MULTIPLICATION

Chapter four...

What’s the Deal?

- Time to multiply
  - Multiplying two numbers requires a multiply
  - Luckily, in binary that’s just an AND gate!
    - $0*0=0$, $0*1=0$, $1*0=0$, $1*1=1$
  - Generate a bunch of partial products and add ‘em up!
    - Grade-school algorithm
    - From Chapter 1
    - Modify slightly for two’s comp…
Overview

\[ p = x \times y \]
\( x \) (multiplicand), \( y \) (multiplier), and \( p \) (product) signed integers

- SCHEMES
  - a) SEQUENTIAL ADD-SHIFT RECURRENCE
    - CPA, CSA, SIGNED-DIGIT ADDER
    - HIGHER RADIX AND RECODING
  - b) COMBINATIONAL
    - CPA, CSA, SIGNED-DIGIT ADDER
    - HIGHER RADIX AND RECODING
  - c) COLUMN REDUCTION
  - d) ARRAYS WITH \( k \times l \) MULTIPLIERS

Overview

- MULTIPLY-ADD AND MULTIPLY-ACCUMULATE
- SATURATING MULTIPLIERS
- TRUNCATING MULTIPLIERS
- RECTANGULAR MULTIPLIERS
- SQUARERS
- CONSTANT AND MULTIPLE CONSTANT MULTIPLIERS
Sign and Magnitude

- EACH OPERAND:
  - sign with value +1 and −1 and \( n \)-digit magnitude
- RESULT: a sign and a \( 2n \)-digit magnitude
- HIGH-LEVEL ALGORITHM
  \[ \text{sign}(p) = \text{sign}(x) \cdot \text{sign}(y) \]
  \[ |p| = |x||y| \]
- REPRESENTATIONS OF MAGNITUDES

  \[ X = (x_{n-1}, x_{n-2}, \ldots, x_0) \]
  \[ |x| = \sum_{i=0}^{n-1} x_i r^i \] (multiplicand)

  \[ Y = (y_{n-1}, y_{n-2}, \ldots, y_0) \]
  \[ |y| = \sum_{i=0}^{n-1} y_i r^i \] (multiplier)

  \[ P = (p_{2n-1}, p_{2n-2}, \ldots, p_0) \]
  \[ |p| = \sum_{i=0}^{2n-1} p_i r^i \] (product)

Two’s Complement

- RADIX-2 CASE
- EACH OPERAND: \( n \)-BIT VECTOR
- RESULT: \( 2n \)-BIT VECTOR
  \[-(2^{n-1})(2^{n-1} - 1) \leq p \leq (-2^{n-1})(-2^{n-1}) = 2^{2n-2}\]
- \( x_R, y_R \) and \( p_R \) – positive integer representations of \( x, y, \) and \( p \)
- HIGH-LEVEL ALGORITHM
  \[
p_R = \begin{cases} 
    x_R y_R & \text{if } x \geq 0, \ y \geq 0 \\
    2^{2n} - (2^n - x_R)y_R & \text{if } x < 0, \ y \geq 0 \\
    2^{2n} - x_R(2^n - y_R) & \text{if } x \geq 0, \ y < 0 \\
    (2^n - x_R)(2^n - y_R) & \text{if } x < 0, \ y < 0
  \end{cases}
\]
Multiplication (unsigned)

- Instead of using n-1 adders, can iterate with one adder
  - $p[0] = 0$
  - $p[j+1] = r^{-1}(p[j] + xr^jy_j)$ for $j = 0, 1, ..., n-1$
  - $p = p[n]$

- Start with iterative version
- Takes n steps for n bits

Our Book’s Version
Example – n=5, A=27, B=23

\[
\begin{array}{c|c}
P & A \\
\hline
0 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0 \\
\end{array}
\]

\[B = X = 11011\]
\[A = Y = 10111\]
Example – n=5, A=27, B=23

B = X = 11011
A = Y = 10111

Example – n=5, A=27, B=23

B = X = 11011
A = Y = 10111
Example – n=5, A=27, B=23

\[
\begin{array}{l}
\text{P} & \text{A} \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{l}
1 \times B \\
\text{shift} \\
1 \times B \\
\text{shift} \\
1 \times B \\
\text{shift} \\
1 \times B \\
\text{shift} \\
\end{array}
\]

B = X = 11011
A = Y = 10111

Example – n=5, A=27, B=23

\[
\begin{array}{l}
\text{P} & \text{A} \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{l}
1 \times B \\
\text{shift} \\
1 \times B \\
\text{shift} \\
1 \times B \\
\text{shift} \\
1 \times B \\
\text{shift} \\
\end{array}
\]

