Arithmetic Part 1 (Chapter 3.1-3.5, B.5-B.6)

www.cs.wisc.edu/~karu/courses/cs552/

Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar, Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood
Outline

- Representing Integers
  - Unsigned, 2’s Complement
- Addition and subtraction
- Add/Sub ALU
  - full adder, ripple carry, subtraction,
- Carry lookahead
- Overflow
- Barrel shifter
- Defer: multiplication, division, floating-point
Unsigned Integers

- Recall:
  - n bits give rise to \(2^n\) combinations
  - let us call a string of 32 bits as “\(b_{31} b_{30} \ldots b_3 b_2 b_1 b_0\)"
- \(f(b_{31} \ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0\)
- Treat as normal binary number
  - e.g., 0 . . . 011010101
    \[
    = 1 \times 2^7+1 \times 2^6+0 \times 2^5+1 \times 2^4+0 \times 2^3+1 \times 2^2+0 \times 2^1+1 \times 2^0
    \]
    \[
    = 128 + 64 + 16 + 4 + 1 = 213
    \]
- max \(f(111 \ldots 11) = 2^{32} - 1 = 4, 294, 967, 295\)
- min \(f(000 \ldots 00) = 0\)
- range \([0, 2^{32}-1]\) => # values \((2^{32} - 1) - 0 + 1 = 2^{32}\)
More Generally

- Bits have no inherent meaning
- Conventions define meaning
  - E.g., represent negative integers?
  - Seek circuit simplicity & speed
- \( n \) bits can represent finite possibilities: \( 2^n \)
  - Integers countably infinite \( \rightarrow \) overflow
  - Reals uncountably infinite \( \rightarrow \) overflow, underflow, imprecise
## Integer Representation

- **Sign Magnitude:**
  - $000 = +0$
  - $001 = +1$
  - $010 = +2$
  - $011 = +3$
  - $100 = -0$
  - $101 = -1$
  - $110 = -2$
  - $111 = -3$

- **One's Complement:**
  - $000 = +0$
  - $001 = +1$
  - $010 = +2$
  - $011 = +3$
  - $100 = -3$
  - $101 = -2$
  - $110 = -1$
  - $111 = -0$

- **Two's Complement:**
  - $000 = +0$
  - $001 = +1$
  - $010 = +2$
  - $011 = +3$
  - $100 = -4$
  - $101 = -3$
  - $110 = -2$
  - $111 = -1$

- Balance, number of zeros, **ease of arithmetic**
Two's Complement Integers

- \( f(b_{31} b_{30} \ldots b_1 b_0) = -b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0 \)
  - max \( f(0111 \ldots 11) = 2^{31} - 1 = 2147483647 \)
  - min \( f(100 \ldots 00) = -2^{31} = -2147483648 \) (asymmetric)
- range \([-2^{31}, 2^{31}-1]\) => \#values \(2^{31}-1 - (-2^{31} + 1) = 2^{32}\)
- E.g., -6
- 000 \ldots 0110 --> 111 \ldots 1001 + 1 --> 111 \ldots .1010
Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  
  - 1010  \rightarrow  0101 + 1 = 0110
  - 0110  \rightarrow  1001 + 1 = 1010

- Converting n bit numbers into numbers with more than n bits:
  - copy the most significant bit (the sign bit)
    
    - 0010  \rightarrow  0000 0010
    - 1010  \rightarrow  1111 1010

  - Called "sign extension"
Sign extension

- Consider representation of -2:

<table>
<thead>
<tr>
<th>3bit (decimal)</th>
<th>2-bit (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>011 (+3)</td>
<td></td>
</tr>
<tr>
<td>010 (+2)</td>
<td></td>
</tr>
<tr>
<td>001 (+1)</td>
<td>01 (+1)</td>
</tr>
<tr>
<td>000 (0)</td>
<td>00 (0)</td>
</tr>
<tr>
<td>111 (-1)</td>
<td>11 (-1)</td>
</tr>
<tr>
<td>110 (-2)</td>
<td>10 (-2)</td>
</tr>
<tr>
<td>101 (-3)</td>
<td></td>
</tr>
<tr>
<td>100 (-4)</td>
<td></td>
</tr>
</tbody>
</table>
Addition and Subtraction

