18-447 Lecture 2: Computer Arithmetic: Adders

James C. Hoe
Dept of ECE, CMU
January 14, 2009

Announcements: No class on Monday
Verilog Refresher next Wednesday
Review P&H Ch 3

Handouts: Lab 1 and HW1 will be posted on Blackboard this weekend

Binary Number Representation

- Let \( b_{n-1}b_{n-2}...b_2b_1b_0 \) represent an n-bit unsigned integer
  - its value is \( \sum_{i=0}^{n-1} 2^i b_i \)
  - a finite representation between 0 and \( 2^n - 1 \)
  - e.g., \( 1011_{two} = 8_{ten} + 2_{ten} + 1_{ten} = 11_{ten} \)
    (more commonly rewritten as b’1011=11)
- Often written in Hex for easier human consumption
  - to convert, starting from the LSB, map 4 binary digits at a time into a corresponding hex digit; and vice versa
  - e.g., \( 1010\_1011_{two}=AB_{hex} \)

For converting between binary and decimal, memorize decimal values of \( 2^0 \sim 2^{10} \), and remember \( 2^{10} \) is about 1000.
2's-Complement Number Representation

- Let $b_{n-1}b_{n-2}...b_2b_1b_0$ represent an n-bit signed integer
  - its value is $-2^{n-1}b_{n-1} + \sum_{i=0}^{n-2}2^ib_i$
  - a finite representation between $-2^{n-1}$ and $2^{n-1}-1$
  - e.g., assume 4-bit 2's-complement
    - $b'1011 = -8 + 2 + 1 = -5$
    - $b'1111 = -8 + 4 + 2 + 1 = -1$
- To negate a 2's-complement number
  - add 1 to the bit-wise complement
  - assume 4-bit 2's-complement
    - $(-b'1011) = b'0100 + 1 = b'0101 = 5$
    - $(-b'0101) = b'1010 + 1 = b'1011 = -5$
    - $(-b'1111) = b'0000 + 1 = b'0001 = 1$
    - $(-b'0000) = b'1111 + 1 = b'0000 = 0$

Intuition: a 4-bit example

- how to add two numbers
- what it means to "overflow" the number representation
- how to negate a number

Yes, 0 is a positive number in CS
Smaller to Larger
Binary Representation

- Unsigned numbers
  - pad the left with as many 0s as you need (aka 0-extension)
    e.g. 4'b1111 \rightarrow 8'b0000_1111

- 2's-complement numbers
  - positive: pad the left with as many 0s as you need
  - negative: pad the left with as many 1s as you need
    e.g. 4b'1111 \rightarrow 8'b1111_1111
    4b'1110 \rightarrow 8'b1111_1110
  - or generically, pad the left with the same value as the
    original sign-bit as many times as necessary (aka signed-extension)

What about converting from larger to smaller representation?

(Unsigned) Binary Addition

- Long Hand

What about subtraction?
Full Adder

\[
s = a \oplus b \oplus c_{\text{in}} \\
c_{\text{out}} = bc_{\text{in}} + ac_{\text{in}} + ab
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
& a & b & c_{\text{in}} & s \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

3-way majority

a, b, c_{\text{in}} are functionally indistinguishable as inputs

Unsigned Binary Addition

Could use a “half-adder”, but let’s wait
2's-Complement Addition

Overflow? = (a₃ ⊕ b₃) ? 0 : (a₃ ⊕ s₃)
- can't overflow when adding a pos. and a neg. number
- if 2 pos. numbers yield a neg. number ⇒ V; vice

2's-Complement Subtraction

- Subtracting is like adding the negative
- Negation is easy in a 2's-complement representation

How do you build a comparator (i.e., >, <)?
Analysis of an n-bit "Ripple-Carry" Adder

- **Size/Complexity:** $O(n)$
  - $n \times \text{sizeof( Full Adder )}$

- **Critical Path Delay:** $O(n)$
  - $n \times \text{delayof( Full Adder )}$
  - $n \times 2$ gate delays (assuming 2-level SOP is used)

18-447 Simplified Timing

Clock period chosen to be greater than worst-case $T_{pd}$

What about setup-time, hold-time, skew and such?
High-Performance Adder?

- Intel P4 is designed around a clock period that is twice the 16-bit adder latency

- Using a rough estimation
  gate delay ≈ 0.5 ns-per-micron x feature-size
  a 90nm process has gate delay = 45ps

- If Intel used a ripple-carry adder then P4 should be running ~ 1/(2x2x16x45ps) = 347MHz

- Alternatively speaking, 3GHz P4 would have to add 2 16-bit numbers in ~4 gate delays

Cutting Down the Carry Chain

- How to reduce the carry-propagation delay?
  Remember, long-hand is how most of us add, but not the only way

- Can we compute an intermediate carry signal without first computing the earlier ones
  - e.g., let \( c_m \) (or \( s_m \)) be a function of \( a_m \ldots a_0 \) and \( b_m \ldots b_0 \)
    \[
    c_2 = (a_1a_0b_0) + (a_1a_0c_0) + (a_1b_0c_0) + (b_1a_0b_0) + (b_1a_0c_0) + (b_1b_0c_0) + (a_1b_1)
    \]
  - Complexity grows exponentially in \( n \)
    exponential isn't too bad for small \( n \)'s
  - gate delay is 2, independent of \( n \)
    true for small \( n \)'s