B = X = 11011
A = Y = 10111
Example – n=5, A=27, B=23

\[
\begin{array}{c|c}
P & A \\
0 0 0 0 0 & 1 \times B \\
1 1 0 1 1 & B = X = 11011 \\
0 1 1 0 1 1 & shift \quad A = Y = 10111 \\
0 1 1 0 1 1 & 1 \times B \\
1 1 0 1 1 & shift \\
1 0 1 0 0 0 1 & 1 \times B \\
1 1 0 1 1 & shift \\
1 0 1 1 1 1 0 1 & 0 \times B \\
0 0 0 0 & 0 \times B \\
0 1 0 1 1 1 & shift \\
0 1 0 1 1 1 1 0 1 & shift \quad Answer = 621
\end{array}
\]
Our Book’s Version

Pretty much exactly the same as before, but with carry-save results at each iteration.

Speeds things up by not waiting for carries during the iterations.

But, you need to convert the final answer…
Example – n=5, A=27, B=23

B = X = 11011
A = Y = 10111

Example – n=5, A=27, B=23

B = X = 11011
A = Y = 10111
Example – n=5, A=27, B=23

PH

0 0 0 0 0
0 0 0 0 0
1 1 0 1 1

0 1 1 0 1
0 0 0 0 0
1

PL

B = X = 11011
A = Y = 1011

Example – n=5, A=27, B=23

PH

0 0 0 0 0
0 0 0 0 0
1 1 0 1 1

0 1 1 0 1
0 0 0 0 0
1

PL

B = X = 11011
A = Y = 1011
Example – \( n=5, A=27, B=23 \)

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

**B** = \( X = 11011 \)

**A** = \( Y = 10111 \)
Example – n=5, A=27, B=23

$$\begin{array}{ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}$$

B = X = 11011
A = Y = 10111

Example – n=5, A=27, B=23

$$\begin{array}{ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}$$

B = X = 11011
A = Y = 10111

\[PH \rightarrow PL\]

\[PH \rightarrow PL\]
### Example – n=5, A=27, B=23

<table>
<thead>
<tr>
<th>0 0 0 0 0</th>
<th>B = X = 11011</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>A = Y = 10111</td>
</tr>
<tr>
<td>1 1 0 1 1</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>PH</th>
<th>PL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 1 1</td>
<td>0 1 1 0 1</td>
</tr>
</tbody>
</table>

\[ \text{PH} = 621 \]
Carry-Save Iteration

- Still takes $n$ cycles for $n$ bits
  - But, each cycle is much faster
  - No carries in the PP summation
  - Half the bits actually get converted to conventional form during the iteration
  - Have to convert the upper bits with a CPA

- Can we do better?

- How about multiplying by more than one bit per iteration?

Radix-4 Recoding

- Use higher radix for multiplier
  - If radix is $2^k$, effectively considering $k$ bits of the multiplier in each iteration
  - Naïve approach:

<table>
<thead>
<tr>
<th>$i+1$</th>
<th>$i$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Add 0 to P</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Add B to P</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Add 2B to P</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Add 3B to P</td>
</tr>
</tbody>
</table>

- Problem – 3B is “hard”
  - 2B+B, but that involves adding a third argument
Recoding to Avoid 3

- Recode multiplier into (non-redundant) digit set \{-1,0,1,2\}
  - This will produce a carry, so take that into account

- MULTIPLIER RECODING TO AVOID VALUES \( z_i = 3 \)

\[
y_i + c_i | z_i | c_{i+1}
\begin{array}{ccc}
0 & 0 & 0 \\
1 & 1 & 0 \\
2 & 2 & 0 \\
3 & -1 & 1 \\
4 & 0 & 1
\end{array}
\]

= 4-1
= 4-0

Radix-4 Recoding

- If \( z \) is recoded version of \( y \)
  - Now think of \( z_i \) as a radix-4 bit
  - Recoded in \{-1,0,1,2\} to avoid multiple of 3
  - Now you can shift by 2 each iteration because you’re multiplying by a radix-4 (two-bit) quantity on each iteration

- Pretty much the same hardware
  - Add a recoder...
  - Add a selector to choose between \( X, X' \), and \( 2X \)
  - Remember to add 1 if you’re subtracting (i.e. \( X' + 1 \) is \(-X\)) (can do this “for free” in the following stage...)

\[
y_i + c_i | z_i | c_{i+1}
\begin{array}{ccc}
0 & 0 & 0 \\
1 & 1 & 0 \\
2 & 2 & 0 \\
3 & -1 & 1 \\
4 & 0 & 1
\end{array}
\]
Radix-4 Multiplier

