Number Representations

\[
\text{XVII} \times \text{LIX} = \text{CLXX} - \text{XVII} = \text{D(CCL)LL}
\]

\[
\text{DCCC LLLL} \times \text{X-X X-VII} = \text{MIII} = \text{X-VII} = \text{VIIIIX-VII} = \text{III}
\]
Memory Organization

• Viewed as a large, single-dimension array, with an address.

• A memory address is an index into the array

• "Byte addressing" means that the index points to a byte of memory.

...
Memory Organization

• Bytes are convenient for character data, but most data items use larger "words"

• Memory can be viewed as array of 4-Bytes words.

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits of data</td>
<td>32 bits of data</td>
<td>32 bits of data</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

- \(2^{32}\) bytes with byte addresses from 0 to 232-1
- \(2^{30}\) words with byte addresses 0, 4, 8, ... 232-4

• **MIPS** words are **word aligned** (two least significant bits of a word address are always 00).

• **PowerPC** words need not be word aligned (but access to unaligned words may be slower).
Datapath width

• MIPS fixed-point registers are 32 bits wide
  - It’s called a “32-bit architecture”

• Many architectures are growing to 64-bits.
  - DEC Alpha has been 64-bits all along.
### x86 History

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Data Width</th>
<th>Address Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1971</td>
<td>4004</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1972</td>
<td>8008</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>1974</td>
<td>8080</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>1978</td>
<td>8086</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>1982</td>
<td>80286</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>1985</td>
<td>80386</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>1989</td>
<td>80486</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>1993</td>
<td>Pentium</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>1995</td>
<td>Pentium pro</td>
<td>32</td>
<td>36 physical, 46 virtual</td>
</tr>
<tr>
<td>2001</td>
<td>Itanium</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>
Big Endian vs. Little Endian

4 bytes make up a word, but how??
- Byte address of word is address of first byte.
- But where does this byte go??

Big Endian

Little Endian
Big Endian vs. Little Endian

- Big Endian proponents:
  - IBM, SGI, Sun, Motorola, HP, ...

- Little Endian proponents:
  - DEC, Intel

- Some processors (e.g. PowerPC) provide both
  - If you can figure out how to switch modes or get the compiler to issue “Byte-reversed load’s and store’s”

- Very annoying for writing portable code or for moving data between machines!
# 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:

\[
3 + 2 = 5
\]

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

\[
3 + 3 = 6
\]

\[
\begin{array}{cccc}
1 & 1 \\
0 & 0 & 1 & 1 \\
+ 0 & 0 & 1 & 1 \\
\hline
0 & 1 & 1 & 0
\end{array}
\]
What about negative integers?

• Desirable features of a number system ...
  - obvious representation of 0,1,2...
  - uses adder for addition
  - easy to recognize exceptions (like overflow)
  - 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
  -1 → 1001
  -5 → 1101

• One’s complement -- flip all bits to negate
  -1 → 1110
  -5 → 1010

• Biased -- add constant (e.g. “excess 8”)
  - -1 → 0111, 0 → 1000, 1 → 1001, ... , 7 → 1111
Two’s Complement Representation

- Positive numbers: normal binary representation
- Negative numbers: flip bits (0 $\leftrightarrow$ 1), then add 1

<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>

Smallest 4-bit number: -8

Biggest 4-bit number: 7
Two’s Complement Arithmetic

Uses simple adder for + and - numbers

\[
\begin{align*}
7 + (-6) &= 1 \\
3 + (-5) &= -2
\end{align*}
\]

<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>
Details of 2’s complement notation

• **Negation**
  - flip bits and add 1. *(Magic! Works for + and -)*
  - Might cause overflow.

• **Extend sign when loading into large register**
  - +3 => 0011, 00000011, 0000000000000011
  - -3 => 1101, 11111101, 1111111111111101

• **Overflow detection (need to raise “exception” when answer can’t be represented)**
  
  \[
  \begin{array}{cccc}
  0101 & 5 & & \\
  0110 & 6 & & \\
  1011 & -5 & \text{??!!!} & \\
  \end{array}
  \]
Overflow Detection

So how do we detect overflow?
Key Points

Two’s complement is standard +/- numbers.

Achieves almost all of our goals.

CPU clock speed is driven by adder delay

Adder is used in loads, stores and branches as well as arithmetic.

Thus, using a carry-lookahead adder is important!
Not all integers need a sign

Some numbers are always non-negative
Examples: addresses, character Bytes
Wasteful to always have high-order bit “0”.

