Number Systems and Arithmetic

or

Computers go to elementary school

Questions About Numbers

• How do you represent
  – negative numbers?
  – fractions?
  – really large numbers?
  – really small numbers?

• How do you
  – do arithmetic?
  – identify errors (e.g. overflow)?

• What is an ALU and what does it look like?
  – ALU=arithmetic logic unit

Introduction to Binary Numbers

Consider a 4-bit binary number

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Examples of binary arithmetic:

\[
\begin{align*}
3 + 2 &= 5 \\
&
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 0 \\
\hline & 0 & 0 & 1 & 1
\end{array}
\end{align*}
\]

\[
\begin{align*}
3 + 3 &= 6 \\
&
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 1 \\
\hline & 0 & 0 & 1 & 1
\end{array}
\end{align*}
\]

Negative Numbers?

• We would like a *number system* that provides
  – obvious representation of 0,1,2,...
  – uses for addition
  – single value of 0
  – equal coverage of and numbers
  – easy detection of sign
  – easy negation
Some Alternatives

- Sign Magnitude -- MSB is sign bit, rest the same
  - \(-1\) == 1001
  - \(-5\) == 1101

- One’s complement -- flip all bits to negate
  - \(-1\) == 1110
  - \(-5\) == 1010

Two's Complement Representation

- 2's complement representation of negative numbers
  - Take the bitwise inverse and add 1
- Biggest 4-bit Binary Number: 7
- Smallest 4-bit Binary Number: -8

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Two's Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>1000</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Two’s Complement Arithmetic

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2’s Complement Binary</th>
<th>Decimal</th>
<th>2’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Examples: \(7 - 6 = 7 + (-6) = 1\)  \(3 - 5 = 3 + (-5) = -2\)

Some Things We Want To Know About Our Number System

- negation
- sign extension
  - \(+3\) => 0011, 00000011, 0000000000000011
  - \(-3\) => 1101, 11111101, 1111111111111101
- overflow detection
  - \(0101\)      5
  - \(0110\)      6
Overflow Detection

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 2 \\
+ & 0 & 0 & 1 & 1 & 0 & 1 & 3 \\
\hline
0 & 1 & 0 & 1 & 1 & 0 & 1 & 5
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 0 & 0 & -4 \\
+ & 1 & 1 & 1 & 1 & 0 & 0 & -2 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 0 & -6
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 1 & 0 & 1 & 1 & 7 \\
+ & 0 & 0 & 1 & 1 & 1 & 1 & 3 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & -6
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & -4 \\
+ & 1 & 1 & 1 & 1 & 0 & 1 & -5 \\
\hline
0 & 1 & 1 & 1 & 1 & 0 & 1 & 7
\end{array}
\]

So how do we detect overflow?

A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD

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

A 32-bit ALU

1-bit ALU

32-bit ALU

• ALU Control Lines (ALUop) Function
  - 000 And
  - 001 Or
  - 010 Add
  - 110 Subtract
  - 111 Set-on-less-than
How About Subtraction?

- Keep in mind the following:
  - \((A - B)\) is the same as: \(A + (-B)\)
  - 2’s Complement negate: Take the inverse of every bit and add 1
- Bit-wise inverse of \(B\) is \(!B\):
  - \(A - B = A + (-B) = A + (!B + 1) = A + !B + 1\)

Overflow Detection Logic

- Carry into MSB \(! = \) Carry out of MSB
  - For a N-bit ALU: \(\text{Overflow} = \text{CarryIn}[N - 1] \oplus \text{CarryOut}[N - 1]\)

Zero Detection Logic

- Zero Detection Logic is just one BIG NOR gate
  - Any non-zero input to the NOR gate will cause its output to be zero

Set-on-less-than

- Do a subtract
- use sign bit
  - route to bit 0 of result
  - all other bits zero
The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for an N-bit RC adder: 2N-gate delay

MULTIPLY

- Paper and pencil example:
  Multiplicand  
  Multiplier  

Product = ?

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place multiplicand (1 x multiplicand)
- we’ll look at a couple of versions of multiplication hardware

MULTIPLY HARDWARE

Version 1

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg
Multiply Algorithm
Version 1

1. Test
   a. Add multiplicand to product and place the result in Product register

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

1a. Test Multiplier
   a. Multiplier0 = 1
   b. Multiplier0 = 0

1b. Multiplier0 = 0
   a. Multiplier0 = 0
   b. Multiplier0 = 1

Done

Observations on Multiply
Version 1

• 1 clock per cycle => 32-96 clocks per multiply
  – Ratio of multiply to add 32:1 to 96:1
• 1/2 bits in multiplicand always 0 => 64-bit adder is wasted
• MIPS registers and are left and right half of Product
• Gives us MIPS instruction MultU
• What about multiplication?
  – easiest solution is to make both positive & remember whether to complement product when done.