Example – n=5, X=27, Y=23

- $X=11011$
- $Y=01011$
  - $11+0 = 3, z=-1, c=1$
  - $01+1 = 2, z=2, c=0$
  - $01+0 = 1, z=1, c=0$
- $Z=121$

- CSA extended to 8 bits
  - Range of result

\[
z_i = y_i + c_i - 4c_{i+1}
\]

<table>
<thead>
<tr>
<th>$y_i$</th>
<th>$c_i$</th>
<th>$z_i$</th>
<th>$c_{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example – n=5, X=27, z=12

Y = 01011

Invert the bits and add 1 (in the following stage) to compute -x

Remember to sign-extend the negative terms... We now have signed PPs because of the -1 multiplier bit
Example – $n=5$, $X=27$, $z=121$

X = 11011

$11 + 0 = 3 = -X + \text{carry}$

$-1 \times X$

Y = 01011

01 + 1 = 2 = 2X

Shift the term to the left to get $2X$
Example – n=5, X=27, z=121

\[ Y = 010111 \]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

\[ -1 \times X \]

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[ 2 \times X \]

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[ 01 + 1 = 2 = 2X \]

Example – n=5, X=27, z=121

\[ Y = 010111 \]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

\[ -1 \times X \]

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[ 2 \times X \]

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[ 01 + 1 = 2 = 2X \]
Example – n=5, X=27, z=121

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

X=11011

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Y = 010111

Example – n=5, X=27, z=121

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

X=11011

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Y = 010111

01 + 0 = 1 = X
Example – $n=5$, $X=27$, $z=121$

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[X=11011\]

\[Y = \overset{01}{01011}\]

\[01 + 0 = 1 = X\]

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[X=11011\]

\[Y = \overset{1}{01011}\]

\[1 + 0 = 1 = X\]

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[X=11011\]

\[Y = \overset{1}{01011}\]

\[1 + 0 = 1 = X\]

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[X=11011\]

\[Y = \overset{1}{01011}\]

\[1 + 0 = 1 = X\]

\[\text{Look at } 2n \text{ bits for the answer...}\]

\[\text{Note: only 3 iterations instead of 5 without recoding}\]
Hardware for Recoding

- Design some hardware based on the recoding truth table
  - Code in terms of:
    - One: $1 = x$, $0 = 2x$
    - Neg: $1 = \text{complement}$, $0 = \text{direct}$
    - Zero: $1 = \text{force-0}$, $0 = \text{direct}$

$$z_i = y_i + c_i - 4c_{i+1}$$

<table>
<thead>
<tr>
<th>$y_i$</th>
<th>$c_i$</th>
<th>$z_i$</th>
<th>$c_{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>Zero</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>Zero</td>
</tr>
</tbody>
</table>

Radix-4 Multiplier

- One: $1 = x$, $0 = 2x$
- Neg: $1 = \text{complement}$, $0 = \text{direct}$
- Zero: $1 = \text{direct}$, $0 = \text{force-0}$

$$z_i = y_i + c_i - 4c_{i+1}$$

<table>
<thead>
<tr>
<th>$y_i$</th>
<th>$c_i$</th>
<th>$z_i$</th>
<th>$c_{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hardware for Recoding

- BASED ON MULTIPLIER BITS \((M_1, M_0)\) and CARRY FLAG \(C\)

\[
\begin{align*}
one &= M_0 \oplus C = \\
&\begin{cases}
0 & \text{select } 2x \\
1 & \text{select } x
\end{cases}
\end{align*}
\]

\[
\begin{align*}
neg &= M_1 \cdot C \cdot M_1 \cdot M_0 = \\
&\begin{cases}
0 & \text{select direct} \\
1 & \text{select complement}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
zero &= M_1 \cdot M_0 \cdot C \cdot M'_1 \cdot M'_0 \cdot C' = \\
&\begin{cases}
0 & \text{load non-zero multiple} \\
1 & \text{load zero multiple (clear)}
\end{cases}
\end{align*}
\]

\[
C_{next} = M_1 M_0 \bullet M_1 C
\]

<table>
<thead>
<tr>
<th>(y_i + c_i)</th>
<th>(z_i)</th>
<th>(c_{i+1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
z_i = y_i + c_i - 4c_{i+1}
\]

Figure 4.4: RECORDER IMPLEMENTATION.
What about Signed?

