Lecture 21: Multiplier Circuits
Warmup

• Recall long multiplication of base-10 by hand:

\[ \begin{array}{c}
12 \\
\times \quad 56 \\
\hline
56 \\
+ 120 \\
\hline
672
\end{array} \]

• In base-2 (binary), we do the same thing:

\[ \begin{array}{c}
011 \\
\times \quad 101 \\
\hline
101 \\
+ 000 \\
\hline
1101
\end{array} \]
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. \( P \leftarrow 0, A \leftarrow \text{multiplicand}, B \leftarrow \text{multiplier} \)
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 \(n-1\) more 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
Convince yourself

• What’s -3 x 5?

\[
\begin{array}{c}
1101 \\
\times \ 0101 \\
\hline
\end{array}
\]
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)

**Multiplicand**

\[ X_3 \ X_2 \ X_1 \ X_0 \]

**Multiplier**

\[ Y_3 \ Y_2 \ Y_1 \ Y_0 \]

**Partial Products**

\[ \begin{array}{c}
X_3Y_0 \ X_2Y_0 \ X_1Y_0 \ X_0Y_0 \\
+ \ X_3Y_1 \ X_2Y_1 \ X_1Y_1 \ X_0Y_1 \\
+ \ X_3Y_2 \ X_2Y_2 \ X_1Y_2 \ X_0Y_2 \\
+ \ X_3Y_3 \ X_2Y_3 \ X_1Y_3 \ X_0Y_3 \\
\end{array} \]

**Output**

\[ Z_7 \ Z_6 \ Z_5 \ Z_4 \ Z_3 \ Z_2 \ Z_1 \ Z_0 \]

**Propagation delay** \( \sim 2N \)
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,

\[ 3_{10} = 0011, \ 2_{10} = 0010, \ 3_{10} = 0011 \]

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

\[
\begin{align*}
    3_{10} & \quad 0011 \\
    + & \quad 2_{10} \quad 0010 \\
    c & \quad 0010 \quad = \quad 2_{10} \\
    s & \quad 0110 \quad = \quad 6_{10} \\
    & \quad 1000 \quad = \quad 8_{10}
\end{align*}
\]

• In general, carry-save addition takes in 3 numbers and produces 2.
  • Sometimes called a “3:2 compressor”: 3 input signals into 2 in a potentially lossy operation
  • 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
Array Multiplier Again

Each row: n-bit adder with AND gates

Fast carry-propagate adder

What is the critical path?
Carry-save Addition

CSA is associative and commutative. 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.
- It doesn’t matter where you add the carries and sums, as long as you eventually do add them.
- This structure is the basis of the \textit{Wallace Tree Multiplier}.
- Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final sum.
- Multiplier delay $\alpha \log_{3/2}N + \log_2N$
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 = $1/4t_{PD,FA}$ instead of $1/8t_{PD,FA}$
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 2*A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sub 2*A</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>

$B_{K+1,K}*A = 0*A \rightarrow 0$

= 1*A → A

= 2*A → 4A - 2A

= 3*A → 4A - A

A “1” in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!
Example

Shifted left

(A) 0111
(B) x 1010

(10[0] sub 2A) -01110
(101 sub A) -0111
(001 add A) +0111

01000110
Bit-serial Multiplier

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

```
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!)

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

\((-3)\)
\[\begin{array}{cccc}
1 & 0 & 1 & \text{(X)} \\
\end{array}\]
\((-2)\)
\[\begin{array}{cccc}
1 & 1 & 0 & \text{(Y)} \\
\end{array}\]

\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
+ & 1 & 1 & 1 \\
- & 1 & 1 & 0 \\
\end{array}\]

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

Range: \(-2^{N-1}\) to \(2^{N-1} - 1\)

"sign bit"   "decimal" point

\(N\) bits
Combinational Multiplier (signed)

\[
\begin{array}{cccc}
X3 & X2 & X1 & X0 \\
* & Y3 & Y2 & Y1 & Y0 \\
\end{array}
\]

\[
\begin{array}{cccc}
X3Y0 & X3Y0 & X3Y0 & X3Y0 \\
+ & X3Y1 & X3Y1 & X3Y1 & X3Y1 \\
+ & X3Y2 & X3Y2 & X3Y2 & X3Y2 \\
- & X3Y3 & X3Y3 & X3Y3 & X3Y3 \\
\end{array}
\]

\[
\begin{array}{cccccc}
& & & & Z7 & Z6 & Z5 & Z4 & Z3 & Z2 & Z1 & Z0 \\
\end{array}
\]

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

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

\[
\begin{array}{ccccccc}
X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
+ X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ X_3Y_2 & X_3Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
- X_3Y_3 & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3
\end{array}
\]

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

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).

\[
\begin{array}{cccccccc}
X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
+ & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
+ & X_3Y_3 & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3
\end{array}
\]

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

\[
-B = \overline{B} + 1
\]

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

\[
\begin{array}{cccccccc}
X_3Y_0 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
+ & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 \\
+ & & & & & & & & & 1
\end{array}
\]

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

Step 4: finish computing the constants...

\[
\begin{array}{cccccccc}
X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
+ & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
+ & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 \\
+ & & & & & & & & & 1
\end{array}
\]

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

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

\[ \overline{x3y0} \ x2y0 \ x1y0 \ x0y0 \]
\[ + \overline{x3y1} \ x2y1 \ x1y1 \ x0y1 \]
\[ + \overline{x2y2} \ x1y2 \ x0y2 \]
\[ + \overline{x3y3} \ x2y3 \ x1y3 \ x0y3 \]
\[ + 1 \]

\[ \overline{y0} \]
\[ \overline{y1} \]
\[ \overline{z0} \]
\[ \overline{z1} \]
\[ \overline{z2} \]
\[ \overline{z3} \]
\[ \overline{z4} \]
\[ \overline{z5} \]
\[ \overline{z6} \]
\[ \overline{z7} \]
Example

• What’s -3 x -5?

\[
\begin{array}{c}
1101 \\
\times 1011
\end{array}
\]
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;
wirer [9:0] b;
wire signed [19:0] result = a * $signed(b);
```

To make a signed constant: `10’sh37C`