EECS 427
Lecture 10: Multipliers
Reading: 11.4 (skip Tree Multiplier section, for now)

---

Last Time

- Full adder cells and adder topologies
  - Carry bypass and carry select offer decent speed gains over ripple carry and relatively low complexity
  - Carry lookahead based techniques are much faster for N>16, lots more area/complexity
  - Adder cells: mirror structure or transmission gates
Lecture Overview

- Multiplier implementations
- Multipliers are vital in digital signal processing and standard desktop processors
- They are speed limiting – very slow operations

Binary Multiplication

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 1 & 0 \\
\times & & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
+ & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]

- Multiplicand (M-bits)
- Multiplier (N-bits)
- Partial products
- Result
Key points

• NxM multiplication fits in N+M bits
• 2s complement multiplication is more difficult: Either convert to + numbers and keep track of original signs OR use Booth’s algorithm
• Major steps are:
  1) Partial product generation
  2) Partial product accumulation
  3) Final addition (done using fast carry lookahead techniques)

Generating Partial Products

• All partial products: AND

- Booth’s recoding – reduction of partial product count (more later)
The Array Multiplier

\[ Z = X \times Y \]

MxN Array Multiplier — Critical Path

\[ t_{\text{mult}} = [(M-1)+(N-2)]t_{\text{carry}} + (N-1) \ t_{\text{sum}} + t_{\text{and}} \]

Both carry and sum delays important: T-gate adder cell…
Carry-Save Multiplier

\[ t_{\text{mult}} = (N-1)t_{\text{carry}} + t_{\text{and}} + t_{\text{merge}} \]

Booth Recoding

- To implement 2s complement multiplication, modified Booth recoding is typically used
- Idea: Recode the multiplier value in a higher radix in order to reduce the # of partial products
- Ex: 010111 (23)
  011110 (30)
  1 3 2 690
Modified Booth Recoding

- Now we need to be able to multiply by 0, 1, 2, or 3
  - +3 is not easy to implement; requires two stages
    (+4X – 1X OR +2X + 1X)
- Instead look at three bits at a time and use negatives:
  - ±2X, ±1X, 0
  - Must be able to multiply by 0, 1, 2, -1, -2
  - 0 and 1 are easy, 2X involves a shift left by 1 bit position, -1X: invert all bits and set Cin = 1, -2X: invert all bits, carry in a 1, and shift left by 1 bit

Recoding table

- Instead of 3Y, use –Y, then increment next partial product to add 4Y
- Similarly, for 2Y, use –2Y, then increment next partial product to add 4Y

<table>
<thead>
<tr>
<th>(x_i \cdot x_{i+1} \cdot x_{i+2})</th>
<th>Add to partial product</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>+0Y</td>
</tr>
<tr>
<td>001</td>
<td>+1Y</td>
</tr>
<tr>
<td>010</td>
<td>+1Y</td>
</tr>
<tr>
<td>011</td>
<td>+2Y</td>
</tr>
<tr>
<td>100</td>
<td>-2Y</td>
</tr>
<tr>
<td>101</td>
<td>-1Y</td>
</tr>
<tr>
<td>110</td>
<td>-1Y</td>
</tr>
<tr>
<td>111</td>
<td>0Y</td>
</tr>
</tbody>
</table>
Example

- **010111** (23) Originally: 011110
  - 011 → +2
- **011110** (30)
  - 111 → 0
  - 100 → -2
  - LSB extends with 0s
  - So we have:
  - (+2)(0)\((-2)\)

Example, two 8-bit negative #'s

**10110010** = -78
**10011101** = -99

<table>
<thead>
<tr>
<th>Partial Product</th>
<th>Recode:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111110110010 010→ +1</td>
<td>10011101</td>
</tr>
<tr>
<td>00000001001110 110→ -1</td>
<td></td>
</tr>
<tr>
<td>111101100100 011→ +2</td>
<td></td>
</tr>
<tr>
<td>0010011100 100→ -2</td>
<td></td>
</tr>
<tr>
<td>10001111000101010 = 7722</td>
<td></td>
</tr>
</tbody>
</table>

Ignore carry out into 17th place
In-class example

\[
\begin{array}{r}
010111 \\
\times 011110
\end{array}
\]

Booth Decoding and Partial Product Generation

<table>
<thead>
<tr>
<th>Operation</th>
<th>NEG</th>
<th>ZERO</th>
<th>TWO</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x (-1)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x (-2)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Booth

Y_i

Y_{i+1}

Y_i

TWO

NEG

ZERO
Summary

• Generally, multiply function consists of AND functions to generate the partial products and lots of addition
  – Carry and sum delays of adder cells can be equally critical
• Modified Booth recoding reduces the # of partial products to be added, improves speed
  – Also suitable for 2s complement addition
• Other topics:
  – Can pipeline within the multiplier unit to improve throughput
  – Tree structures to reduce the # of adders needed and speed the result (speed becomes logarithmic in # of bits)

The Design and Implementation of Double-Precision Multiplier in First Generation Cell Processor

• High performance multiplier in 90nm SOI technology.
• 54 x 54 bit multiplier
• Reduce the clock complexity by only allowing dynamic gate in the first half of the clock cycle.
• Large number of partial products compressed in cycle one to reduce the number of flip-flops.
• Heavily pipelined to reach 4GHz frequency target.
Dynamic Booth Encoder and Mux

- Improve the performance of the booth mux.
- Reduce global wire length by reducing the footprint of the booth mux.
- Cross couple NAND gate are used to convert the dynamic signal to static signal.

Static 3-to-2 compressor

- All logic stack height are limited to be 2.
- Transmission gates are tricky when doing sizing.
- Much faster than most standard cell full adder since XOR gates are done in parallel.
Static 4-to-2 compressor

Floorplan Consideration

- Need to reserve space for clock repeaters.
- Need to consider long wire between compressors.
- Data folding.
Reconfigurable SIMD16x16 multipliers for FIR, DCT, FFT filters.

Compact layout and

- Radix-4 Booth Encoding.
- Reduce Sign Extension.
- All logic implemented in static CMOS.
Partial Product Compression

- Simple logic to reduce transistor count and switching activities.
- Optimize the partial product compression tree to improve speed.
- Hybrid adder to reduce the logic overhead.

Layout and Results

- Compact layout to optimize the operand distribution bus (vertical), and carry output (horizontal).
- Achieve 1-cycle 16x16 multiply operation dissipating 9mW when operated at 1GHz and 50°C.