- **MULTIPLICAND IN 2’S COMPLEMENT** $\implies$ **ADDITION AND SHIFT OPERATIONS PERFORMED IN THIS SYSTEM**
- **THE EFFECT OF 2’S COMPLEMENT MULTIPLIER TAKEN INTO ACCOUNT IN TWO WAYS:**
  1. **BY SUBTRACTING INSTEAD OF ADDING IN THE LAST ITERATION**
     \[ y = -y_{n-1}2^{n-1} + \sum_{i=0}^{n-2} y_i 2^i \]
     $\implies$ **CORRECTION STEP**
  2. **BY RECODING THE MULTIPLIER INTO A SIGNED-DIGIT SET**
     \[ \text{This is “Booth Recoding”} \]

Original Booth Radix 2 Recoding

- **Recode the multiplier into** \{-1, 0, 1\}
- **0111 = 100T = 8-1**
- **1100 = 0T00 = -4**
- **Is there a pattern that lets us get the effect of recoding without actually recoding?**
  - **This is the original Booth Recoding algorithm...**
  - **The idea is to recode the numbers so you don’t have to remember to subtract the last term**
Original Booth Radix 2 Recoding

- Recode the multiplier into \{-1, 0, 1\}
- \(0111 = 1001 = 8 - 1\)
- \(1100 = 0T00 = -4\)
  - Look at two bits at a time, right to left
  - If the bit stays the same, add 0
  - If the bit changes from 0 to 1, subtract \(B\)
  - If the bit changes from 1 to 0, add \(B\)
  - See examples on the board…
    - Recoding examples
    - \(-6 \times -5 = 30\)

Radix 4 Booth Recoding

- Recall the naïve approach:

\[
\begin{array}{c|c|c}
 i & i + 1 & \text{Add} \\
\hline
 0 & 0 & Add 0 to \(P\) \\
 0 & 1 & Add \(B\) to \(P\) \\
 1 & 0 & Add 2B to \(P\) \\
 1 & 1 & Add 3B to \(P\) \((4B - B)\)
\end{array}
\]
**Radix 4 Booth Recoding**

- Recall the naïve approach:

<table>
<thead>
<tr>
<th>$i+1$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 → Add 0 to P</td>
</tr>
<tr>
<td>0</td>
<td>1 → Add B to P</td>
</tr>
</tbody>
</table>

Squint at it slightly differently:

- If MSB is 1, add B in at next stage (4B)
- also, If MSB is 1, subtract 2B
- If LSB is 1, add B

Modified Booth Recoding

Look at 3 bits instead of 2....

<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>B</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3B</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4B</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

If previous MSB is 1, add B into current P (4B)
If current MSB is 1, subtract 2B (and carry out = +4B)
If current LSB is 1, add B
Radix-4 Modified Booth Recoding

The only problem is, the circuit is wrong!

(also, remember to add in the carry at the next stage if you’re subtracting…)
More Modified Radix-4 Booth

- Use higher radices for even better reduction in number of iterations
  - Radix-2 Booth reduces one row at a time
    - Recodes into \{-1,0,1\} to handle signed...
  - Radix-4 booth reduces two bits at a time
    - Looks at three bits to choose PP
    - Recodes to \{-1,0,1,2\}
  - Radix-8 booth reduces three bits at a time
    - Looks at four bits to choose PP
    - Recode into \{-3,-2,-1,0,1,2,3,4\}

Higher Radix Booth
Radix-8 Booth

**Table 10.13** Radix-8 modified Booth encoding values

<table>
<thead>
<tr>
<th>$x_{n+2}$</th>
<th>$x_{n+1}$</th>
<th>$x_n$</th>
<th>$x_{n-1}$</th>
<th>Partial Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$Y'$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$Y'$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$2Y'$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$2Y'$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$3Y'$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$3Y'$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$4Y'$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$-4Y'$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$-3Y'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$-3Y'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$-2Y'$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$-2Y'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$-Y$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$-0$</td>
</tr>
</tbody>
</table>

Reduces 3 rows at a time

Encoder and selector are a little trickier…

Oops! More “hard” multiples of 3!

Pre-compute them once at the beginning of the operation…
Radix-8 Booth

Even Higher Radices?

- EXTENSION TO HIGHER RADICES REQUIRES PREPROCESSING OF MORE MULTIPLES

- ALTERNATIVE: USE SEVERAL RADIX-4 AND/OR RADIX-2 STAGES IN ONE ITERATION

EXAMPLE: RADIX-16 MULTIPLIER DIGIT \( \{0, \ldots, 15\} \) RECODED INTO A RADIX-16 SIGNED-DIGIT \( v_i \) IN THE SET \( \{-10, \ldots, 0, \ldots, 10\} \) AND DECOMPOSED INTO TWO RADIX-4 DIGITS \( u_i \) AND \( w_i \) SUCH THAT

\[
v_i = 4u_i + w_i \quad u_i, w_i \in \{-2, -1, 0, 1, 2\}
\]
Radix-16

Essentially doing two radix-4 steps at one time…

Even Higher Radices?

