Multiplication for 2’s Complement System – Booth Algorithm

- Consider an unsigned five bit number:
  \[ B = B_4B_3B_2B_1B_0 \]
  \[ = B_4 \times 16 + B_3 \times 8 + B_2 \times 4 + B_1 \times 2 + B_0 \times 1 \]
- For a 2’s complement number:
  \[ B_{\text{2's comp}} = B_4 \times (-16) + B_3 \times 8 + B_2 \times 4 + B_1 \times 2 + B_0 \times 1 \]
  which can be re-expressed as:
  \[ B_{\text{2's comp}} = (-16) \times B_4 + (16-8) \times B_3 + (8-4) \times B_2 + \]
  \[ (4-2) \times B_1 + (2-1) \times B_0 \]
  \[ = -16 \times (B_4 - B_3) - 8 \times (B_3 - B_2) - 4 \times (B_2 - B_1) - \]
  \[ 2 \times (B_1 - B_0) - 1 \times (B_0 - 0) \]
  The value in parentheses is difference of two consecutive bits, which could be +1,0, or -1.

Example: Use the Booth’s algorithm recoding scheme to perform the multiplication:
\[ 25_{10} \times -19_{10} \]

0 1 1 0 0 1 -- A=25_{10} multiplicant
1 0 1 1 0 1 -- B=-19_{10} multiplier
B_5B_4B_3B_2B_1B_0

\[ P = A \times B = -32 \times (B_5 - B_4) \times A - 16 \times (B_4 - B_3) \times A - \]
\[ 8 \times (B_3 - B_2) \times A - 4 \times (B_2 - B_1) \times A - \]
\[ 2 \times (B_1 - B_0) \times A - 1 \times (B_0 - 0) \times A \]
\[ = -32 \times A + 16 \times A + 0 \times A - 4 \times A + 2 \times A - 1 \times A \]
The logic for ALU Select lines is implemented by two NAND gates.
Multiplication

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
\times & B_3 & B_2 & B_1 & B_0 \\
\hline
R_{0,3} & R_{0,2} & R_{0,1} & R_{0,0} \\
R_{1,3} & R_{1,2} & R_{1,1} & R_{1,0} \\
R_{2,3} & R_{2,2} & R_{2,1} & R_{2,0} \\
R_{3,3} & R_{3,2} & R_{3,1} & R_{3,0} \\
\end{array}
\]

Sum of partial products

Using adders to add rows
Multiplication Using Adders

\[
A_3 A_2 A_1 A_0 \\
\times \quad B_3 B_2 B_1 B_0
\]

\[
\begin{align*}
R_{0,3} & \quad R_{0,2} & \quad R_{0,1} & \quad R_{0,0} \\
R_{1,3} & \quad R_{1,2} & \quad R_{1,1} & \quad R_{1,0} \\
R_{2,3} & \quad R_{2,2} & \quad R_{2,1} & \quad R_{2,0} \\
R_{3,3} & \quad R_{3,2} & \quad R_{3,1} & \quad R_{3,0}
\end{align*}
\]

Sum of partial products

1st level adder

Multiplication Using Adders

\[
A_3 A_2 A_1 A_0 \\
\times \quad B_3 B_2 B_1 B_0
\]

\[
\begin{align*}
R_{0,3} & \quad R_{0,2} & \quad R_{0,1} & \quad R_{0,0} \\
R_{1,3} & \quad R_{1,2} & \quad R_{1,1} & \quad R_{1,0} \\
R_{2,3} & \quad R_{2,2} & \quad R_{2,1} & \quad R_{2,0} \\
R_{3,3} & \quad R_{3,2} & \quad R_{3,1} & \quad R_{3,0}
\end{align*}
\]

