Lecture 24: Multiplier Circuits and Shifters
Announcements
Warmup

• Recall long multiplication of base-10 by hand:

\[
\begin{array}{c}
12 \\
\times 56
\end{array}
\]

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

\[
\begin{array}{c}
011 \\
\times 101
\end{array}
\]
Many different circuits exist for multiplication. Each one has a different balance between speed (performance) and amount of logic (cost).
“Shift and Add” Multiplier

Control Algorithm:
1. \( P \leftarrow 0, A \leftarrow \text{multiplicand}, \quad B \leftarrow \text{multiplier} \)
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.

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

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**:

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

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?

\[
\begin{array}{c}
1101 \\
x 0101 \\
\end{array}
\]
Outline for Multipliers

- 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.

Each row: \( n \)-bit adder with AND gates

What is the critical path?
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,

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

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

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

carry-save add

carry-propagate add

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

Any fast carry-propagate adder
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 **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.

= register
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 rows and halve the latency of the multiplier!

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

<table>
<thead>
<tr>
<th>$B_{K+1}$</th>
<th>$B_K$</th>
<th>$B_{K-1}$</th>
<th>action</th>
<th>$B_{K+1,K} \cdot A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>add 0</td>
<td>$0 \cdot A \rightarrow 0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>add A</td>
<td>$1 \cdot A \rightarrow A$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>add A</td>
<td>$2 \cdot A \rightarrow 4A - 2A$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>add 2*A</td>
<td>$3 \cdot A \rightarrow 4A - A$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sub 2*A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sub A</td>
<td>$-2A + A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sub A</td>
<td>$-A + A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>add 0</td>
<td>$-A + A$</td>
</tr>
</tbody>
</table>

A “1” in this bit means the previous stage needed to add $4 \cdot A$. Since this stage is shifted by 2 bits with respect to the previous stage, adding $4 \cdot A$ 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

$7_{10}$

$10_{10}$

$70_{10}$
Bit-serial Multiplier

- Bit-serial multiplier ($n^2$ cycles, one bit of result per $n$ cycles):