<table>
<thead>
<tr>
<th>Partial Product Selection Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier Bits</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
</tbody>
</table>

Starts to get a little silly…

Hard multiples of 3X, 5X, 7X (6X is just 2*3X)
Radix 4 Booth – Our Book’s Version

- Two’s complement (signed)
  - MSB has negative weight...
    - $(z_i = y_i + c_i - 4c_{i+1})$
      - $y_i$ | $c_i$ | $z_i$ | $c_{i+1}$
      - 0 0 0 0
      - 0 1 1 1
      - 1 1 0 0
      - 0 1 1 1
      - 1 1 0 1
      - 1 0 1 0
      - 1 0 0 0
      - 0 1 0 0
      - 0 0 1 1
      - 0 0 0 1

See example: $-5 \times -6 = 30$

Modify recoder for MSB only...

Example...

- $A = 100001 = -31$
  - $B = 111011 = -5$
  - $2B = 110110$
  - $-B = 000101$
  - $-2B = 001010$

Remember to use arithmetic shifts...


Array Multiplication (unsigned)

- Pencil and paper method
  \[ p = x \times y = x \sum_{i=0}^{n-1} Y_i r^i = \sum_{i=0}^{n-1} x r^i Y_i \]
  - Compute \( n \) terms of \( x r^i Y_i \) and then sum them
  - The \( i \)th term requires an \( i \)-position shift, and a multiplication of \( x \) by the single digit \( Y_i \)
  - Requires \( n-1 \) additions

Multiplication (unsigned)

\[
\begin{array}{c}
\text{multiplicand} & \quad 1101 \quad (13) \\
\text{multiplier} & \times \quad 1011 \quad (11) \\
\hline
1101 \\
1101 \\
1101 \\
0000 \\
1101 \\
\hline
10001111 \quad (143) \\
128 + 8 + 4 + 2 + 1 = 143
\end{array}
\]
Our running example as an array

\[\begin{array}{cccccc}
1 & 1 & 0 & 1 & 1 & = X = B = 27 \\
1 & 0 & 1 & 1 & 1 & = Y = A = 23 \\
1 & 1 & 0 & 1 & 1 & \\
1 & 1 & 0 & 1 & 1 & \\
1 & 1 & 0 & 1 & 1 & \\
0 & 0 & 0 & 0 & 0 & \\
1 & 1 & 0 & 1 & 1 & \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & = 621
\end{array}\]

Generate PPs, Add ‘em Up!

\[\begin{array}{ccccccccccccccccccc}
& y_5 & y_4 & y_3 & y_2 & y_1 & y_0 \\
x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\
x_5y_5 & x_4y_5 & x_3y_5 & x_2y_5 & x_1y_5 & x_0y_5 \\
x_5y_4 & x_4y_4 & x_3y_4 & x_2y_4 & x_1y_4 & x_0y_4 \\
x_5y_3 & x_4y_3 & x_3y_3 & x_2y_3 & x_1y_3 & x_0y_3 \\
x_5y_2 & x_4y_2 & x_3y_2 & x_2y_2 & x_1y_2 & x_0y_2 \\
x_5y_1 & x_4y_1 & x_3y_1 & x_2y_1 & x_1y_1 & x_0y_1 \\
x_5y_0 & x_4y_0 & x_3y_0 & x_2y_0 & x_1y_0 & x_0y_0 \\
p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}\]

FIG 10.68 Partial products

From Harris/Weste
Generate PPs, Add ‘em Up!

From Harris/Weste
Speed Things Up

- Chapter 3 was all about how to speed up the addition of multiple numbers
  - First Idea: Use carry-save adder to save time!
  - Use same sequential algorithm, but use carry-save for as much as you can...

Array w/ Carry-Save

From Harris/Weste
Note that you can squish the trapezoid into a rectangle for better layout.
What about Signed?

