Many different circuits exist for multiplication. Each one has a different balance between speed (performance) and amount of logic (cost).
“Shift and Add” Multiplier

- Sums each partial product, one at a time.
- In binary, each partial product is shifted versions of A or 0.

Control Algorithm:
1. \( P \leftarrow 0, A \leftarrow \) multiplicand,
   \( B \leftarrow \) multiplier
2. If LSB of B==1 then add A to P else add 0
3. Shift \([P][B]\) right 1
4. Repeat steps 2 and 3 \( n-1 \) times.
5. \([P][B]\) has product.

- Cost \( \alpha n, T = n \) clock cycles.
- What is the critical path for determining the min clock period?
“Shift and Add” Multiplier

Signed Multiplication:

*Remember* for 2’s complement numbers MSB has negative weight:

\[ X = \sum_{i=0}^{N-2} x_i 2^i - x_{n-1} 2^{n-1} \]

ex: \(-6 = 11010_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 - 1 \cdot 2^4 \)
\[
= 0 + 2 + 0 + 8 - 16 = -6
\]

- Therefore for multiplication:
  a) subtract final partial product
  b) sign-extend partial products
- Modifications to shift & add circuit:
  a) adder/subtractor
  b) sign-extender on P shifter register
Outline

- Combinational multiplier
- Latency & Throughput
  - Wallace Tree
  - Pipelining to increase throughput
- Smaller multipliers
  - Booth encoding
  - Serial, bit-serial
- Two’s complement multiplier
Unsigned Combinational Multiplier
Array Multiplier

Single cycle multiply: Generates all $n$ partial products simultaneously.

What is the critical path?
Combinational Multiplier (unsigned)

\[
\begin{array}{cccccc}
X3 & X2 & X1 & X0 \\
* & Y3 & Y2 & Y1 & Y0 \\
\hline
X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & X3Y1 & X2Y1 & X1Y1 & X0Y1 \\
+ & X3Y2 & X2Y2 & X1Y2 & X0Y2 \\
+ & X3Y3 & X2Y3 & X1Y3 & X0Y3 \\
\hline
Z7 & Z6 & Z5 & Z4 & Z3 & Z2 & Z1 & Z0 \\
\end{array}
\]

- **Multiplicand**
- **Multiplier**

Partial products, one for each bit in multiplier (each bit needs just one AND gate)

Propagation delay \( \approx 2N \)
Carry-Save Addition

- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- “Carry-save” addition can help.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.

Example: sum three numbers, \(3_{10} = 0011, 2_{10} = 0010, 3_{10} = 0011\)

\[
\begin{align*}
3_{10} & \quad 0011 \\
+ & \quad 2_{10} \quad 0010 \\
\text{c} & \quad 0100 = 4_{10} \\
\text{s} & \quad 0001 = 1_{10}
\end{align*}
\]

- In general, carry-save addition takes in 3 numbers and produces 2.
- Whereas, carry-propagate takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition.
Carry-save Circuits

- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.
- Carry-save is fast (no carry propagation) and cheap (same cost as ripple adder)
Array Multiplier using Carry-save Addition

Fast carry-propagate adder
Carry-save Addition

CSA is associative and communitive. For example:

\[((X_0 + X_1) + X_2) + X_3\] = \[(X_0 + X_1) + (X_2 + X_3)\]

• A balanced tree can be used to reduce the logic delay.

• This structure is the basis of the **Wallace Tree Multiplier**.

• Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final sum.

• Multiplier delay \(\alpha \log_{3/2} N + \log_2 N\)
Increasing Throughput: Pipelining

Idea: split processing across several clock cycles by dividing circuit into pipeline stages separated by registers that hold values passing from one stage to the next.

Throughput = $\frac{1}{4t_{PD,FA}}$ instead of $\frac{1}{8t_{PD,FA}}$
Smaller Combinational Multipliers

Idea: If we could use, say, 2 bits of the multiplier in generating each partial product we would halve the number of columns and halve the latency of the multiplier!

Booth’s insight: rewrite 2*A and 3*A cases, leave 4A for next partial product to do!
Booth recoding

(On-the-fly canonical signed digit encoding!)

<table>
<thead>
<tr>
<th>$B_{K+1}$</th>
<th>$B_K$</th>
<th>$B_{K-1}$</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>add 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>add $A$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>add $A$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>add $2A$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sub $2A$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sub $A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sub $A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>add 0</td>
</tr>
</tbody>
</table>

$A$ "1" in this bit means the previous stage needed to add $4A$. Since this stage is shifted by 2 bits with respect to the previous stage, adding $4A$ in the previous stage is like adding $A$ in this stage!
Bit-serial Multiplier

- Bit-serial multiplier \( (n^2 \text{ cycles, one bit of result per } n \text{ cycles}) \):

- Control Algorithm:

```plaintext
repeat n cycles { // outer (i) loop
  repeat n cycles { // inner (j) loop
    shiftA, selectSum, shiftHI
  }
  shiftB, shiftHI, shiftLOW, reset
}
```

Note: The occurrence of a control signal \( x \) means \( x=1 \). The absence of \( x \) means \( x=0 \).
Signed Multipliers
Combinational Multiplier (signed!)

