EE 486 lecture 10: Multiply iteration 
and an introduction to Divide

M. J. Flynn

Patent searching

- In addition to your homework assignment 
also go to http://www.uspto.gov/ptdf/index.html 
and download the first page (image not text) 
of three arithmetic oriented patents 1) from 
patent applications 2) from issued patents 
and 3) one of whose inventors is “Rumynin”

Multipliers

<table>
<thead>
<tr>
<th>Type</th>
<th>D= Delay (CSAs)</th>
<th>W</th>
<th>D,W for 27b case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wallace</td>
<td>2log(\frac{1}{2}) n</td>
<td>depends</td>
<td>7</td>
</tr>
<tr>
<td>Binary tree</td>
<td>2[log,n-1]</td>
<td>2[log,n]</td>
<td>8, 10</td>
</tr>
<tr>
<td>ZM (type1)</td>
<td>2 sqrt(n)</td>
<td>5</td>
<td>9, 5</td>
</tr>
<tr>
<td>OS (type1)</td>
<td>sqrt(2n-6)</td>
<td>6</td>
<td>8, 6</td>
</tr>
<tr>
<td>Higher Order array</td>
<td>2 sqrt(n)</td>
<td>5</td>
<td>9, 5</td>
</tr>
</tbody>
</table>

Trees, arrays and iteration

- Arrays and trees need not be fully built out, 
but smaller structures can be iterated on to 
produce the product. While requiring 
multiple passes through the hardware, the 
simpler pp generation and reduction can 
make the processes faster than expected.
- The trick is (as with clocking) low iteration 
overhead

The Earle latch

- Adds latch to existing 
logic with little 
overhead, no delay.
- Pipeline rate is 4 gate 
delays (2 to transit and 
2 to hold).
- Becomes the basis for 
iterative multipliers.
Earle latch

- Cycle time of 4 gate delays (2 for clock and transit and 2 for ~clock).
- No additional transit delay
- Useful in pipelining CSA arrays and trees
- BUT, iterate on what?

Iterate on the CPA

- Brings the reduced product back into the tree.
- The tree now has \( n/I+1 \) inputs where \( I \) is the number of iterations.
- The CPA output is shifted by \( k \) \( n/I \) bits, \( k \) the multiplier encoder

Iterate on the tree

Inputs now \( (n/I) +2 \)
But no longer need the CPA as part of the iteration
Need to shift the CPA output to align the pp’s

Iterate on a “compressor”

- Add a [4:2] at the bottom of the tree, then iterate on the two outputs.
- Need to balance the iteration delays. Iteration is now 4 gate delays. The tree and the pp generator must support this rate.

Iteration

- Iteration reduces the cost of pp generation by 1 and of pp reduction somewhat less than that.
- If the iteration overhead can be kept low the result may compare favorably to a fully built tree.

Example, 64x64 using 8x8 generators and (5,5,4) counters

- Consider a fully built tree. The height, \( n \), is 15. There are 3 (5,5,4) counter delays in the tree.
- The pp generation uses 64 pp generators and 152 counters and a 128b CPA
Same multiplier with iteration

- Suppose we iterate on a single level of (5,5,4) counters. We reduce 5 pp’s in iteration 1 and 3 pp’s in each iteration thereafter. So 1+4 iterations covers the 15 pp’s. Now we need about 30 pp generators and only 32 or so counters. The CPA spans only about 74b.
- The counter delay is 5, ignoring the iteration overhead.

Two 64b multipliers compared

<table>
<thead>
<tr>
<th>Type</th>
<th>pp’s needed to generated</th>
<th>Depth in (5,5,4)’s</th>
<th>CPA size required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>15</td>
<td>3</td>
<td>128b</td>
</tr>
<tr>
<td>Iterative (one level)</td>
<td>3</td>
<td>5 (plus clocking overhead)</td>
<td>74b</td>
</tr>
</tbody>
</table>

Multipliers

- Multipliers (especially n > 50) are big and expensive, with potentially serious wire congestion and length problems.
- Yes, it’s possible to make a really awful multiplier that’s slower than a better design with half the hardware.
- Indeed, there’s room for yet one more multiplier patent! Good luck.

Divide

- Divisor/dividend = quotient + rem./dividend
- Basic approaches:
  - Subtractive (restoring, non restoring and SRT)
  - Multiplicative (Newton-Raphson, Binomial expansion in Taylor series)
  - Table look up (bipartite tables)

How important is divide?

- How much hardware should be devoted to divide and square root?
- Can compilers schedule long latency ops?
- What’s the role of multiple issue?
- Can the multiplier be shared with MPY and DIV?
- Can the divide be shared with DIV and SQRT?
Compiler effects

- Compiler optimization has 2 effects.
  - It decreases the number of LD/ST instructions, increasing the frequency of DIV.
  - Makes DIV relatively more important, but it also can manage dependencies better.

<table>
<thead>
<tr>
<th>Latency (~)</th>
<th>Excess CPI</th>
<th>Area (rbe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1b</td>
<td>&gt;40</td>
<td>0.5</td>
</tr>
<tr>
<td>2b</td>
<td>20–40</td>
<td>1.3</td>
</tr>
<tr>
<td>4b, simple NR</td>
<td>10–20</td>
<td>0.04–1</td>
</tr>
<tr>
<td>8b</td>
<td>4–10</td>
<td>0.01–0.07</td>
</tr>
<tr>
<td>Very High radix</td>
<td>&lt;4</td>
<td>&lt;0.01</td>
</tr>
</tbody>
</table>

Multiple issue effects

- Multiple issue machines need access to results earlier, even with optimized code.
- Earlier usage means more CPI delay due to DIV latency.
- Note the AT =k behavior for the DIV implementations.

Area-time tradeoffs

- The higher the issue rate the higher the CPI delay due to DIV. The relative effects are even more significant.
- MI processors need good divide support.

Sharing a multiplier with a divider

- If a multiplier is shared with the divide unit the conflict cost is low, especially for fast algorithms (.02–.03 excess CPI).
- The problem is more of implementation...a multi-function unit has more overhead.

Subtractive divide

- Restoring
- Non restoring
- Shift over 0’s
- Brute force (multiple subtrators)
Restoring and non restoring

- To find \( q_0 \ldots q_n \): \( R(n) = \) Dividend; trial \( q_i = 1 \); \( d \) is the divisor \( R(0) = \) remainder
- \( R(i+1) = R(i) - q_i \times 2^i \); if result is negative then set \( q_i = 0 \) and restore \( \ldots R(I) + 2^i \); if positive then set \( q_i = 1 \) and proceed.
- Non restore… if negative then set \( q_i= 0 \) and proceed with \( R(i+1) = R(i) + q_i \times 2^i \);
- Note that \( -2^i + d = -d \)

Speeding up division

- Skip over 0’s and skip over 1’s; similar to multiply
- Higher radix: radix 4 (2b/iteration) or radix 8 (3b/iteration). Widely used with redundant digit set.
- Simplest approach is to use multiple subtractors… also called brute force. Need 3 for 2b and 7 for 3b.

Redundancy in division

- Multiple subtractors can be replaced with redundant digit sets. This leads to SRT division (lecture 12)
- Usually limited to 2b / iteration.