bits of Y.

1. If the bit $Y[0] = 1$, add X.
2. Shift both the product and Y to the *right* one bit.
3. Repeat for the n bits of Y.

Fixed latency of $\mathcal{O}(n)$.

Booth recoding

Booth proposed an algorithm based on the fact that

a string of ones	$\cdots 011 \cdots 110 \cdots$
is equal to	$\cdots 100 \cdots 0\bar{1}0 \cdots$.

Instead of adding repeatedly, add only twice. The recoding is simple:

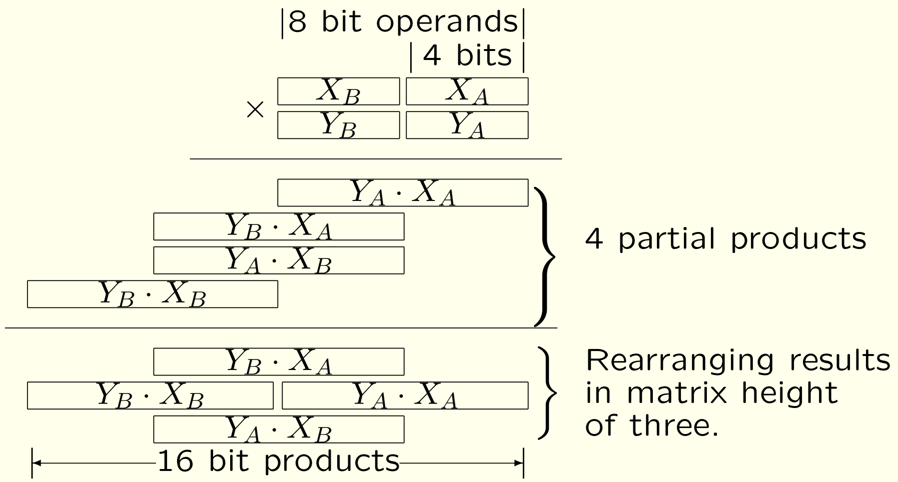
1. On a transition from 0 to 1, put $\bar{1}$ at the location of the 1.
2. On a transition from 1 to 0, put 1 instead of the 0.
3. Put zeros at all the remaining locations. (i.e. skip groups of zeros and groups of ones.)

- It has a variable latency but, on average, the use of the Booth algorithm reduces the time delay.
- The worst case is $(01010101\cdots = 1\bar{1}1\bar{1}1\bar{1}1\bar{1}) \Rightarrow \mathcal{O}(n)$ delay.

Types of hardware multipliers

- Sequential: using any of the methods mentioned so far.
- Parallel:
  1. Simultaneous generation of all the partial products.
  2. Parallel reduction of the partial products to two bit vectors.
  3. A final Carry Propagate Adder (CPA).
- Iterative: not fully parallel and not fully sequential.

PP generation using ROMs



Implementation of 8×8 *unsigned* multiplication using four 256×8 ROMs, where each ROM performs 4×4 multiplication.

Booth revisited

- For a parallel implementation, we want a fixed number of partial products. A smaller number is considered better.
- The original Booth algorithm recodes the number in a redundant format $(\bar{1}, 0, 1)$ by scanning overlapped groups of 2 bits. The worst case is n PPs.
- If we scan 2 *new* bits in each group and use the set $\{\bar{2}, \bar{1}, 0, 1, 2\}$ we get almost $\frac{n}{2}$ PPs.

Original bits			Booth recoding			Value
y_{j+1}	y_j	y_{j-1}	y_{j+1}	y_j	y_{j-1}	
0	0	0	0	0	0	+0
0	0	1	0	1	0	+1
0	1	0	1	$\bar{1}$	0	+1
0	1	1	1	0	0	+2
1	0	0	$\bar{1}$	0	0	−2
1	0	1	$\bar{1}$	1	0	−1
1	1	0	0	$\bar{1}$	0	−1
1	1	1	0	0	0	−0

$y_{j-1} = 1$ indicates that it is a string of ones.

Another way of looking at it

Let us think that we are converting from a non-redundant representation to a redundant one.

For each group of two (new) bits we generate a value and a possible carry into the next higher group.

