COMPUTER ARITHMETIC
2ND PART
MULTIPLICATIONS

Real-time Signal Processing
High-speed multipliers are one of the key components in RISCs, DSPs, graphic accelerators and so on.

Multipliers must consider multiplication from 8 to 64 bits

Example: Double precision IEEE floating point standard consider 54 bits for mantissa. Therefore, the multiplier should be 54-bit wide.
Speeding Up Multiplication

- Multiplication involves **2** basic operations:
  - Generation of partial products +
  - Their accumulation

- **2 ways to speed up:**
  - Reducing number of partial products
  - and/or accelerating accumulation

- **3 types of high-speed multipliers:**
  - Sequential Multiplier
  - Parallel Multiplier
  - Array Multiplier
Speeding Up Multiplication

- **Sequential multiplier** - generates partial products sequentially and adds each newly generated product to previously accumulated partial product.

- **Parallel multiplier** - generates partial products in parallel, accumulates using a fast multi-operand adder.

- **Array multiplier** - array of identical cells generating new partial products; accumulating them simultaneously.
  - No separate circuits for generation and accumulation.
  - Reduced execution time but increased hardware complexity.
Sequential Multiplier

Reducing the number of partial products
Reducing Number of Partial Products

- Examining 2 or more bits of multiplier at a time
- Requires generating A (multiplicand), 2A, 3A
- Reduces number of partial products to n/2 - each step more complex
- Several algorithm which do not increase complexity, one is Booth's algorithm
- Fewer partial products generated for groups of consecutive 0’s and 1’s
Booth’s Algorithm - Rules

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$x_{i-1}$</th>
<th>Operation</th>
<th>Comments</th>
<th>$y_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>shift only</td>
<td>string of zeros</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>shift only</td>
<td>string of ones</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>subtract and shift</td>
<td>beginning of a string of ones</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>add and shift</td>
<td>end of a string of ones</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Recoding** multiplier $x_{n-1} x_{n-2} ... x_1 x_0$ in SD code
- **Recoded** multiplier $y_{n-1} y_{n-2} ... y_1 y_0$
- $x_i, x_{i-1}$ of multiplier examined to generate $y_i$
- Previous bit - $x_{i-1}$ - only reference bit
- $i=0$ - reference bit $x_{-1}=0$
- Simple recoding - $y_i = x_{i-1} - x_i$
- No special order - bits can be recoded in parallel
Example – Multiplier SD Recorded

Example: Multiplier 0011110011(0) recoded as

Taking 2 bits each time from the LSB

0 0 1 1 1 1 0 0 1 1 1 (0)

0 – 1 = 1

0 0 1 1 1 1 1 0 0 1 1

1 – 1 = 0

0 0 1 1 1 1 1 0 0 1 1 (0)
Example – Multiplier SD Recorded

\[
\begin{array}{cccccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 (0) \\
\hline
0 - 1 &=& 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 (0) \\
\hline
1 - 1 &=& 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
\end{array}
\]
### Example – Multiplier SD Recorded

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(0)</td>
</tr>
<tr>
<td>□</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0 = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(0)</td>
</tr>
<tr>
<td>□</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1 = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example – Multiplier SD Recorded

- Following the same sequence, we arrive to:
  \[
  \begin{array}{cccccccc}
  0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
  \end{array}
  \]

- From the original number that was
  \[
  \begin{array}{cccccccc}
  0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \ (0)
  \end{array}
  \]

- Therefore, we reduce the number of one’s from 6 to 4. This will provide a speed up in the multiplication
### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>(-5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>×</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(X)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(Y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add (-A)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Add (A)</td>
<td>+</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Shift</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Add (-A)</td>
<td>+</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Shift</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Sign Bit

- Two's complement - sign bit \( x_{n-1} \) must be used
- Deciding on add or subtract operation - no shift required - only prepares for next step
- Verify only for negative values of \( X \) (\( x_{n-1} = 1 \))
- 2 cases
  \[
  A \cdot X = A \cdot \tilde{X} - A \cdot x_{n-1} \cdot 2^{n-1}
  \]
  where \( \tilde{X} = \sum_{j=0}^{n-2} x_j 2^j \)
- Case 1 - \( A \) subtracted - necessary correction
- Case 2 - without sign bit - scan over a string of 1's and perform an addition for position \( n-1 \)
  - When \( x_{n-1} = 1 \) considered - required addition not done
  - Equivalent to subtracting \( A \cdot 2^{n-1} \) - correction term

<table>
<thead>
<tr>
<th></th>
<th>( x_{n-1} )</th>
<th>( x_{n-2} )</th>
<th>( y_{n-1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Booth’s Algorithm - Properties