• Similar to decimal (carry/borrow twos instead of tens)
• Identical operation for signed and unsigned
  – E.g. **Unsigned vs Signed**
    
    |   |   |   |
    |---|---|---|
    | 0011 | 3 | 3 |
    | 1010 | 10 | -6 |
    | 1101 | 13 | -3 |
Interesting cases

• Show computation in 4-bit 2’s complement representation
  \[ 4 + 4 \]

\[ (-4) + (-4) \]

• Overflow: later
ALU bit-slice

- Bit-wise operation
  - and, or, add
  - Full adder?
  - Sub?

![ALU bit-slice diagram]

- 1-bit Full Adder
- CarryIn
- S-select
- Result
- CarryOut
- Add
- Or
- And

A → CarryIn → and → S-select → Result
B → or → 1-bit Full Adder → add → Mux → Result

(11)
Full adder

- Three inputs and two outputs
- Cout, s = F(a,b,Cin)
  - Cout: only if at least two inputs are set
  - S: only if exactly one input or all three inputs are set
- Logic?

\[
\begin{array}{cc}
\text{a} & \rightarrow \ \text{or} \\
\text{b} & \rightarrow \\
\text{a} & \rightarrow \\
\text{Cin} & \rightarrow \\
\text{Cin} & \rightarrow \\
\text{b} & \rightarrow \\
\end{array}
\]

\[
\begin{array}{cc}
\text{a} & \rightarrow \ \text{or} \\
\text{b} & \rightarrow \\
\text{a} & \rightarrow \\
\text{Cin} & \rightarrow \\
\text{Cin} & \rightarrow \\
\text{Cin} & \rightarrow \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \rightarrow \ \text{or} \\
\text{b} & \rightarrow \\
\text{a} & \rightarrow \\
\text{Cin} & \rightarrow \\
\text{Cin} & \rightarrow \\
\text{Cin} & \rightarrow \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\text{a} \\
\text{b} \\
\text{Cin} \\
\end{array}
\]
Subtract

$A - B = A + (-B)$

- form two complement by invert and add one
Ripple-carry adder
Problem: Slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
  - Delay = 32x CP(Fast adder) + XOR

- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products
  - Flatten expressions to two levels

Can you see the ripple? How could you get rid of it?

\[
\begin{align*}
c_1 &= b_0 c_0 + a_0 c_0 + a_0 b_0 \\
c_2 &= b_1 c_1 + a_1 c_1 + a_1 b_1 \\
c_3 &= b_2 c_2 + a_2 c_2 + a_2 b_2 \\
c_4 &= b_3 c_3 + a_3 c_3 + a_3 b_3 \\
\end{align*}
\]

\[
\begin{align*}
c_2 &= b_1 (b_0 c_0 + a_0 c_0 + a_0 b_0) + a_1 (b_0 c_0 + a_0 c_0 + a_0 b_0) + a_1 b_1 \\
c_3 &= b_1 b_0 c_0 + b_1 a_0 c_0 + b_1 a_0 b_0 + a_1 b_0 c_0 + a_1 a_0 c_0 + a_1 a_0 b_0 + a_1 b_1 \\
c_3 &= ? \\
c_{31} &= ? \text{ Not feasible! Why? Exponential fanin}
\end{align*}
\]
Carry look-ahead

• An approach in-between our two extremes

• Motivation:
  – If we didn't know the value of carry-in, what could we do?
  – When would we always generate a carry?
    • \( g_i = a_i \times b_i \)
  – When would we propagate the carry?
    • \( p_i = a_i + b_i \)

• Did we get rid of the ripple?
CLA: Plumbing Analogy

\[
\begin{align*}
&g_0, p_0, c_1, g_0, p_0, g_1, p_1, g_0, p_0, g_2, p_2, g_3, p_3, c_4 \\
&c_0, c_0, g_0, p_0, g_0, c_0, g_0, p_0, g_0, c_0
\end{align*}
\]
Carry-lookahead adder

\[ C_1 = G_0 + C_0 \cdot P_0 \]
\[ C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \]
\[ C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \]

where:
- \( G = A \text{ and } B \)
- \( P = A \text{ xor } B \)

Truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>C-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>kill</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C-in</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>C-in</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>generate</td>
</tr>
</tbody>
</table>
Carry-Lookahead Adder

• Waitaminute!
  – Nothing has changed
  – Fanin problems if you flatten!
  • Linear fanin, not exponential
  – Ripple problem if you don’t!
• Enables divide-and-conquer
• Figure out Generate and Propagate for 4-bits together
• Compute hierarchically
Carry Lookahead adder

- Height of tree = \( O(\lg(n)) \)
- 32 bit addition: \( k \times \lg(32) = k \times 5 \)
Block level signals

P0

G0
Cascaded CLA

C1 = G0 + C0 ⋅ P0
C2 = G1 + G0 ⋅ P1 + C0 ⋅ P0 ⋅ P1
C3 = G2 + G1 ⋅ P2 + G0 ⋅ P1 ⋅ P2 + C0 ⋅ P0 ⋅ P1 ⋅ P2

C4 = \ldots (22)
Carry-Lookahead Adder

- **Hierarchy**
  - \( G_{i, k} = G_{j+1, k} + P_{j+1, k} \cdot G_{i,j} \) (assume \( i < j +1 < k \))
  - \( P_{i,k} = P_{i,j} \cdot P_{j+1, k} \)
  - \( G_{0,7} = G_{4,7} + P_{4,7} \cdot G_{0,3} \)
  - \( P_{0,7} = P_{0,3} \cdot P_{4,7} \)

- \( G_{8,15} = G_{12,15} + P_{12,15} \cdot G_{8,11} \)
- \( P_{8,15} = P_{8,11} \cdot P_{12, 15} \)

- \( G_{0,15} = G_{8,15} + P_{8,15} \cdot G_{0,7} \)
- \( P_{0,15} = P_{0,7} \cdot P_{8, 15} \)
Computing G’s and Ps

- Parallel algorithm in hardware
Computing C’s

- Different tree.
- Note propagation order
- Worth spending some time to think about the intricacies
Carry-selection: Guess

\[ CP(2n) = 2 \times CP(n) \]

\[ CP(2n) = CP(n) + CP(mux) \]

Carry-select adder
### Overflow

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1001</td>
</tr>
</tbody>
</table>

#### Examples:

- $7 + 3 = 10$ but ...
- $-4 - 5 = -9$ but ...

![Binary Addition Diagram]

CS/ECE 552  
(27)  
Sankaralingam
Overflow

- Overflow: the result is too large (or too small) to represent properly
  - Example: -8 \leq = 4\text{-bit binary number} \leq 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive

On your own: Prove you can detect overflow by:
- Carry into MSB \bigoplus Carry out of MSB
Overflow detection

• Carry into MSB ⊕ Carry out of MSB
  - For N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>X XOR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Negative, Zero

- Required for conditional branches
- Zero
  - How?
  - NOR all 32 bits
  - Avoid 33rd bit (carry out)
- Negative may be required on overflow
  - If (a<b) jump: jump taken if a-b is negative
- Tempting to consider MSB
  - E.g. if (-5 < 4) branch
  - Branch should be taken, but (-5-4) computation results in overflow... so MSB is 0
  - E.g. if (7 < -3) branch
  - Branch should not be taken but (7- (-3)) results in overflow... so MSB is 1.
Shift

- E.g., Shift left logical for \( d<7:0> \) and shamt<2:0>
  - Using 2-1 muxes called Mux(select, in0, in1)
  - \( \text{stage0}<7:0> = \text{Mux}(\text{shamt}<0>, d<7:0>, 0 || d<7:1>) \)
  - \( \text{stage1}<7:0> = \text{Mux}(\text{shamt}<1>, \text{stage0}<7:0>, 00 || \text{stage0}<6:2>) \)
  - \( \text{dout}<7:0> = \text{Mux}(\text{shamt}<2>, \text{stage1}<7:0>, 0000 || \text{stage1}<3:0>) \)

- Other operations
  - Right shift
  - Arithmetic shifts
  - Rotate