1. EXTEND RANGE BY REPLICATING THE SIGN BIT OF MULTIPLES
   • PRODUCT HAS $2n$ BITS

2. THE MULTIPLE $xy_{n-1}2^{n-1}$ SUBTRACTED INSTEAD OF ADDED
   \[ y = -y_{n-1}2^{n-1} + \sum_{i=0}^{n-2} y_i2^i \]
   • COMPLEMENT AND ADD

3. RECODE THE (2’S COMPLEMENT) MULTIPLIER INTO THE DIGIT SET {-1,0,1}
   • NO ADVANTAGE IN FOLLOWING THIS APPROACH

Sign-Extension Trick Again

• Simplification of sign extension based on
  \[ (-s) + 1 - 1 = (1 - s) - 1 = s' - 1 \]
Consequently,
\[
\begin{array}{cccc}
  x_{n-1}y_i & x_{n-2}y_i & \ldots & x_0y_i \\
\end{array}
\]

is replaced by
\[
\begin{array}{cccc}
  (x_{n-1}y_i)' & x_{n-2}y_i & \ldots & x_0y_i \\
-1
\end{array}
\]
Two’s Comp Array

Sign extend all PPs
Subtract last term (invert and add 1)

Apply sign extension trick…
Two’s comp of \(-15 \times 2^3\) is 10001000

(show computation)

Precompute \(x_0 \cdot y_3 + y_3 + 1\) and reshuffle some terms…

Another Derivation…

\[
P = \left( -y_{m-1} \cdot 2^{m-1} + \sum_{j=0}^{m-2} y_j \cdot 2^j \right) \left( -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i \right)
\]

\[
= \sum_{i=0}^{n-2} \sum_{j=0}^{m-2} x_i y_j \cdot 2^{i+j} + x_{n-1} \cdot y_{m-1} \cdot 2^{M+N+2} - \left( \sum_{i=0}^{n-2} x_i y_{m-1} \cdot 2^{i+m-1} + \sum_{j=0}^{m-2} x_{n-1} y_j \cdot 2^{j+n-1} \right)
\]

Look at the two’s comp of the two bit arrays and multiply them out….
Assemble the PPs

Simplify the PPs

Essentially the same result as in our book (different constant…)
Also called a Baugh-Wooley two’s complement signed multiplier
What about Booth?

- Can also apply Booth to array multiplication
  - Reduces the number of rows you need to add up
  - Same technique – recode multiplier into higher radix
  - Use Booth Encoder to decide which term
  - Use Booth MUX to select the terms
Modified Booth Recoding

Look at 3 bits instead of 2…….

<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>B</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3B</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4B</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

If previous MSB is 1, add B into current P (4B)
If current MSB is 1, subtract 2B (and carry out = +4B)
If current LSB is 1, add B

More Modified Radix-4
Array w/ Carry-Save

Booth Array w/ Carry-Save

From Harris/Weste
Modified Booth, Radix-4

Form negative by inverting and adding 1 in the next row...

Figure 10.76 Radix-4 Booth-encoded partial products with sign extension

Remember to sign-extend because of potentially negative terms!

Sign Extension Trick

- As usual – apply the trick for sign extension
  - Note that even unsigned Booth multiply needs sign extension

- Another version of the trick
  - Sign extension bits are all 1’s or all 0’s
    - If you add 1 to a string of 1’s (and ignore the carry) you get a string of 0’s
    - If you add 0 to a string of 1’s, nothing changes
  - You can use this to invert the 1’s based on a “control signal”
Sign Extension Trick

- So, replace $\text{ssssss}$ with $\text{111111}$

- If $s = 0$ this is $111111$
  $1 = 000000$

- If $s = 1$, this is $111111$
  $0 = 111111$

- You can then combine all those 1’s ahead of time...

What we started with...

[Diagram of Radix-4 Booth-encoded partial products with sign extension]
Example of sign extension

Replace \textit{sssss} with \textit{11111} \textit{s}

Now add them up ahead of time (show example)

Signed Version

- Last row (PP8) is always 0 (was Y or 0 in unsigned)
  - Because multiplier is sign extended so that $x_{17} = x_{16} = x_{15}$
  - This means according to the Radix-4 table, that the PP is always +0 or -0
Same Idea in Ercegovac/Lang

- **TWO CASES**
  1. BIT ARRAY ADDED BY A LINEAR ARRAY OF ADDERS
     - SEQUENTIAL RECODING INTO \{-1,0,1,2\} SUFFICIENT
  2. BIT ARRAY ADDED BY A TREE OF ADDERS
     - PARALLEL RECODING INTO \{-2,-1,0,1,2\} REQUIRED

Parallel Radix-4 Recoding

- **RADIX-2 MULTIPLIER**
  \[ y_{n-1}, y_{n-2}, \ldots, y_1, y_0 \]
  \( y_i \) - multiplier bit; \( v_j \in \{0,1,2,3\} \) - radix-4 multiplier digit
  \[ v_j = 2y_{2j+1} + y_{2j} \quad j = \left( \frac{n}{2} - 1, \ldots, 0 \right) \]