- Multiplication starts from least significant bit
- If started from most significant bit - longer adder/subtractor to allow for carry propagation
- No need to generate recoded SD multiplier (requiring 2 bits per digit)
  - Bits of original multiplier scanned - control signals for adder/subtractor generated
- Booth's algorithm can handle two's complement multipliers
  - If unsigned numbers multiplied - 0 added to left of multiplier ($x_n=0$) to ensure correctness
Drawbacks to Booth's Algorithm

- Variable number of add/subtract operations and of shift operations between two consecutive add/subtract operations
  - Inconvenient when designing a synchronous multiplier
- Algorithm inefficient with isolated 1's
- Example:
  - 001010101(0) recoded as 011111111, requiring 8 instead of 4 operations
- Situation can be improved by examining 3 bits of X at a time rather than 2
Radix-4 Modified Booth Algorithm

- Bits $x_i$ and $x_{i-1}$ recoded into $y_i$ and $y_{i-1}$ - $x_{i-2}$ serves as reference bit
- Separately - $x_{i-2}$ and $x_{i-3}$ recoded into $y_{i-2}$ and $y_{i-3}$ - $x_{i-4}$ serves as reference bit
- Groups of 3 bits each overlap - rightmost being $x_1 x_0 (x_{-1})$, next $x_3 x_2 (x_1)$, and so on

\[ \begin{align*}
\bullet & \quad \bullet & \quad \bullet & \quad x_7 & \quad x_6 & \quad x_5 & \quad x_4 & \quad x_3 & \quad x_2 & \quad x_1 & \quad x_0 & \quad (x_{-1}) \\
& & & Y_7 Y_6 & Y_5 Y_4 & Y_3 Y_2 & Y_1 Y_0
\end{align*} \]
Radix-4 Algorithm - Rules

- \( i=1,3,5,... \)
- Isolated 0/1 handled efficiently
- If \( x_{i-1} \) is an isolated 1, \( y_{i-1} = 1 \) - only a single operation needed
- Similarly - \( x_{i-1} \) an isolated 0 in a string of 1's -...10(1)... recoded as ...11... or ...01... - single operation performed
- Exercise: To find required operation - calculate \( x_{i-1} + x_{i-2} - 2x_i \) for odd \( i \)'s and represent result as a 2-bit binary number \( y_i y_{i-1} \) in SD

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( x_{i-1} )</th>
<th>( x_{i-2} )</th>
<th>( y_i )</th>
<th>( y_{i-1} )</th>
<th>operation</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+0</td>
<td>string of zeros</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( \bar{1} )</td>
<td>0</td>
<td>(-2A)</td>
<td>beginning of 1's</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( \bar{1} )</td>
<td>(-A)</td>
<td>beginning of 1's</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(+A)</td>
<td>end of 1's</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(+2A)</td>
<td>end of 1's</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( \bar{1} )</td>
<td>(-A)</td>
<td>a single 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>+0</td>
<td>string of 1's</td>
</tr>
</tbody>
</table>
Radix-4 vs. Radix-2 Algorithm

- $01|01|01|01|(0)$ yields $01|01|01|01|$, number of operations remains 4 - the minimum
- $00|10|10|10|(0)$ yields $01|01|01|10|$, requiring 4, instead of 3, operations
- Compared to radix-2 Booth's algorithm - less patterns with more partial products; Smaller increase in number of operations
- Can design $n$-bit synchronous multiplier that generates exactly $n/2$ partial products
- Even $n$ - two's complement multipliers handled correctly; Odd $n$ - extension of sign bit needed
- Adding a 0 to left of multiplier needed if unsigned numbers are multiplied and $n$ odd - 2 0’s if $n$ even
Example

- $n/2=3$ steps; 2 multiplier bits in each step
- All shift operations are 2 bit position shifts
- Additional bit for storing correct sign required to properly handle addition of $2A$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$X$</th>
<th>$Y$</th>
<th>$\bar{A}$</th>
<th>$+2A$</th>
<th>$-A$</th>
<th>$17$</th>
<th>$-9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>11</td>
<td>01</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>01</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add $-A$</td>
<td>+</td>
</tr>
<tr>
<td>2-bit Shift</td>
<td>1 11 10 11 11</td>
</tr>
<tr>
<td>Add $2A$</td>
<td>+</td>
</tr>
<tr>
<td>2-bit Shift</td>
<td>0 10 00 10</td>
</tr>
<tr>
<td>Add $-A$</td>
<td>+</td>
</tr>
<tr>
<td>Result</td>
<td>01 11 01 11</td>
</tr>
<tr>
<td>2-bit Shift</td>
<td>00 01 11 01 11</td>
</tr>
<tr>
<td>Add $-A$</td>
<td>+</td>
</tr>
<tr>
<td>Result</td>
<td>11 01 10 01 11</td>
</tr>
</tbody>
</table>
Parallel Multiplier

