Lecture 3
Representing Data on the Computer
Ramani Duraiswami
AMSC/CMSC 662
Fall 2009
Bits and Bytes; Hexadecimal

- A bit is a single binary digit that can take on one of the two values 0 and 1.
- A byte is a group of eight bits.
- Since a hexadecimal digit (base 16) can be represented by four bits, bytes can be described by pairs of hexadecimal digits.
  - 0, 1, 2, 3, 4, 5, 6, 7, 8,
  - 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000
  - 9, A (10), B(11), C(12), D(13), E(14), F(15)
  - 1001, 1010, 1011, 1100, 1101, 1110, 1111
  - 01011110_2 may be represented by the number 5E_{16}.
Words & Addresses

• Memory locations on a 32 bit machine, usually consist of 4 bytes => called a word
• Relationship between words and data of various sizes:
  – byte 8bits, 1 byte
  – short or half word 16bits, 2 bytes
  – word 32bits, 4 bytes
  – long or double word 64 bits, 8 bytes
• Memory is addressed using an index, which is itself a binary number
• Addresses, usually are available for every byte
• Addresses can be grouped by bit-shifts
  – byte xx...xxxx
  – half word xx...xxx0
  – word xx...xx00
  – double word xx...x000
• Recall that words/memory are shipped across a bus
  – Contiguous blocks can be loaded easier
Memory fragmentation

• Usually memory is allocated in chunks of a word or of two words.
• If the data, e.g. a C-struct or a Fortran 90 Type may consists of a mixture of a four byte variable, a two byte variable and a four byte variable.
• This will cause wastage of two bytes due to memory fragmentation.

<table>
<thead>
<tr>
<th>low order bits of the address</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>x...x00</td>
<td>A</td>
</tr>
<tr>
<td>x...x10</td>
<td>B</td>
</tr>
<tr>
<td>x...x00</td>
<td>X</td>
</tr>
<tr>
<td>x...x10</td>
<td>C</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Little Endian/Big Endian Ordering

• Ordering of bytes in a word
• we could store its bytes in any order, as long as the particular ordering is consistent from word to word.
• In practice, there are only two orders used: big endian and little endian.
• Consider a four byte word ABCD.
  – The byte A is called the leading or most significant byte.
  – The byte D is called the trailing or least significant byte.
• In big-endian representation the bytes are stored in increasing memory locations beginning with the leading byte.
• In little-endian representation the bytes are stored in increasing memory locations beginning with the trailing byte.
• ABCD is stored as follows in the two methods
<table>
<thead>
<tr>
<th>address</th>
<th>BE</th>
<th>LE</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>a+1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>a+2</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>a+3</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>
• Way of storing is usually unimportant, except for two situations
  – Transferring binary data between machines
  – Using bit shift operations
Bit operations

- Very efficient set of operations that are provided in processors, and that have representations in programming languages
- Will return to these in a later class
Unsigned Integers

• On a machine nonnegative integers can be represented by regarding the bits in a word as a binary number, that is, an unsigned integer.
• Integers can be added, subtracted, multiplied, and divided.
• Addition and subtraction are the fastest operations.
• Multiplication can be almost as fast as addition.
• Division is much slower.
• However, multiplication and division by two can be implemented using shifts.
• Exceptions
• However, the result of these operations cannot always be represented in the computer.
  
  \[13_{10} + 5_{10} = 1101_2 + 0101_2 = 10010_2\]

• If we stay with 4 bit memory locations, the above sum cannot be represented.
  
  This situation is called an arithmetic exception. Arithmetic exceptions can be handled by an automatic default or by trapping to an exception handler.

• In some situations, when we are performing calculations modulo some number, we may discard the extra bit.

• This gives the answer \(0010_2 = 2_{10}\) which is just \(13 + 5 \pmod{16}\). In many applications this is just what we want.
Exception handling

• In others this is a wrong result and we need to use exception handling

• Operations leading to exceptions
  – a + b: Overflow
  – a - b: Negative result, i.e., a < b
  – a*b: Overflow
  – a/b: Division by zero or noninteger result

• This may need to bring in logic that causes the process to stop, and bring in further information from main memory and may be computationally expensive.