Hence, “unsigned” and “signed” operations.

How are they different??
Add’s and subtract’s?
Load’s and stores?
Shifts?
Floating point numbers

Binary fractions:

\[ 1011_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 \]

so...

\[ 101.01_2 = 1\times2^2 + 0\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2} \]

example:

\[ .75 = 3/4 = 1/2 + 1/4 = .11_2 \]
Recall Scientific Notation

- **sign**
- **decimal point**
- **exponent**
- **Mantissa**
- **radix (base)**

**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)
# IEEE 754 Floating-Point Numbers

Single precision representation of \((-1)^S \times 2^{E-127} \times (1.M)\)

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

- **sign bit**
- **exponent:** excess 127 binary integer (actual exponent is e = E - 127)
- **mantissa:** (but leading “1” is omitted)

- **0** = 0 00000000 00 . . . 0
- **-1.5** = 1 01111111 10 . . . 0
- **325** = 101000101 = 1.01000101 x 2^8
  = 0 10001111 010001010000000000000000
- **.02** = .0011001101100... = 1.1001101100... x 2^{-3}
  = 0 01111100 1001101100...

- range of about \(2 \times 10^{-38}\) to \(2 \times 10^{38}\)
- special representation of 0 (E = 00000000)
- some extra values also carefully defined:
  - “denormalized numbers” (no hidden “1”) for very small numbers
  - “Not A Number” (NAN) for results like divide-by-zero, etc.
Double Precision Floating Point in IEEE 754

<table>
<thead>
<tr>
<th>sign</th>
<th>S</th>
<th>E</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
</table>

- **exponent:** excess 1023
- **mantissa:** hidden integer bit: 1.M
- **binary integer**
- actual exponent is $e = E - 1023$

$N = (-1)^S \times 2^{E-1023} \times (1.M)$

- 52 (+1) bit mantissa
- range of about $2 \times 10^{-308}$ to $2 \times 10^{308}$
How do you compare numbers?

- 0
  - 0 00000000 0000000000...
- 1.5 * 2^{-100}
  - 0 00011011 1000000000...
- 1.75 * 2^{-100}
  - 0 00011011 1100000000...
- 1.5 * 2^{100}
  - 0 11100011 1000000000...
- 1.75 * 2^{100}
  - 0 11100011 1100000000...

- Does this work with negative numbers, as well?
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

• Not exact
  - Usually, rounding throws away bits at the end.
  - Subtracting two nearly-equal numbers doesn’t lose bits, but answer doesn’t have as many significant bits.
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
Floating Point Accuracy

• Extremely important in scientific calculations
• Very tiny errors can accumulate over time
• IEEE 754 has four “rounding modes”
  - always round up (toward $+\infty$)
  - always round down (toward $-\infty$)
  - truncate (i.e. round towards zero)
  - 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 three extra bits and calls them *guard, round, and sticky*. 
Floating point anomalies

• Suppose huge/tiny > $2^{53}$
  - tiny + (huge - huge) = tiny + 0 = tiny
  - (tiny + huge) - huge = huge - huge = 0

• Addition is not associative!
  - IEEE 754 requires it to be commutative.

• Sometimes, compilers rearrange code, producing different results.
  - May depend on language or compiler options.
  - Small changes can ultimately have a big effect.
    e.g., whether there is a divide-by-zero exception
Java floating point

- Java Virtual Machine is exactly defined
  - Operations give same answer on all machines
- But some machines have hardware to give more accurate answers
  - Intel x86 uses 80-bit floating-point registers
  - PowerPC uses extra-long register for FMA
  - Result can be 2x speedup on some calculations
- To comply with JVM, extra instructions are needed to “dumb down” the answers!
  - Result is slower code and less accurate answers
- (Java Grande group is exploring these issues)
Computer of the day

• Univac I – first commercial computer (’51) ...
  - Designed by Eckert & Mauchley (creators of ENIAC)
  - Only 8 tons (ENIAC was 20 tons). Clock speed 2.25 MHz.
  - Mercury delay line memory. 1st machine with interrupts.
  - 48 machines built - priced $1M to $1.5M
  - In 50’s, “Univac” was synonymous with “computer”

• ... and first fights over intellectual property
  - E&M applied for patent in ’47
    • U. of P. dean said university should get patent
    • E&M were fired or quit
  - Lawsuit in 60’s, Honeywell v. Sperry Rand over patents
    • 164 cubic feet of evidence
    • Decision: Atanasoff (Iowa State) invented computer in 30’s.