Sum of partial products

2nd level adder
**Multiplication Using Adders**

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
\times & B_3 & B_2 & B_1 & B_0 \\
\hline
R_{0,3} & R_{0,2} & R_{0,1} & R_{0,0} \\
R_{1,3} & R_{1,2} & R_{1,1} & R_{1,0} \\
R_{2,3} & R_{2,2} & R_{2,1} & R_{2,0} \\
R_{3,3} & R_{3,2} & R_{3,1} & R_{3,0} \\
\end{array}
\]

3rd level adder

Sum of partial products

---

**Row Reduction Method for Multiplication**

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
\times & B_3 & B_2 & B_1 & B_0 \\
\hline
R_{0,3} & R_{0,2} & R_{0,1} & R_{0,0} \\
R_{1,3} & R_{1,2} & R_{1,1} & R_{1,0} \\
R_{2,3} & R_{2,2} & R_{2,1} & R_{2,0} \\
R_{3,3} & R_{3,2} & R_{3,1} & R_{3,0} \\
\end{array}
\]

Sum of partial products
Using Carry Save Adders

Row Reduction Method for Multiplication

\[
\begin{array}{c}
A_3 \ A_2 \ A_1 \ A_0 \\
\times \quad B_3 \ B_2 \ B_1 \ B_0 \\
\hline
R_{3,3} \ R_{3,2} \ R_{3,1} \ R_{3,0} \\
R_{2,3} \ R_{2,2} \ R_{2,1} \ R_{2,0} \\
R_{1,3} \ R_{1,2} \ R_{1,1} \ R_{1,0} \\
R_{0,3} \ R_{0,2} \ R_{0,1} \ R_{0,0} \\
\{ \text{1st level adder} \} \\
\{ \text{(row-reduction unit)} \} \\
\hline
\text{Sum of partial products}
\end{array}
\]
Row Reduction Method for Multiplication

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
\times & B_3 & B_2 & B_1 & B_0 \\
\hline
R_{0,3} & R_{0,2} & R_{0,1} & R_{0,0} \\
R_{1,3} & R_{1,2} & R_{1,1} & R_{1,0} \\
R_{2,3} & R_{2,2} & R_{2,1} & R_{2,0} \\
F_5 & F_4 & F_3 & F_2 & F_1 & F_0 \\
C_5 & C_4 & C_3 & C_2 & C_1 & C_0
\end{array}
\]

1st level adder (row-reduction unit)

\[
\begin{array}{cccc}
R_{3,3} \\
F_6 & F_5 & F_4 & F_3 & F_2 & F_1 & F_0 \\
C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
\end{array}
\]

2nd level adder (row-reduction unit)

Use a regular adder to add these two rows
Generalized row reduction method:

- Multipland: Multiplier

Partial Product Array

Intermediate Partial Product Sums

Formed in Parallel then Summed

Example: design a high speed multiplier
- $56 \times 56$ bit
- longest available row reduction unit: 15-4
- the final stage is a LACA with 8-bit basic adders.
**Multiplication with Sectioning**

- Design an 8×8 multiplier using 4×4 multipliers.

\[
\begin{array}{cccc}
X_3 & X_2 & X_1 & X_0 \\
\times & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
R_{0,3} & R_{0,2} & R_{0,1} & R_{0,0} \\
R_{1,3} & R_{1,2} & R_{1,1} & R_{1,0} \\
R_{2,3} & R_{2,2} & R_{2,1} & R_{2,0} \\
R_{3,3} & R_{3,2} & R_{3,1} & R_{3,0} \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
\]
\[ \begin{array}{cccccccc} X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\ \times \ & Y_7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 \end{array} \]

\[ \begin{array}{cccccccc} R_{0,7} & R_{0,6} & R_{0,5} & R_{0,4} & R_{0,3} & R_{0,2} & R_{0,1} & R_{0,0} \\ \end{array} \]

\[ \begin{array}{cccccccc} R_{1,7} & R_{1,6} & R_{1,5} & R_{1,4} & R_{1,3} & R_{1,2} & R_{1,1} & R_{1,0} \\ \end{array} \]

\[ P_1 = X_{3,0} \times Y_{3,0} \]

\[ P_2 = X_{2,4} \times Y_{3,0} \]
Division

\[ D_D = Q \times D_S + R \]

- dividend
- quotient
- divisor
- remainder

- The most straightforward method is to mimic the operations of paper-and-pencil long division for positive numbers.