Speeding up the addition of partial products by using adding trees
Accumulating the Partial Products

- After generating partial products either directly or using smaller multipliers
- Accumulate these to obtain final product
  - A fast multi-operand adder
- Should take advantage of particular form of partial products - reduce hardware complexity
- They have fewer bits than final product, and must be aligned before added
- Expect many columns that include fewer bits than total number of partial products - requiring simpler counters
Example - Six Partial Products

- Generated when multiplying unsigned 6-bit operands using one-bit-at-a-time algorithm
- 6 operands can be added using 3-level carry-save tree
- Number of (3,2) counters can be substantially reduced by taking advantage of the fact that all columns but 5 contain fewer than 6 bits
- Deciding how many counters needed - redraw matrix of bits to be added:
Reduce Complexity - Use (2,2) Counters (HAs)

(a) Level 1 carry-save addition.

(b) Results of level 1.

(c) Level 2 carry-save addition.

(d) Level 3 carry-save addition.

\[ \square \text{ Number of levels still 3, but fewer counters } \]
Modified Matrix for Negative Numbers

- Sign bits must be properly extended
- In row 1: 11 instead of 6 bits, and so on
- Increases complexity of multi-operand adder
- If two's complement obtained through one's complement - matrix increased even further
Adding Signed Numbers

- The previous technique to reduce the number of counters cannot be employed in signed number multiplication.

- Modified Booth’s Algorithm can be employed to reduce the number of addition and then a Sign-Digit (SD) Adder should be employed in the adder tree.
Alternative Techniques for Partial Product Accumulation

- Reducing number of levels in tree - speeding up accumulation
- Achieving more regular design
- Tree structures usually have irregular interconnects
  - Irregularity complicates implementation - area-inefficient layouts
- Number of tree levels can be lowered by using reduction rate higher than 3:2
- Achieve 2:1 reduction rate by using SD adders
  - SD adder also generates sum in constant time
  - Number of levels in SD adder tree is smaller
  - Tree produces a single result rather than two for CSA tree
SD Adder Tree vs. CSA Tree

- **SD** - no need for a sign bit extension when negative partial products - no separate sign bit
- Design of **SD** adder more complex - more gates and larger chip area - each signed digit requires two ordinary bits (or multiple-valued logic)
- Comparison between the two must be made for specific technology
- **Example:**
  - **32x32** Multiplier based on **radix-4** modified Booth's algorithm - 16 partial products
  - CSA tree with **6** levels, **SD** adder tree with **4** levels
  - Sophisticated logic design techniques and layout schemes result in less area-consuming implementations
(4;2) Compressors

- Same reduction rate of 2:1 achieved without SD representations by using (4;2) compressors.
- Designed so that $c_{out}$ is not a function of $c_{in}$ to avoid a ripple-carry effect.
- (4;2) compressor may be implemented as a multi-level circuit with a smaller overall delay compared to implementation based on 2 (3,2) counters.
Example Implementation

- Delay of 3 exclusive-or gates to output $S$ vs. delay of 4 exclusive-or gates
  - 25% lower delay
Other Multi-Level Implementations of a (4;2) Compressor

- All implementations must satisfy

\[ x_1 + x_2 + x_3 + x_4 + c_{in} = S + 2(C + c_{out}) \]

- \( c_{out} \) should not depend on \( c_{in} \) to avoid horizontal rippling of carries

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( c_{out} )</th>
<th>( C )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( c_{in} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( a )</td>
<td>( \bar{a} )</td>
<td>( c_{in} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( b )</td>
<td>( \bar{b} )</td>
<td>( c_{in} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( c )</td>
<td>( \bar{c} )</td>
<td>( c_{in} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( 1 )</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( c_{out} )</th>
<th>( C )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( d )</td>
<td>( \bar{d} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( e )</td>
<td>( \bar{e} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( 1 )</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( f )</td>
<td>( \bar{f} )</td>
<td>( c_{in} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( 1 )</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( 1 )</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( 1 )</td>
<td>( c_{in} )</td>
<td>( \overline{c_{in}} )</td>
</tr>
</tbody>
</table>
Complete Structure of Wallace Tree

(a) Wallace tree.

- Balanced and overturned-stairs have regular structure - can be designed in a systematic way
Complete Structure of Over-turned Tree

(b) Overturned-stairs tree.

