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:

\[
3 + 2 = 5 \\
0 + 0 + 1 + 1 + 1 \\
\]

\[
3 + 3 = 6 \\
0 + 0 + 1 + 1 + 1 \\
\]

Negative Numbers?

- We would like a number system that provides
  - obvious representation of 0,1,2...
  - uses adder for addition
  - single value of 0
  - equal coverage of positive and negative 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

So how do we detect overflow?

Designing an Arithmetic Logic Unit

A One Bit ALU

• This 1-bit ALU will perform AND, OR, and ADD
A 32-bit ALU

1-bit ALU

32-bit ALU

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: Overflow = CarryIn\([N - 1]\) XOR 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 \( \times \) 1011

- Paper and pencil example:
  - \( \times \) 1001

- Binary makes it easy:
  - \( 0 \rightarrow \) place 0 (0 x multiplicand)
  - \( 1 \rightarrow \) place multiplicand (1 x multiplicand)

The point -> ripple carry adders are slow. Faster addition schemes are possible that accelerate the movement of the carry from one end to the other.
MULTIPLY HARDWARE

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg

Observations on Multiply

- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about signed 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

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg
Divide Hardware

- 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.

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

\[ 1.673 \times 10^{24} \]

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

Single precision:
- **Exponent:** excess 127, binary integer (actual exponent is \( e = E - 127 \))
- **Mantissa:** sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M

\[ N = (-1)^S 2^{e-127}(1.M) \]

0 < E < 255

- 0 = 0 00000000 0000000000...
- 1.5 * 2^{-100} = 0 00011011 0000000000...
- 1.75 * 2^{-100} = 0 00011011 1000000000...
- 1.5 * 2^{100} = 0 11100011 0000000000...
- 1.75 * 2^{100} = 0 11100011 1000000000...

- Range of about 2 \times 10^{-38} to 2 \times 10^{38}
- Always normalized (so always leading 1, thus never shown)
- Special representation of 0 (E = 00000000) (why?)
- Can do integer compare for greater-than, sign

What do you notice?

- 0
- 1.5 * 2^{-100}
- 1.75 * 2^{-100}
- 1.5 * 2^{100}
- 1.75 * 2^{100}

- Does this work with negative numbers, as well?

Double Precision Floating Point

Representation of floating point numbers in IEEE 754 standard:

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

Double precision:
- **Exponent:** excess 1023, binary integer (actual exponent is \( e = E - 1023 \))
- **Mantissa:** sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M

\[ N = (-1)^S 2^{e-1023}(1.M) \]

0 < E < 2048

- 52 (+1) bit mantissa
- Range of about 2 \times 10^{-308} to 2 \times 10^{308}

Floating Point Addition

- How do you add in scientific notation?
  \( 9.962 \times 10^4 + 5.231 \times 10^2 \)

- Basic Algorithm
  1. Align
  2. Add
  3. Normalize
  4. Round
**FP Addition Hardware**

- Sign
- Exponent
- Significand
- Sign
- Exponent
- Significand

- Compare exponents
- Shift smaller number right
- Add
- Normalize
- Round
- Rounding hardware
- Sign
- Exponent
- Significand

**Floating Point Multiplication**

- How do you multiply in scientific notation?
  - $(9.9 \times 10^4)(5.2 \times 10^2) = 5.148 \times 10^7$

- Basic Algorithm
  1. Add exponents
  2. Multiply
  3. Normalize
  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

**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.