Example:

```
   1011  Quotient, Q
  __________  ← dividend, D_D
          111010  Q_3\times D_s
           101
        __________  R\times D_s, continue
          10010  Q_2\times D_s, shifted
            000
        __________  R\times D_s, continue
          10010  Q_1\times D_s, shifted
            101
        __________  R\times D_s, continue
          1000  Q_0\times D_s, shifted
            101
        __________  R\times D_s, done
            11
```
A block diagram for such a divider:

- The division process involves repetitive shifts and subtraction operations.
Fig 6.10 Parallel Array Divider

R := (c → D: ¬c → (D-d-bi) mod 2):

Borrow always computed

Division by Repeated Multiplication

- Cost-effective if system contains high-speed multiplier
- Q = D_D / D_S
  - In each iteration, a factor f_i is generated & used to multiply both divisor D_S and dividend D_D.
    - Q = (D_D × f_0 × f_1 × f_2 ...)/(D_S × f_0 × f_1 × f_2 ...)
    - f_i is so chosen that D_S × f_0 × f_1 × f_2 ... converges rapidly toward 1.
    - If the denominator converges toward 1, the numerator converges toward Q.
For simplicity, assume $D_D$ & $D_S$ are positive normalized fraction: $D_S=1-x$ where $x<1$.

- Set $f_0 = 1+x$
  
  $\Rightarrow D_S \times f_0 = 1-x^2$ (closer to 1 than $D_S$)
  $\Rightarrow Q = \frac{(D_D \times (1+x))}{(1-x^2)}$

- Set $f_1 = 1+x^2$

  $\Rightarrow D_S \times f_0 \times f_1 = 1-x^4$ (even closer to 1)
  $\Rightarrow Q = \frac{(D_D \times (1+x) \times (1+x^2))}{(1-x^4)}$

- $f_0 = 1+x = 1+(1-D_S) = 2-D_S$ (2's complement of $D_S$)
- $f_1 = 1+x^2 = 1+(1-D_S \times f_0) = 2 \times D_S \times f_0 = 2-D_S f_0$

  $\Rightarrow f_i = 2-D_S \times f_0 \times \ldots f_{i-1} = 2-D_S (i-1)$

Example: (1). $0.4/0.7$:

<table>
<thead>
<tr>
<th>$D_D$</th>
<th>$0.4000000$</th>
<th>$D_S$</th>
<th>$0.7000000$</th>
<th>$f_0$</th>
<th>$1.3000000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_D$1</td>
<td>$0.5200000$</td>
<td>$D_S$1</td>
<td>$0.9099999$</td>
<td>$f_1$</td>
<td>$1.0900000$</td>
</tr>
<tr>
<td>$D_D$2</td>
<td>$0.5668000$</td>
<td>$D_S$2</td>
<td>$0.9918999$</td>
<td>$f_2$</td>
<td>$1.0081000$</td>
</tr>
<tr>
<td>$D_D$3</td>
<td>$0.5713911$</td>
<td>$D_S$3</td>
<td>$0.9999344$</td>
<td>$f_3$</td>
<td>$1.0000656$</td>
</tr>
<tr>
<td>$D_D$4</td>
<td>$0.5714286$</td>
<td>$D_S$4</td>
<td>$0.9999999$</td>
<td>$f_4$</td>
<td>$1.0000000$</td>
</tr>
<tr>
<td>$D_D$5</td>
<td>$0.5714286$</td>
<td>$D_S$5</td>
<td>$1.0000000$</td>
<td>$f_5$</td>
<td>$1.0000000$</td>
</tr>
<tr>
<td>$D_D$6</td>
<td>$0.5714286$</td>
<td>$D_S$6</td>
<td>$1.0000000$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(2). 0.7/0.4:

\[
\begin{align*}
D_{D_0} & = 0.7000000 & D_{S_0} & = 0.4000000 & f_0 & = 1.5999999 \\
D_{D_1} & = 1.1999999 & D_{S_1} & = 0.6400000 & f_1 & = 1.3599999 \\
D_{D_2} & = 1.5231999 & D_{S_2} & = 0.8704000 & f_2 & = 1.1295999 \\
D_{D_3} & = 1.7206066 & D_{S_3} & = 0.9832038 & f_3 & = 1.0002821 \\
D_{D_4} & = 1.7495062 & D_{S_4} & = 0.9997178 & f_4 & = 1.0002821 \\
D_{D_5} & = 1.7499998 & D_{S_5} & = 0.9999999 & f_5 & = 1.0000001 \\
D_{D_6} & = 1.7499999 & D_{S_6} & = 1.0000000
\end{align*}
\]

(3). 0.1/0.15:

\[
\begin{align*}
D_{D_0} & = 0.1000000 & D_{S_0} & = 0.1500000 & f_0 & = 1.8499999 \\
D_{D_1} & = 0.1850000 & D_{S_1} & = 0.2775000 & f_1 & = 1.7224999 \\
D_{D_2} & = 0.3186625 & D_{S_2} & = 0.4779938 & f_2 & = 1.5220062 \\
D_{D_3} & = 0.4850063 & D_{S_3} & = 0.7275094 & f_3 & = 1.2724905 \\
D_{D_4} & = 0.6171659 & D_{S_4} & = 0.9257489 & f_4 & = 1.0742511 \\
D_{D_5} & = 0.6629912 & D_{S_5} & = 0.9944868 & f_5 & = 1.0055132 \\
D_{D_6} & = 0.6666464 & D_{S_6} & = 0.9999696
\end{align*}
\]

- The # of iterations required is determined by the value of \(D_S\).
- It's better to use a fixed # of iterations
  - To assure that the process converges to the correct answer for all data, instead of using 2-\(D_S\) to calculate \(f_0\), use a ROM to find an appropriate value for \(f_0\).
  - It can then guarantee correct results after a fixed # of iterations.
Suppose ROM has $2^8$ words

(a) If $D_S$ is 8-bit, one iteration is sufficient
   \[ f_0 = 1/D_S \]

(b) If $D_S$ is > 8-bit, more than one iteration is required, $D_S \cdot f_0 = 1-x \& x < 2^{-8}$
   - At the 2nd iteration, $D_S \cdot f_0 \cdot f_1 = 1-x^2$
     - the difference from 1 is $< 2^{-16}$
   - At the $i$th iteration ($i > 2$)
     \[ D_S \cdot f_0 \cdot f_1 \cdot \ldots \cdot f_{i-1} = D_{S_{i-1}} = 1-x^{2(i-1)} \]
     - the difference from 1 is $< (2^{-8})^{2(i-1)}$

(3rd iteration error $< 2^{-32}$)
(4th iteration error $< 2^{-64}$)
Fig 6.14 Floating-Point Number Format

- $s$ is sign, $e$ is exponent, and $f$ is significand (mantissa)
- We will assume a fraction mantissa, but some representations have used integers

Floating Point Arithmetic

Floating point addition

- The difficulty when adding two floating point numbers stems from the fact that the mantissas, in general, have different significance.

$$A = B + C$$

$$= M_B \times r_s^{E_B} + M_C \times r_s^{E_C}$$

- Before the two numbers can be properly added together, the mantissas must be aligned.

$$A = (M_B \times r_s^{E_B-E_C} + M_C) \times r_s^{E_C}$$ (assume $|B| < |C|$)
- This involves determining which operand value is smaller, and then aligning the mantissa of that operand appropriately with the mantissa of the larger operand.
- The alignment is accomplished by shifting the mantissa of the smaller operand to line up with the digits of the same significance in the larger operand.
- The amount of the alignment, i.e. the # of positions to shift, is determined by the difference in the exponents.
The selection of the appropriate mantissa to be aligned is made based on a comparison of the magnitude of the two exponents.