What about large \( n \)'s?
**Carry-Select Adder**

3 adders operate in parallel!!!

\[ \text{cost} = 1.5 \cdot n \cdot \text{FA}_{\text{size}} + \text{mux} \]
\[ \text{delay} = 0.5 \cdot n \cdot \text{FA}_{\text{delay}} + \text{mux-delay} \]
if \( n=16 \) \( \Rightarrow \sim 16 \text{ gate-delays} \)

**Multi-Stage CSA**

\[ \text{cost} = \frac{(2k-1)}{k} \cdot n \cdot \text{FA}_{\text{size}} + \text{mux's} \quad \text{--- for k-stage} \]
\[ \text{delay} = \frac{n \cdot \text{FA}_{\text{delay}}}{k} + (k-1) \cdot \text{mux-delay} \]

- \( k=4, n=16 \) \( \Rightarrow \sim 8 \text{ gate-delay} + 3 \text{ mux-delay} \)
- \( k=8, n=16 \) \( \Rightarrow \sim 4 \text{ gate-delay} + 7 \text{ mux-delay} \)
- \( k=16, n=16 \) \( \Rightarrow \sim 2 \text{ gate-delay} + 15 \text{ mux-delay} \)
Variable-Length CSA

- doubles the cost
- delay set by the longest adder stage, grows by \( O(n^{1/2}) \) with careful critical path tuning

Can we have cut-down the carries without 2x cost?

Carry Generate and Propagate

- If \( a \cdot b \) then \( c_{\text{out}} \) is 1 regardless of \( c_{\text{in}} \) (carry generate)
- if \( a \oplus b \) then \( c_{\text{out}} \) is the same as \( c_{\text{in}} \) (carry propagate)

\[
g_i = a_i \cdot b_i \\
p_i = a_i \oplus b_i
\]

Local decisions based on \( a_i \) and \( b_i \) only
Small Carry-Look-Ahead Adder

Given $g_i = a_i \cdot b_i$
$p_i = a_i \oplus b_i$
$c_{i+1} = g_i + (p_i \cdot c_i)$

Thus
$c_1 = g_0 + (p_0 \cdot c_0)$
$c_2 = g_1 + (p_1 \cdot c_1) = g_1 + (p_1 \cdot (g_0 + (p_0 \cdot c_0))) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$
$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$
$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$
and so on

- We can compute $c_n$ in $O(\log n)$ gate delay and $O(n^2)$ size, only manageable for small $n$
- Given $c_n$ we can compute $s_n$ for a constant additional delay

Prefix Carry-Look-Ahead

Given
$c_1 = g_0 + (p_0 \cdot c_0)$
$c_2 = g_1 + (p_1 \cdot c_1) = g_1 + (p_1 \cdot (g_0 + (p_0 \cdot c_0))) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$
$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$
$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$

As a 4-arity group
$G_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$
$P_4 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$

\[ \text{CLA}_n \]
Prefix Carry-Look-Ahead

This structure can be recursed: \(O(\log n)\) delay, \(O(n)\) size

Computing Individual Carries

Example: 8-bit, 2-ary CLA

\[
\begin{align*}
c_{i0} &= C_{in} \\
c_{i1} &= g_0 + p_0 \cdot C_{in} \\
c_{i2} &= G_{10} + P_{10} \cdot C_{in} \\
c_{i3} &= g_2 + p_2 \cdot c_{i2} \\
c_{i4} &= G_{30} + P_{30} \cdot C_{in} \\
c_{i5} &= g_4 + p_4 \cdot c_{i4} \\
c_{i6} &= G_{54} + P_{54} \cdot c_{i4} \\
c_{i7} &= g_6 + p_6 \cdot (G_{54} + P_{54} \cdot c_{i4}) \\
c_{i8} &= G_{70} + P_{70} \cdot C_{in}
\end{align*}
\]
Large Adder using Carry-Skip

Fast enough and cheaper than computing individual ci's by G.P.

Adder at a Glance

- **Ripple Adder**
  - $O(n)$ size, $O(n)$ delay
- **Carry-Select Adder**
  - $O(n)$ size, $O(n^{0.5})$ delay
- **Carry-Look-Ahead Adder**
  - $O(n^2)$ size, $O(\log n)$ delay
- **Prefix Adder**
  - $O(n)$ size, $O(\log n)$ delay
- But, remember all approaches have design sweet-spots and make different tradeoffs
- There also are circuit-level adder tricks (e.g., Manchester carry chain)
Black Magic of Adder Design

- High-performance adder designs are extremely important to high-performance computing
- Studied extensively in theoretical frameworks
- Worked on extensively in practice
- Nevertheless remain very much a trial-and-error design exercise
- For a 64-bit adder, one might construct
  - adders of various (short) length using 2-level logic
  - a 16-bit adder from small adders with variable-length carry-select
  - a 32-bit adder from 2 16-bit CSA with CLA to determine carry for the upper 16 bits
  - a 64-bit 2-stage CSA adder from 3 32-bit adders