- **RECODING ALGORITHM**
  1. Obtain \( w_j \) and \( t_{j+1} \) such that
     \[ v_j = w_j + 4t_{j+1} \]
  2. Obtain
     \[ z_j = w_j + t_j \]

- **TO AVOID CARRY PROPAGATION:**
  \[ -2 \leq w_j \leq 1 \quad 0 \leq t_{j+1} \leq 1 \]
Bit-level Implementation

- radix-2 multiplier

\[ Y = (y_{m-1}, y_{m-2}, \ldots, y_0) \ y_i \in \{0,1\} \]

- recoded radix-4 multiplier

\[ Z = (z_{m-4}, z_{m-3}, \ldots, z_0) \ z_i \in \{-2, -1, 0, 1, 2\} \]

<table>
<thead>
<tr>
<th>( y_{j+1} )</th>
<th>( y_j )</th>
<th>( y_{j-1} )</th>
<th>( y_{j-2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 -2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10.12 Radix-4 modified Booth encoding values

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

Recoder Implementation

- \( sign = 1 \) if \( z_j \) is negative
- \( one = 1 \) if \( z_j \) is either 1 or -1
- \( two = 1 \) if \( z_j \) is either 2 or -2.

\[
\begin{align*}
\text{sign} &= y_{2j+1} \\
\text{one} &= y_{2j} \oplus y_{2j-1} \\
\text{two} &= y_{2j+1}y_{2j}y_{2j-1} + y_{2j+1}y_{2j}y_{2j-1}
\end{align*}
\]

- carry-in: \( c = sign \)
Gates...

Array for Unsigned

<table>
<thead>
<tr>
<th>33</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>z_{33}</td>
<td>s_3</td>
<td>s_2</td>
<td>s_1</td>
<td>s_0</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>z_{31}</td>
<td>s_f</td>
<td>s_f</td>
<td>s_f</td>
<td>s_f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>z_{29}</td>
<td>s_g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>z_{27}</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
</tbody>
</table>

(a)

| z_{30} | s'_1 | e | e | e | e | e | e | e | e | e | e | e | e |
| z_{31} | s_f | f | f | f | f | f | f | f | f | f | f | f | f | c_u |
| z_{29} | s_g | g | g | g | g | g | g | g | g | g | g | g | g | g | c_f |
| z_{27} | h | h | h | h | h | h | h | h | h | h | h | h | h | h | c_w |

-1 -1 -1

(b)

| z_{30} | 1 | s'_1 | s_0 | s_2 | e | e | e | e | e | e | e | e | e |
| z_{31} | s_f | f | f | f | f | f | f | f | f | f | f | f | f | c_u |
| z_{29} | s_g | g | g | g | g | g | g | g | g | g | g | g | g | g | c_f |
| z_{27} | h | h | h | h | h | h | h | h | h | h | h | h | h | h | c_w |

(c)

Figure 4.15: (a) IMPLEMENTATION OF RECORDER. (b) IMPLEMENTATION OF MULTIPLE GENERATOR.

Figure 4.16: RADIUS 4-BIT MATRIX FOR MULTIPLICATION OF MAGNITUDES (n = 7).
Array for Signed

Radix-8 Booth

Figure 4.11: BASED-MATRIX FOR 2’S COMPLEMENT MULTIPLICATION ($n = 8$)
Now Add up the PPs!

- We’ve seen a number of ways of building the PP arrays for Radix-2, Radix-4 Booth, etc.
  - Now we have to add them up
  - Back to Chapter 3...

Linear CSA array
Another View of a Linear Array

Trees instead of Linear
Re-shape the PP Array?

Reduction by Rows...
Remember – pipelineing

Figure 4.15: PIPELINED LINEAR CSA MULTIPLIER FOR POSITIVE INTEGERS \((n = 4)\)

Reduction by Columns

Figure 4.17: REDUCTION BY COLUMNS USING ESA and IRCA \((n = 8)\): Cost 36 FUs, 7 IRCA, 16x8 IRCA.
Remember…

- When you move from Radix-2 to Radix-4 you only reduce the levels of the tree by one
  - And you increase delay by adding the recoder!
  - For full arrays, Radix-4 may not be worth the cost
    - Although size will go down because of fewer PPs, so maybe it will...
  - Also, CPA at the bottom is now wider than for a linear array...

