Cairo University
Electronics and Communications Department

Computer Arithmetic:
(Re)Learning the multiplication

Hossam A. H. Fahmy

The fused multiply add

Since the multiplication involves repeated additions, can $ab + c$ be performed in one step? *Why?*

- In the calculation of scalar products, matrix multiplications, or polynomial evaluation we often iterate on a an instruction such as $(sum = sum + a_i b_i)$.
- Making this instruction a single operation that is both *faster* and more *accurate* is beneficial.
- If there is no hardware support for the division and square root, then the presence of a FMA instruction speeds the software implementations of those two operations.
- We can also get the "lower" part of a multiplication using the FMA: $H = ab + 0.0$, $L = ab - H$.

Is FMA faster?

It is faster than two separate operations.

- The additional input is just another bit vector that can be easily summed at the end of the reduction tree.
- If the number of PPs does not completely fill the tree then summing that additional vector "should not" increase the time delay.

Is FMA good for everything?

The FMA raises some issues.

- Does the instruction format support having three inputs and a separate destination? If not, then an instruction such as $c = c + ab$ might be appropriate for most applications.
- The architecture must supply the FMA unit with three inputs $\Rightarrow$ increased wiring.
- Do the increased wiring and control lines to reconfigure the unit slow down the normal addition $(a \times 1 + c)$ or multiplication $(a \times b + 0)$?

Is FMA more accurate?

Doing two separate operations yields two roundings, for example:

1. $RNE(a \times b)$ followed by
2. $RNE(RNE(a \times b) + c)$.

On the other hand, a single FMA yields $RNE(a \times b + c)$. The two results are not always equal.

The result of FMA is mathematically better. However, the IEEE standard of 1985 requires the separate roundings. The current revision of the standard includes the FMA as a single operation.

Practical issues and the datapath width

- In the normal binary FP adder, we shift the smaller number to the right. *why?*
- In the FMA, the result of the multiplication might be the smaller. We do not want to wait till that result is ready to shift it.

⇒ Allow a much wider datapath where the addend operand c may be shifted to the left with respect to the product if c is larger.

In effect, we get a datapath that is $3n$ bits wide for n bits operands.

On top of that, subnormal numbers represent special cases. (See the paper describing the zSeries floating point unit.)

Back to the CPA of normal multipliers

Looking back at

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

we see that the arrival times of the bits to the final CPA are not equal.

The final CPA

Rather than a straight adder of size $2n$, there is an opportunity to reach a more area-time effective implementation:

- If the lower part is needed (as in a FMA) use a ripple carry adder.
- If the lower part is not needed just generate the bits needed for correct rounding.
- Use a carry select for the most significant bits.

The price of Booth recoding

- In a direct non-Booth 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, less PPs exist but we spend some time, area, and power:
  1. to recode the bit string into the redundant form, and
  2. to select the correct multiple of the multiplicand using a multiplexer whose inputs are 0, X, $2X$, and their negatives.
- In a Booth 3 multiplier, we get a smaller number of PPs with a slightly more complicated recoding and selection. However, we have hard multiples.

Use redundancy once more

It is possible to eliminate the hard multiple by using redundancy: instead of calculating $3X$ introduce $2X$ and X into the PP array.

- The hardware does not know a priori which PP will be $3X$. Hence, for each PP the multiplier must have two bit vectors.
- The number of rows becomes $\approx \frac{n}{3} \times 2$ which is worse than the Booth 2.

Partial redundancy is better

While the Booth recoder and the selection is going, a reduction of the $3X$ takes place. (Bewick 1994)

$$
\begin{array}{cccccccccl}
\cdots & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \leftarrow 2X \\
\cdots & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \leftarrow X \\
 & & & & & \Downarrow & & & & \\
\cdots & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \\
\cdots & & \bullet & & & \bullet & & & &
\end{array}
$$

We get a full bit vector and a sparse bit vector for each hard multiple. *How is the sparse vector better?*

How many bits is the digit?

- The sparse PPs should not align otherwise the array height will increase.
  - ⇒ For a redundant Booth 3, do not use a digit size that is a multiple of 3.
  - ⇒ The alignment will occur at the least common multiple of the digit size and the order of the Booth algorithm used.
- The longer the digit, the longer it will take to generate it. That time must balance the time of the recoding and selection otherwise, it will slow the multiplier.

What about $-3X$?

- If we complement the full and sparse vectors we get instead of the sparse vector a vector full of ones with a few zeros.
- A simple solution is to bias all the multiples to make them positive, i.e. use $k-3X$, $k-2X$, $k-X$, k, $k+X$, $k+2X$, and $k+3X$ where k is the bias.
- Then compensate for all the biases by a single constant equal to $-k$ multiplied by the number of the PPs and add that compensation constant as one additional row in the array.
- The calculation of this compensation occurs at *design* time not at *run* time.

Remember the wires

- Arrays use less wires than trees.
- If the technology and logic family restrict the number of wires per bit pitch then a higher order array is probably the best choice, otherwise a tree is faster.
- With technologies below $100nm$, the wires dominate the delay not the gates.
- According to Al-Twaijry (1997):
  - Redundant Booth has longer wires and is affected by that.
  - At $100nm$, wires represent 70% of the multiplier delay.
  - With a higher Booth, the problems of wires are less.
  - A procedure may be developed to connect the outputs of the previous compressors to the inputs of the following ones to balance the delays.

Conclusions

- Go ahead and optimize the FMA!
- Booth 2 is fast while Booth 3 uses less area.
- Redundant Booth is a good idea but it does not achieve a very large advantage to the point of adapting it in the industry. (Real companies run on profits!)
- Procedures to balance the delays (in general, not necessarily for multipliers) are incorporated in more CAD tools now.