Multi-input Adders

- Suppose we want to add $k$ $N$-bit words
  - Ex: $0001 + 0111 + 1101 + 0010 = 10111$
- Straightforward solution: $k-1$ $N$-input CPAs
  - Large and slow
Carry Save Addition

- A full adder sums 3 inputs and produces 2 outputs
  - Carry output has twice weight of sum output
- N full adders in parallel are called *carry save adder*
  - Produce N sums and N carry outs

\[
\begin{array}{cccc}
X_4 & Y_4 & Z_4 & X_3 \quad Y_3 \quad Z_3 \quad X_2 \quad Y_2 \quad Z_2 \quad X_1 \quad Y_1 \quad Z_1 \\
C_4 & S_4 & C_3 & S_3 & C_2 & S_2 & C_1 & S_1 \\
\end{array}
\]

\[
\begin{array}{cccc}
X_{N\ldots1} & Y_{N\ldots1} & Z_{N\ldots1} \\
C_{N\ldots1} & S_{N\ldots1} \\
\end{array}
\]

n-bit CSA
CSA Application

- Use k-2 stages of CSAs
  - Keep result in carry-save redundant form
- Final CPA computes actual result
CSA Application

- Use k-2 stages of CSAs
  - Keep result in carry-save redundant form
- Final CPA computes actual result
Multiplication

- Example:
  \[
  \begin{array}{c}
  1100 : 12_{10} \\
  0101 : 5_{10}
  \end{array}
  \]

  multiplier

  multiplicand

  partial products

  product

- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product
Multiplication

Example:

\[
\begin{array}{c}
1100 : 12_{10} \\
0101 : 5_{10} \\
1100 \\
0000 \\
1100 \\
0000 \\
\hline
00111100 : 60_{10}
\end{array}
\]

- multiplicand
- multiplier
- partial products
- product

- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product
# General Form

- **Multicand:** \( Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0) \)
- **Multiplier:** \( X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0) \)
- **Product:** \[
P = \left( \sum_{j=0}^{M-1} y_j 2^j \right) \left( \sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}
\]

<table>
<thead>
<tr>
<th>( y_5 )</th>
<th>( y_4 )</th>
<th>( y_3 )</th>
<th>( y_2 )</th>
<th>( y_1 )</th>
<th>( y_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_5 )</td>
<td>( x_4 )</td>
<td>( x_3 )</td>
<td>( x_2 )</td>
<td>( x_1 )</td>
<td>( x_0 )</td>
</tr>
<tr>
<td>( x_0 y_5 )</td>
<td>( x_0 y_4 )</td>
<td>( x_0 y_3 )</td>
<td>( x_0 y_2 )</td>
<td>( x_0 y_1 )</td>
<td>( x_0 y_0 )</td>
</tr>
<tr>
<td>( x_1 y_5 )</td>
<td>( x_1 y_4 )</td>
<td>( x_1 y_3 )</td>
<td>( x_1 y_2 )</td>
<td>( x_1 y_1 )</td>
<td>( x_1 y_0 )</td>
</tr>
<tr>
<td>( x_2 y_5 )</td>
<td>( x_2 y_4 )</td>
<td>( x_2 y_3 )</td>
<td>( x_2 y_2 )</td>
<td>( x_2 y_1 )</td>
<td>( x_2 y_0 )</td>
</tr>
<tr>
<td>( x_3 y_5 )</td>
<td>( x_3 y_4 )</td>
<td>( x_3 y_3 )</td>
<td>( x_3 y_2 )</td>
<td>( x_3 y_1 )</td>
<td>( x_3 y_0 )</td>
</tr>
<tr>
<td>( x_4 y_5 )</td>
<td>( x_4 y_4 )</td>
<td>( x_4 y_3 )</td>
<td>( x_4 y_2 )</td>
<td>( x_4 y_1 )</td>
<td>( x_4 y_0 )</td>
</tr>
<tr>
<td>( x_5 y_5 )</td>
<td>( x_5 y_4 )</td>
<td>( x_5 y_3 )</td>
<td>( x_5 y_2 )</td>
<td>( x_5 y_1 )</td>
<td>( x_5 y_0 )</td>
</tr>
</tbody>
</table>

- **Partial Products**
- **Product**
Each dot represents a bit
Array Multiplier

\[ x_0 \quad x_1 \quad x_2 \quad x_3 \]

\[ p_0 \quad p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad p_6 \quad p_7 \]

\[ y_0 \quad y_1 \quad y_2 \quad y_3 \]

\[ B \quad A \]

\[ \text{critical path} \]

\[ \text{CSA Array} \]

\[ \text{CPA} \]

\[ \text{Sin A Cin} \]

\[ \text{Cout} \quad \text{Sout} \]

\[ \text{A B} \]

\[ \text{Sin} \]

\[ \text{Cout} \quad \text{Cin} \quad \text{Sout} \]

\[ \text{Cout} \quad \text{Cin} \quad \text{Sout} \]
Rectangular Array

- Squash array to fit rectangular floorplan
Fewer Partial Products

- Array multiplier requires N partial products
- If we looked at groups of r bits, we could form N/r partial products.
  - Faster and smaller?
  - Called radix-$2^r$ encoding
- Ex: r = 2: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder 😞
Booth Encoding

- Instead of 3Y, try \(-Y\), then increment next partial product to add 4Y
- Similarly, for 2Y, try \(-2Y + 4Y\) in next partial product

<table>
<thead>
<tr>
<th>Inputs (x_{2i+1})</th>
<th>Inputs (x_{2i})</th>
<th>Inputs (x_{2i-1})</th>
<th>Partial Product (PP_i)</th>
<th>SINGLE (_i)</th>
<th>DOUBLE (_i)</th>
<th>NEG (_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Y</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Booth Encoding

- Instead of 3Y, try –Y, then increment next partial product to add 4Y
- Similarly, for 2Y, try –2Y + 4Y in next partial product

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Partial Product</th>
<th>Booth Selects</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_{2i+1})</td>
<td>(x_{2i})</td>
<td>(x_{2i-1})</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Primary Inputs
Partial products can be negative
  - Require sign extension, which is cumbersome
  - High fanout on most significant bit
Booth Example

- $Y \times X; Y = 011001; X = 100111$ (unsigned)
- $(N+1)/2 = 4$ partial products

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Partial Product $PP_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{2i+1}$</td>
<td>$x_{2i}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sign Extension

$PP_0 = -Y$

$PP_1 = 2Y$

$PP_2 = -2Y$

$PP_3 = Y$

\[
\begin{array}{ccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]
Booth Hardware

- Booth encoder generates control lines for each PP
  - Booth selectors choose PP bits

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Partial Product</th>
<th>Booth Selects</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{2i-1}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Simplified Sign Ext.

- Sign bits are either all 0’s or all 1’s
  - Note that all 0’s is all 1’s + 1 in proper column
  - Use this to reduce loading on MSB
Even Simpler Sign Ext.

- No need to add all the 1’s in hardware
  - Precompute the answer!
Wallace Trees

- Perform PP additions in parallel
- Reduces stages, but yields irregular layout with long wires
- Requires $\log_{3/2}(N/2)$ stages