The resulting number of the addition/subtraction is provided to the “Post-Normalization” unit.

Examples:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input A</td>
<td>Input B</td>
<td>Result</td>
</tr>
<tr>
<td>0.8045</td>
<td>0.7133</td>
<td>1.5177</td>
</tr>
</tbody>
</table>

0.8045 + 0.7133 = 1.5177

0.8045 - 0.8032 = 0.0012

The post normalization unit must be capable of shifts of one or more positions for the mantissa and adjust the size of the exponent to reflect the normalization.

Floating point addition, then, requires many more operations, and hence more hardware, than its integer counterpart.
Design a network to align the smaller mantissa to be added to the larger mantissa.

- Assume that the mantissa is 24 bits
- The alignment network must be capable of shifting any number of bits, from 0 to 24 (shift left).
- Assume the adders used to compare the exponents provide a binary number (size: 0 to 24; hence 5 bits $S_4S_3S_2S_1S_0$) which indicates how far the number needs to be shifted in the alignment process.
Fig 6.11  A \( N \times N \) Bit Crossbar Design for Barrel Rotator

Properties of the Crossbar Barrel Shifter

- There is a 2-gate delay for any length shift
- Each output line is effectively an \( n \) way multiplexer for shifts of up to \( n \) bits
- There are \( n^2 \) 3-state drivers for an \( n \) bit shifter
  - For \( n = 32 \), this means 1024 3-state drivers
- For 32 bits, the decoder is 5 bits (1 out of 32)
- The minimum delay but large number of gates in the crossbar prompts a compromise:
  - the logarithmic barrel shifter
The LSB of this number is used by the first level of MUXs to shift the number by 1 bit (the 1 condition), or provide no shift at all (the 0 condition).

Similarly, the second LSB controls the 2nd set of MUXs to shift the number by 2 more or not to shift.

Similarly, the MSB controls the 5th set of MUXs to shift the number by 16 more or not to shift.
**Floating Point Multiplication**

- $A = B \times C$
  
  $= M_B \times r_S^{EB} \times M_C \times r_S^{Ec}$
  
  $= M_B \times M_C \times r_S^{EB+Ec}$

---

**Post normalization unit only needs to shift the result by at most one bit position.**

- Consider two extreme cases:

  **Largest / Largest**
  
<table>
<thead>
<tr>
<th>Base 2</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1111</td>
<td>0.9999</td>
</tr>
<tr>
<td>$\times$ 0.1111</td>
<td>$\times$ 0.9999</td>
</tr>
<tr>
<td>0.1110</td>
<td>0.9998</td>
</tr>
</tbody>
</table>

  Aligned properly, $\Rightarrow$ no postnormalization.

  **Smallest / smallest**
  
<table>
<thead>
<tr>
<th>Base 2</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1000</td>
<td>0.1000</td>
</tr>
<tr>
<td>$\times$ 0.1000</td>
<td>$\times$ 0.1000</td>
</tr>
<tr>
<td>0.0100</td>
<td>0.0100</td>
</tr>
</tbody>
</table>

  Not aligned properly, $\Rightarrow$ postnormalization of one digit position.
Floating Point division

\[ A = B / C \]

\[ = M_B \times r_s^{E_B} / (M_C \times r_s^{E_C}) \]

\[ = (M_B / M_C) \times r_s^{E_B - E_C} \]

Exponent B Exponent C Mantissa B Mantissa C
Exponent subtract Divide Exponent Adjust Post-Normalization

The result of the mantissa division may require post-normalization by at most one bit position in opposite direction of the multiplier.