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

- Control Algorithm:

  ```
  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!)

\[ \text{X} \times \text{Y} = (-3) \times (-2) \]

\[
\begin{array}{cccc}
(-3) & & 1 & 0 & 1 & \text{(X)} \\
(-2) & \times & 1 & 1 & 0 & \text{(Y)} \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & \text{Y0} \times \text{X} = 0 \\
+ & 1 & 1 & 1 & 0 & 1 & 2\text{Y1} \times \text{X} = -6 \\
- & 1 & 1 & 0 & 1 & \quad & 4\text{Y2} \times \text{X} = -12 \\
\hline
( +6 ) & & & & & 0 & 0 & 0 & 1 & 1 & 0
\end{array}
\]
Combinational Multiplier (signed)

\[
\begin{array}{cccc}
X_3 & X_2 & X_1 & X_0 \\
\times & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 \\
X_2Y_0 & X_1Y_0 & X_0Y_0 \\
+ & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 \\
+ & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ & X_3Y_2 & X_3Y_2 & X_3Y_2 \\
+ & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
- & X_3Y_3 & X_3Y_3 \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}
\]

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

\[
\begin{array}{cccc}
X_3 & X_2 & X_1 & X_0 \\
\times & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 \\
X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
\end{array}
\]

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

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

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

\[
\begin{array}{cccccc}
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0 \\
\hline
X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
& X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
& & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
& & & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 \\
& & & & & 1 & 1 & 1 & 1 \\
\end{array}
\]

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}{cccccc}
X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
+ & 1 & & & & & & \\
+ & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ & 1 & & & & & & \\
+ & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
+ & 1 & & & & & & \\
+ & X_3Y_3 & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 \\
+ & 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!

Step 4: finish computing the constants...

\[
\begin{array}{cccccc}
X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
+ & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
+ & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 \\
+ & 1 & & & & & 1 \\
\end{array}
\]

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

\[
\begin{array}{cccc}
\overline{x_3y_0} & x_2y_0 & x_1y_0 & x_0y_0 \\
+ & \overline{x_3y_1} & x_2y_1 & x_1y_1 & x_0y_1 \\
+ & x_3y_2 & x_2y_2 & x_1y_2 & x_0y_2 \\
+ & 1 & x_3y_3 & x_2y_3 & x_1y_3 & x_0y_3 \\
\end{array}
\]
Example

• What’s -3 x -5?

\[
\begin{array}{c}
1101 \\
\times 1011 \\
\end{array}
\]
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`
Outline

- Constant Coefficient Multiplication
- Shifters
- Counters
Constant Multiplication

- Our multiplier circuits so far has assumed both the multiplicand (A) and the multiplier (B) can vary at runtime.
- What if one of the two is a constant?
  \[ Y = C \times X \]
- “Constant Coefficient” multiplication comes up often in signal processing and other hardware. Ex:
  \[ y_i = \alpha y_{i-1} + x_i \]

  where \( \alpha \) is an application dependent constant that is hard-wired into the circuit.

- How do we build and array style (combinational) multiplier that takes advantage of the constancy of one of the operands?
Multiplication by a Constant

- If the constant C in C*X is a power of 2, then the multiplication is simply a shift of X.
- Ex: 4*X

What about division?

What about multiplication by non-powers of 2?
Multiplication by a Constant

- In general, a combination of fixed shifts and addition:
  - Ex: $6 \times X = 0110 \times X = (2^2 + 2^1) \times X = 2^2 \times X + 2^1 \times X$

- Details:
Another example: $C = 23_{10} = 010111$

- In general, the number of additions equals one less than the number of 1’s in the constant.
- Using carry-save adders (for all but one of these) helps reduce the delay and cost, and using balanced trees helps with delay, but the number of adders is still the number of 1’s in C minus 2.

- Is there a way to further reduce the number of adders (and thus the cost and delay)?
Multiplication using Subtraction

- Subtraction is approximately the same cost and delay as addition.
- Consider C*X where C is the constant value $15_{10} = 01111$. C*X requires 3 additions.
- We can “recode” 15

  \[
  01111 = (2^3 + 2^2 + 2^1 + 2^0)
  \]

  \[
  \to 10001 = (2^4 - 2^0)
  \]

  where 1 means negative weight.

- Therefore, 15*X can be implemented with only one subtractor.
Canonic Signed Digit Representation

- CSD represents numbers using 1,  \( \overline{1} \), & 0 with the least possible number of non-zero digits.
  - Strings of 2 or more non-zero digits are replaced.
  - Leads to a unique representation.

- To form CSD representation might take 2 passes:
  - First pass: replace all occurrences of 2 or more 1’s:
    \( 01..10 \) by \( 10..\overline{10} \)
  - Second pass: same as above, plus replace \( 01\overline{1}0 \) by \( 0010 \) and \( \overline{0}1\overline{1}0 \) by \( 00\overline{1}0 \)

- Examples:
  - \( 0010111 = 23 \)
  - \( 0011001 = 32 - 8 - 1 \)
  - \( 011101 = 29 \)
  - \( 001\overline{1}001 = 32 - 8 - 1 \)
  - \( 100\overline{1}01 = 32 - 4 + 1 \)
  - \( 1011010 = 64 - 8 - 2 \)
  - \( 100\overline{1}0\overline{1}0 = 32 - 8 - 1 \)
  - \( 0110110 = 54 \)
  - \( 01\overline{1}0111 = 23 \)
  - \( 01\overline{1}0001 = 32 - 8 - 1 \)
  - \( 10\overline{1}01\overline{1}0 = 32 - 8 - 1 \)

- Can we further simplify the multiplier circuits?
**“Constant Coefficient Multiplication” (KCM)**

Binary multiplier: $Y = 231 \times X = (2^7 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0) \times X$

- CSD helps, but the multipliers are limited to shifts followed by adds.
  - CSD multiplier: $Y = 231 \times X = (2^8 - 2^5 + 2^3 - 2^0) \times X$

- How about shift/add/shift/add…?
  - KCM multiplier: $Y = 231 \times X = 7 \times 33 \times X = (2^3 - 2^0) \times (2^5 + 2^0) \times X$

- No simple algorithm exists to determine the optimal KCM representation.
- Most use exhaustive search method.
Fixed Shifters / Rotators Defined

Logical Shift

Rotate

Arithmetic Shift
Variable Shifters / Rotators

- Example: $X \gg S$, where $S$ is unknown when we synthesize the circuit.
- Uses: shift instruction in processors (ARM includes a shift on every instruction), floating-point arithmetic, division/multiplication by powers of 2, etc.
- One way to build this is a simple shift-register:
  a) Load word,  b) shift enable for $S$ cycles,  c) read word.

- Worst case delay $O(N)$, not good for processor design.
- Can we do it in $O(\log N)$ time and fit it in one cycle?
Log Shifter / Rotator

- Log(N) stages, each shifts (or not) by a power of 2 places, $S=[s_2;s_1;s_0]$:

![Diagram of Log Shifter / Rotator]
LUT Mapping of Log shifter

Efficient with 2to1 multiplexors, for instance, 3LUTs.

Virtex6 has 6LUTs. Naturally makes 4to1 muxes:

Reorganize shifter to use 4to1 muxes.

Final stage uses F7 mux
“Improved” Shifter / Rotator

- How about this approach? Could it lead to even less delay?

- What is the delay of these big muxes?
- Look a transistor-level implementation?

Left-shift with rotate
Barrel Shifter

Cost/delay?
Connection Matrix

- Generally useful structure:
  - $N^2$ control points.
  - What other interesting functions can it do?
Cross-bar Switch

- Nlog(N) control signals.
- Supports all interesting permutations
  - All one-to-one and one-to-many connections.
- Commonly used in communication hardware (switches, routers).