Original bits				
y_{j+1}	y_j	c_{in}	c_{out}	value
0	0	0	0	$+0$
0	0	1	0	$+1$
0	1	0	0	$+1$
0	1	1	0	$+2$
1	0	0	1	-2
1	0	1	1	-1
1	1	0	1	-1
1	1	1	1	-0

We choose c_{out} and the value in this particular manner to make $c_{out} = y_{j+1}$ and hence reduce the logic gates needed to generate it.

The values chosen are also easy to generate by a simple shifting of X or $-X$.

Modified Booth 2 algorithm

In the modified version, we produce $(\frac{n}{2} + 1)$ PPs

Algorithm for *unsigned* numbers:

1. Pad the *LSB* with one 0.
2. Pad the *MSB* with two 0 if n is even and one 0 if n is odd.
3. Divide the multiplier into overlapping groups of 3-bits.
4. Determine the scale factor from the recoding table.
5. Select the multiplicand multiples.
6. Sum the partial products.

The price of Booth 2

- In a direct multiplier, $PP_j = Xy_j = \sum_{i=0}^{i=n-1} x_i y_j 2^i$. An array of *AND* gates is enough.
- In a Booth 2 multiplier, some time is used for recoding and for selecting the correct multiplicand multiple.

We have less PPs but we spend some time, area, and power:

1. to recode the bit string into the redundant form,
2. to select the correct multiple of the multiplicand using a multiplexer whose inputs are 0, X, $2X$, and their negatives, and
3. to sign extend the PPs since some of them might be negative although we are sure that for unsigned numbers the final product is positive.

Signed multiplication

If only the multiplicand is signed and represented in 2's complement the algorithm works fine. However, for a signed 2's complement multiplier we need yet another modification:

1. Pad the *LSB* with one 0.
2. *If n is even do not pad the MSB ($\frac{n}{2}$ PPs) and if n is odd pad the MSB with one 0 ($\frac{n+1}{2}$ PPs).*
3. Divide the multiplier into overlapping groups of 3-bits.
4. Determine the scale factor from the recoding table.
5. Select the multiplicand multiples.
6. Sum the partial products.

Negation of X

- To get $-X$ or $-2X$ invert each bit and add one at the *LSB*.
- If we use Booth 2 as just described, we need this inversion only if $y_{j+1} = 1$ (note that $-0 = 111\cdots111 + 1$).
- We are sure that the *LSB* location is empty in the lower PPs.

$$
\begin{array}{cccccccccccccccc}
a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
b_9 & b_9 & b_9 & b_9 & b_9 & b_8 & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & & y_1 \\
c_9 & c_9 & c_9 & c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & & y_3 & & \\
d_9 & d_8 & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 & & y_5 & & & & \\
e_7 & e_6 & e_5 & e_4 & e_3 & e_2 & e_1 & e_0 & & y_7 & & & & & & \\
\hline
p_{15} & p_{14} & p_{13} & p_{12} & p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Sign extension

Since $(sss\cdots sss)\mathbf{mod_{2^n}} = (111\cdots111{+}\bar{s})\mathbf{mod_{2^n}}$ then we can reduce the needed summation to

$$
\begin{array}{cccccccccccccccc}
 & & & & & \bar{s}_0 & s_0 & s_0 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
 & & & & 1 & \bar{s}_1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & y_1 \\
 & & 1 & \bar{s}_2 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & y_3 & & \\
1 & \bar{s}_3 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & y_5 & & & & \\
 & & & & & & & \vdots & & & & & & & &
\end{array}
$$

Booth 3

Since the use of two new bits reduced the number of PPs, we might use three bits and reduce it further.

y_{j+2}	y_{j+1}	y_j	y_{j-1}	value	y_{j+2}	y_{j+1}	y_j	y_{j-1}	value
0	0	0	0	+0	1	0	0	0	−4
0	0	0	1	+1	1	0	0	1	−3
0	0	1	0	+1	1	0	1	0	−3
0	0	1	1	+2	1	0	1	1	−2
0	1	0	0	+2	1	1	0	0	−2
0	1	0	1	+3	1	1	0	1	−1
0	1	1	0	+3	1	1	1	0	−1
0	1	1	1	+4	1	1	1	1	−0

We get almost $\frac{n}{3}$ PPs but:

1. the multiple $3X$ is a *hard multiple*,
2. the recoding logic is more complex, and
3. there is a need for a bigger multiplexer.

PP generation conclusions

- The PPs are independent and it is possible to generate them all in parallel.
- Reducing the number of PPs decreases the cost of their summation but increases that of their generation.