Divide: Paper & Pencil

Quotient

Divisor 1000   1101010 Dividend

Remainder

• See how big a number can be subtracted, creating quotient bit on each step
  – Binary ⇒ 1 * divisor or 0 * divisor
• Dividend = Quotient x Divisor + Remainder

DIVIDE HARDWARE
Version 1

• 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg
Divide Algorithm
Version 1

• Takes n+1 steps for n-bit Quotient & Rem.

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0011</td>
<td>0000</td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

2a. Shift the Quotient register to the left setting the new rightmost bit to 1.

2b. Restore the original value by adding the Divisor register to the Remainder register, and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

3. Shift the Divisor register right 1 bit.

Divide Hardware Version 1

• Again, 64-bit adder is unnecessary.
• Quotient grows as remainder shrinks
• Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
• Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  – Note: Dividend and Remainder must have same sign
  – Note: Quotient negated if Divisor sign & Dividend sign disagree

Key Points

• Instruction Set drives the ALU design
• ALU performance, CPU clock speed driven by adder delay
• Multiplication and division take much longer than addition, requiring multiple addition steps.

So Far

• Can do logical, add, subtract, multiply, divide, ...
• But........
  – what about fractions?
  – what about really large numbers?
Binary Fractions

1011₂ = 1x2³ + 0x2² + 1x2¹ + 1x2⁰
so...
101.011₂ = 1x2² + 0x2¹ + 1x2⁰ + 0x2⁻¹ + 1x2⁻² + 1x2⁻³

e.g.,
.75 = 3/4 = 3/2² = 1/2 + 1/4 = .11

Recall Scientific Notation

+6.02 x 10⁻³
1.673 x 10²

Issues:
° Arithmetic (+, -, *, /)
° Representation, Normal form
° Range and Precision
° Rounding
° Exceptions (e.g., divide by zero, overflow, underflow)
° Errors
° Properties  (negation, inversion, if A = B then A - B = 0)

Floating-Point Numbers

Representation of floating point numbers in IEEE 754 standard:

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>E</td>
<td>M</td>
</tr>
</tbody>
</table>

(exponent: excess 127)

(actual exponent is e = E - 127)

N = (-1)ˢ 2^E-127 (1.M)

0 < E < 255

0 = 0 00000000 0 . . . 0
325 = 101000101 X 2³ = 1.1000101 X 2³
.02 = .0011001101100... X 2⁻³ = .1001101100...

*range of about 2 X 10⁻³⁸ to 2 X 10³⁸
*always normalized (so always leading 1, thus never shown)
*special representation of 0 (E = 00000000) (why?)
*can do integer compare for greater-than, sign

Double Precision Floating Point

Representation of floating point numbers in IEEE 754 standard:

<table>
<thead>
<tr>
<th>1</th>
<th>11</th>
<th>20</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>E</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

(exponent: excess 1023)

(actual exponent is e = E - 1023)

N = (-1)ˢ 2^E-1023 (1.M)

0 < E < 2048

* 52 (+1) bit mantissa
* range of about 2 X 10⁻³⁰⁸ to 2 X 10³⁰⁸
Floating Point Addition

• How do you add in scientific notation?
  \[ 9.962 \times 10^4 + 5.231 \times 10^2 \]

• Basic Algorithm
  1.
  2.
  3.
  4.

Floating Point Multiplication

• How do you multiply in scientific notation?
  \[ (9.9 \times 10^9)(5.2 \times 10^5) = 5.148 \times 10^9 \]

• Basic Algorithm
  1.
  2. Multiply
  3.
  4. Round
  5. Set Sign

FP Accuracy

• Extremely important in scientific calculations
• Very tiny errors can accumulate over time
• IEEE 754 FP standard has four rounding modes
  – always round up (toward +\(\infty\))
  – always round down (toward -\(\infty\))
  – truncate
  – round to nearest
    => in case of tie, round to nearest even
• Requires extra bits in intermediate representations
Extra Bits for FP Accuracy

- *Guard bits* -- bits to the right of the least significant bit of the significand computed for use in normalization (could become significant at that point) and rounding.
- IEEE 754 has two extra bits and calls them *guard* and *round*.

Key Points

- Floating Point extends the range of numbers that can be represented, at the expense of precision (accuracy).
- FP operations are very similar to integer, but with pre- and post-processing.
- Rounding implementation is critical to accuracy over time.