- Building blocks indicated with dotted lines
Complete Structure of Balanced Tree

- Building blocks indicated with dotted lines

(c) Balanced tree.
Fused Multiply-Add Unit

- Performs \(A \times B\) followed by adding \(C\)
  - \(A \times B + C\) done as single and indivisible operation
- Multiply only: set \(C=0\); add (subtract) only: set \(B=1\)
  - Can reduce overall execution time of chained multiply and then add/subtract operations
- Example: Evaluation of a polynomial \(a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0\) through \([(a_n x + a_{n-1}) x + a_{n-2}] x + \ldots\)
- Independent multiply and add operations can not be performed in parallel
- Another advantage for floating-point operations - rounding performed only once for \(A \times B + C\) rather then twice for multiply and add
  - Rounding introduces computation errors - reducing number of roundings reduces overall error
Implementing Fused Multiply-Add Unit

- **A, B, C** - significants; **E_A, E_B, E_C** - exponents of operands
- **CSA** tree generates partial products and performs carry-save accumulation to produce 2 results which are added with properly aligned C
- Adder gets 3 operands - first reduces to 2 \((3,2)\) counters, then performs carry-propagate addition
- Post-normalization and rounding executed next
Array Multipliers

Generation and accumulation of partial products is done at the same time by an array of similar cells.
The two basic operations - generation and summation of partial products - can be merged, avoiding overhead and speeding up multiplication.

Iterative array multipliers (or array multipliers) consist of identical cells, each forming a new partial product and adding it to previously accumulated partial product.

- Gain in speed obtained at expense of extra hardware.
- Can be implemented so as to support a high rate of pipelining.
**Illustration - 5 x 5 Multiplication**

<table>
<thead>
<tr>
<th></th>
<th>$a_4$</th>
<th>$a_3$</th>
<th>$a_2$</th>
<th>$a_1$</th>
<th>$a_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_4$</td>
<td>$a_4 \cdot x_0$</td>
<td>$a_3 \cdot x_0$</td>
<td>$a_2 \cdot x_0$</td>
<td>$a_1 \cdot x_0$</td>
<td>$a_0 \cdot x_0$</td>
</tr>
<tr>
<td></td>
<td>$a_4 \cdot x_1$</td>
<td>$a_3 \cdot x_1$</td>
<td>$a_2 \cdot x_1$</td>
<td>$a_1 \cdot x_1$</td>
<td>$a_0 \cdot x_1$</td>
</tr>
<tr>
<td></td>
<td>$a_4 \cdot x_2$</td>
<td>$a_3 \cdot x_2$</td>
<td>$a_2 \cdot x_2$</td>
<td>$a_1 \cdot x_2$</td>
<td>$a_0 \cdot x_2$</td>
</tr>
<tr>
<td></td>
<td>$a_4 \cdot x_3$</td>
<td>$a_3 \cdot x_3$</td>
<td>$a_2 \cdot x_3$</td>
<td>$a_1 \cdot x_3$</td>
<td>$a_0 \cdot x_3$</td>
</tr>
<tr>
<td></td>
<td>$a_4 \cdot x_4$</td>
<td>$a_3 \cdot x_4$</td>
<td>$a_2 \cdot x_4$</td>
<td>$a_1 \cdot x_4$</td>
<td>$a_0 \cdot x_4$</td>
</tr>
</tbody>
</table>

- **Straightforward implementation** -
  - Add first 2 partial products ($a_4x_0, a_3x_0, \ldots, a_0x_0$ and $a_4x_1, a_3x_1, \ldots, a_0x_1$) in row 1 after proper alignment
  - The results of row 1 are then added to $a_4x_2, a_3x_2, \ldots, a_0x_2$ in row 2, and so on
5 x 5 Array Multiplier for Unsigned Numbers (Braun Multiplier)

- Basic cell - FA accepting one bit of new partial product $a_i x_j$ + one bit of previously accumulated partial product + carry-in bit

- No horizontal carry propagation in first 4 rows - carry-save type addition - accumulated partial product consists of intermediate sum and carry bits

- Last row is a ripple-carry adder - can be replaced by a fast 2-operand adder (e.g., carry-look-ahead adder)
Array Multiplier for Two’s Complement Numbers

- Product bits like $a_4 x_0$ and $a_0 x_4$ have negative weight
- Should be subtracted instead of added
Type I and II Cells

- Type I cells: 3 positive inputs - ordinary FAs
- Type II cells: 1 negative and 2 positive inputs
- Sum of 3 inputs of type II cell can vary from -1 to 2
  - c output has weight +2
  - s output has weight -1
- Arithmetic operation of type II cell -

\[ x + y - z = 2c - s \]