Notes on Final CPA

- You need one to convert to conventional
  - Typical fast adders assume that all bits show up at the same time
  - This is NOT true in multipliers
  - Because of PP tree shape…

![Diagram showing time and a region divided into Middle Region and LS Region]
Notes on Final CPA

- For LSR
  - Ripple might be fine
- For middle
  - Any fast adder
- For MSR
  - Carry-select might be best

Partially Combinational

Figure 4.19: RADIX 2^2 SEQUENTIAL MULTIPLIER USING CSA TREE.
Arrays of Smaller Multipliers

\[ p = a \times b \]

\[
A = (a_{k-1}, a_{k-2}, \ldots, a_0) \\
B = (b_{l-1}, b_{l-2}, \ldots, b_0) \\
P = (p_{k+l-1}, p_{k+l-2}, \ldots, p_0)
\]

- USE OF \( k \times l \) MODULES
- OPERANDS DECOMPOSED INTO DIGITS OF RADIX \( 2^k \) AND \( 2^l \)

\[
x = \sum_{i=0}^{(n/k)-1} x_i \cdot 2^{ki} \\
y = \sum_{j=0}^{(n/l)-1} y_j \cdot 2^{lj}
\]

\[
p = x \cdot y = \sum_{i=0}^{(n/k)-1} x_i y_j \cdot 2^{ki+lj} = \sum_{i,j=0}^{n} p_{i,j} \cdot 2^{ki+lj}
\]

\( (n/k) \times (n/l) \) MODULES NEEDED

---

12x12 multiply using 4x4 modules

\( a_x, b_x, c_x \) are 4-bit chunks of the 12 bit number

Now multiply those forms to get the final expression

\[
x = a_x \cdot 2^8 + b_x \cdot 2^4 + c_x
\]

\[
y = a_y \cdot 2^8 + b_y \cdot 2^4 + c_y
\]

\[
x \times y = a_x a_y \cdot 2^{16} + a_x b_y \cdot 2^{12} + b_x a_y \cdot 2^{12} + b_x b_y \cdot 2^8 + c_x a_y \cdot 2^8 + c_x b_y \cdot 2^4 + c_y a_y \cdot 2^4 + c_y b_y \cdot 2^0 + c_x c_y
\]

---

Figure 4.20: 12 x 12 MULTIPLICATION USING 4 x 4 MULTIPLIERS: BIT MATRIX
Multiply Accumulate

- Multiply-add: \( S = X \times Y + W \)

\[
\begin{array}{cccccccccccccccc}
13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
S_{a0} & s_a & s_e & e & e & e & e & e & e & e & e & e & e & e & e \\
S_{a1} & 1 & s_f & f & f & f & f & f & f & f & f & f & f & f & f \\
S_{a2} & 1 & s'_f & g & g & g & g & g & g & g & g & g & g & g & g \\
S_{a3} & h & h & h & h & h & h & h & c_d & w & w & w & w & w & w \\
w & w & w & w & w & w & w & w & w & w & w & w & w & w & w \\
\end{array}
\]

Figure 4.20: Radix-4 bit-serial for multiply add of magnitudes \( n = 7 \). \( x, y \) are radix-4 digits obtained by multiplier encoding.

- Multiply-accumulate:

\[
S = \sum_{i=1}^{m} X[i] \times Y[i]
\]

\[
S[i + 1] = X[i] \times Y[i] + S[i]
\]

Multiply Accumulate

Figure 4.22: Block-diagrams of: (a) Multiply-add unit. (b) Multiply-accumulate unit.
Saturating Multiply

If result is larger than $n$ for $n$-bit ops, pin the output at $2^{n-1}$ (saturate at max $n$-bit answer)

Or $2^{n-1} - 1$ and $-2^{n-1}$ for two’s comp

Truncating Multiplier

Typically used for multiplying fractions where you only want $n$ bits of the $2n$ bit result (most significant $n$ bits)
Squarer

Some simplification possible in the PP array because of symmetry of arguments.

(a)

(b)

Figure 4.25: Bit-array simplification in squaring of magnitudes ($n = 6$).

Constant Multiplier

- Pre-compute as much as you can...

$$45X = 5X \times 9 = X(2^3 + 1)(2^3 + 1)$$

```
X
  \downarrow SL2
  4X
  ADDER
5X
  \downarrow SL3
  40X
  ADDER
45X
```

SLK - shift left k positions
Summary

- **Sequential**
  - Higher radix reduces the number of iterations at the expense of longer cycles and more hardware (recoders, muxes)

- **Combinational**
  - Only one cycle, but can be a long cycle!
  - Really big

- **Hybrid**
  - Combine sequential and combinational…

Summary

- **For arrays**
  - Trees are good for large arrays because of reduction in number of levels
  - But area grows because of interconnect
  - This also has an impact on delay

- **Large multiplies can be expressed as interconnection of smaller multiplies**
  - Possibly useful for multiple precision units
  - Media processing?