\((-3) \times (-2)\)

\((-3)\) \hspace{1cm} \begin{array}{ccc} 1 & 0 & 1 \end{array} \hspace{1cm} (X)

\((-2)\) \hspace{1cm} \begin{array}{ccc} 1 & 1 & 0 \end{array} \hspace{1cm} (Y)

\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
+ & 1 & 1 & 1 & 0 & 1 \\
- & 1 & 1 & 0 & 1 & \\
\hline
0 & 0 & 0 & 1 & 1 & 0
\end{array}

\(Y_0 \times X = 0\)

\(2Y_1 \times X = -6\)

\(4Y_2 \times X = -12\)

\(+6\)
Combinational Multiplier (signed)

\[
\begin{array}{cccc}
X_3 & X_2 & X_1 & X_0 \\
* & Y_3 & Y_2 & Y_1 & Y_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 \\
X_2Y_0 & X_1Y_0 & X_0Y_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 \\
X_2Y_1 & X_1Y_1 & X_0Y_1 \\
\end{array}
\]

\[
\begin{array}{cccc}
X_3Y_2 & X_3Y_2 & X_3Y_2 & X_3Y_2 \\
X_2Y_2 & X_1Y_2 & X_0Y_2 \\
\end{array}
\]

\[
\begin{array}{cccc}
X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 \\
X_2Y_3 & X_1Y_3 & X_0Y_3 \\
\end{array}
\]

\[
\begin{array}{cccc}
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0 \\
\end{array}
\]

There are tricks we can use to eliminate the extra circuitry we added...
2’s Complement Multiplication
(Baugh-Wooley)

Step 1: two’s complement operands so high order bit is $-2^{N-1}$. Must sign extend partial products and subtract the last one

$$
\begin{array}{c}
X_3 & X_2 & X_1 & X_0 \\
* & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
X_3 Y_0 & X_3 Y_0 & X_3 Y_0 & X_3 Y_0 \\
X_2 Y_0 & X_1 Y_0 & X_0 Y_0 \\
\hline
X_3 Y_1 & X_3 Y_1 & X_3 Y_1 & X_3 Y_1 \\
X_2 Y_1 & X_1 Y_1 & X_0 Y_1 \\
\hline
X_2 Y_2 & X_2 Y_2 & X_2 Y_2 & X_2 Y_2 \\
X_1 Y_2 & X_1 Y_2 & X_0 Y_2 \\
\hline
-X_3 Y_3 & X_3 Y_3 & X_2 Y_3 & X_1 Y_3 & X_0 Y_3
\end{array}
$$

Step 2: don’t want all those extra additions, so add a carefully chosen constant, remembering to subtract it at the end. Convert subtraction into add of (complement + 1).

$$
\begin{array}{c}
X_3 Y_0 & X_3 Y_0 & X_3 Y_0 & X_3 Y_0 & X_3 Y_0 & X_2 Y_0 & X_1 Y_0 & X_0 Y_0 \\
+ & & & & & & & 1 \\
X_3 Y_1 & X_3 Y_1 & X_3 Y_1 & X_3 Y_1 & X_2 Y_1 & X_1 Y_1 & X_0 Y_1 \\
+ & & & & & & & 1 \\
X_2 Y_2 & X_2 Y_2 & X_2 Y_2 & X_2 Y_2 & X_1 Y_2 & X_0 Y_2 \\
+ & & & & & & & 1 \\
-X_3 Y_3 & X_3 Y_3 & X_2 Y_3 & X_1 Y_3 & X_0 Y_3 \\
+ & & & & & & & 1 \\
\hline
\end{array}
$$

$$-B = \sim B + 1$$

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

$$
\begin{array}{c}
X_3 Y_0 & X_2 Y_0 & X_1 Y_0 & X_0 Y_0 \\
+ & \sim X_3 Y_1 & X_2 Y_1 & X_1 Y_1 & X_0 Y_1 \\
+ & X_2 Y_2 & X_1 Y_2 & X_0 Y_2 \\
+ & X_3 Y_3 & X_2 Y_3 & X_1 Y_3 & X_0 Y_3 \\
+ & & & & & & & 1 \\
- & & & & & & & 1 & 1 & 1 & 1
\end{array}
$$

Step 4: finish computing the constants...

$$
\begin{array}{c}
X_3 Y_0 & X_2 Y_0 & X_1 Y_0 & X_0 Y_0 \\
+ & \sim X_3 Y_1 & X_2 Y_1 & X_1 Y_1 & X_0 Y_1 \\
+ & X_2 Y_2 & X_1 Y_2 & X_0 Y_2 \\
+ & X_3 Y_3 & X_2 Y_3 & X_1 Y_3 & X_0 Y_3 \\
+ & & & & & & & 1 \\
+ & 1 & 1 & 1 & 1
\end{array}
$$

Result: multiplying 2’s complement operands takes just about same amount of hardware as multiplying unsigned operands!
2’s Complement Multiplication

\[
\begin{array}{cccc}
\overline{x_3} & x_2 & x_1 & x_0 \\
+ & \overline{x_3} & x_2 & x_1 & x_0 \\
+ & x_2 & x_1 & x_0 & x_0 \\
+ & 1 & & & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
x_3 & x_2 & x_1 & x_0 \\
y_3 & y_2 & y_1 & y_0 \\
z_7 & z_6 & z_5 & z_4 \\
z_0 & & &
\end{array}
\]
**Multiplication in Verilog**

You can use the “*” operator to multiply two numbers:

```verilog
wire [9:0] a, b;
wire [19:0] result = a*b;  // unsigned multiplication!
```

If you want Verilog to treat your operands as signed two’s complement numbers, add the keyword `signed` to your `wire` or `reg` declaration:

```verilog
wire signed [9:0] a, b;
wire signed [19:0] result = a*b;  // signed multiplication!
```

Remember: unlike addition and subtraction, you need different circuitry if your multiplication operands are signed vs. unsigned. Same is true of the `>>>` (arithmetic right shift) operator. To get signed operations all operands must be signed.

```verilog
wire signed [9:0] a;
wire [9:0] b;
wire signed [19:0] result = a*$signed(b);
```

To make a signed constant: 10’sh37C