- s and c outputs given by

\[ s = (x + y - z) \mod 2 \]
\[ c = \frac{(x + y - z) + s}{2} \]
Type I’ and II’ Cells

- Type II' cells: 2 negative inputs and 1 positive
- Sum of inputs varies from -2 to 1
  - c output has weight -2
  - s output has weight +1

- Type I' cell: all negative inputs - has negatively weighted c and s outputs
- Counts number of -1's at its inputs - represents this number through c and s outputs
- Same logic operation as type I cell - same gate implementation
- Similarly - types II and II' have the same gate implementation
Booth’s Algorithm Array Multiplier

- For two's complement operands
- \( n \) rows of basic cells - each row capable of adding or subtracting a properly aligned multiplicand to previously accumulated partial product
  - Cells in row \( i \) perform an add, subtract or transfer-only operation, depending on \( x_i \) and reference bit

- 4-bit operands
- Controlled add/subtract/shift (CASS)
Booth’s Algorithm Array Multiplier - details

- First row - most significant bit of multiplier
- Resulting partial product need be shifted left before adding/subtracting next multiple of multiplicand
- A new cell with input Pin=0 is added

♦ Number of bits in partial product increases by one each row - expand multiplicand before adding/subtracting it
♦ Accomplished by replicating sign bit of multiplicand
Properties and Delay

- Cannot take advantage of strings of 0's or 1's - cannot eliminate or skip rows
- Only advantage: ability to multiply negative numbers in two's complement with no need for correction
- Operation in row $i$ need not be delayed until all upper $(i-1)$ rows have completed their operation
- $P_0$, generated after one CASS delay (plus delay of CTRL), $P_1$ generated after two CASS delays, and $P_{2n-2}$, generated after $(2n-1)$ CASS delays
- Similarly can implement higher-radix multiplication requiring less rows
- Building block: multiplexer-adder circuit that selects correct multiple of multiplicand $A$ and adds it to previously accumulated partial product
Pipelining

- Important characteristic of array multipliers - allow pipelining
- Execution of separate multiplications overlaps
- The long delay of carry-propagating addition must be minimized
- Achieved by replacing CPA with several additional rows - allow carry propagation of only one position between consecutive rows
- To support pipelining, all cells must include latches - each row handles a separate multiplier-multiplicand pair
- Registers needed to propagate multiplier bits to their destination, and propagate completed product bits
Pipelined Array Multiplier

Latched full adder with an AND gate.

Latched half adders.
Basic Array Multiplier

- Very regular structure - can be implemented as a rectangular-shaped array - no waste of chip area

- $n$ least significant bits of final product are produced on right side of rectangle; $n$ most significant bits are outputs of bottom row of rectangle

- Highly regular and simple layout but has two drawbacks:
  - Requires a very large area, proportional to $n^2$, since it contains about $n^2$ FAs and AND gates
  - Long execution time $T$ of about $2n \Delta_{FA}$ ($\Delta_{FA}$ - delay of FA)

- More precisely, $T$ consists of $(n-1)\Delta_{FA}$ for first $(n-1)$ rows and $(n-1)\Delta_{FA}$ for CPA (ripple-carry adder)

- $AT$ is proportional to $n^3$
Pipelined & Booth Array Multipliers

- Required area increases even further (CPA replaced)
- Latency of a single multiply operation increases
- However, throughput increases.
- Booth based array multiplier offers no advantage
  - \( A \) - order of \( n^2 \) and \( T \) - linear in \( n \)
- Radix-4 Booth can potentially be better - only \( n/2 \) rows - could reduce \( T \) and \( A \) by factor of two
Pipelined & Booth Array Multipliers

- However, actual delay & area are higher - recoding logic and, more importantly, partial product selectors, add complexity & interconnections - longer delay per row

- Also, since relative shift between adjacent rows is two bits, must allow carry to propagate horizontally
  - Can be achieved locally or in last row - then carry propagation through $2n-1$ bits (instead of $n-1$)

- Exact reduction depends on design and technology
Radix-8 Booth & CSA Tree

- Similar problems with radix-8 Booth's array multiplier
  - In addition, 3A should be precalculated
  - Reduction in delay and area may be less than expected 1/3
  - Still, may be cost-effective in certain technologies and design styles

- Partial products can be accumulated using a cascade or a tree structure with shorter execution time

- But CSA tree structures have irregular interconnects - no area-efficient layout with a rectangular shape

- Moreover - overall width $2n$ usually required - multiplier area of order $2n \log k$

- $AT$ may increase as $2n \log^2 k$