• Fatal exceptions: cause process to abort

• Default handling: may be turned on

• For division it is generally agreed that division by zero is fatal

• There is also agreement about what to do when the result is not an integer

• E.g., 17/3 = 5.6667 -> 5

• The exact quotient should be truncated toward zero.
Signed Integers

- Stored in a four byte word
- Can have two byte, byte, and 8 byte versions
- Need to figure out how to represent sign:
  - **Sign magnitude**: if the first bit is zero, then the number is positive. Otherwise, it is negative.
    - 0 0 1 1 Denotes +11.
    - 1 0 1 1 Denotes -11.
    - Zero: Both 0 0 0 0 and 1 0 0 0 represent zero
  - **Two’s complement**: As before the if the first bit is zero the number is positive
    - However values for the negative numbers are determined by subtraction of the number from $2^n$.
    - There is one more negative number possible

- Signed numbers can overflow or underflow.
- Two's complement representation seems unnatural, but in fact it is often preferred because it makes addition easier to implement in silicon.
Floating point

• Attempt to
  – Handle decimal numbers
  – increase the range of numbers that can be represented
  – Provide a standard by which exceptions are consistently handled
Scientific Notation

-6.023 $\times$ $10^{-23}$

- **Sign**
- **Normalized Mantissa**
- **Exponent**
- **Base**
- **Sign of Exponent**
Floating point on a computer

- Using fixed number of bits represent real numbers on a computer
- Once a base is agreed we store each number as two numbers and two signs
  - Mantissa and exponent
- Mantissa is usually “normalized”
- If we have infinite spaces to store these numbers, we can represent arbitrarily large numbers
- With a fixed number of spaces for the two numbers (mantissa and exponent)
Binary Floating Point Representation

• Same basic idea as scientific notation
• Modifications and improvements based on
  – Hardware architecture
  – Efficiency (Space & Time)
  – Additional requirements
    • Infinity
    • Not a number (NaN)
    • Not normalized
    • etc.
Floating point on a computer

• If we wanted to store $15 \times 2^{11}$, we would need 16 bits:
  \[
  01111100000000000000000000000000
  \]

• Instead we store it as three numbers

• $(-1)^S \times F \times 2^E$, with $F = 15$ saved as 01111 and $E = 11$ saved as 01011.

• Now we can have fractions/decimals, too:
  \[
  \text{binary } .101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}.
  \]
### IEEE-754 (single precision)

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa (significand)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00000000000000000000000</td>
</tr>
</tbody>
</table>

\[
(-1)^s \times 1.M \times 2^{E-127}
\]

- **Sign**: 1 is understood
- **Mantissa (w/o leading 1)**
- **Base**: 2
- **Exponent**: E-127
IEEE-754 (double precision)

$$(-1)^s \times 1.f \times 2^e$$

- Sign
- 1 is understood
- Mantissa (w/o leading 1)
- Base
- Exponent
IEEE - 754

Most nonzero floating-point numbers are normalized. This means they can be expressed as

\[ x = \pm (1 + f) \cdot 2^e \]

The quantity \( f \) is the fraction or mantissa and \( e \) is the exponent. The fraction satisfies

\[ 0 \leq f < 1 \]

and must be representable in binary using at most 52 bits. In other words, \( 2^{52} f \) is an integer in the interval

\[ 0 \leq 2^{52} f < 2^{52} \]

The exponent \( e \) is an integer in the interval

\[ -1022 \leq e \leq 1023 \]

The finiteness of \( f \) is a limitation on \textit{precision}. The finiteness of \( e \) is a limitation on \textit{range}. Any numbers that don’t meet these limitations must be approximated by ones that do.

Double-precision floating-point numbers are stored in a 64 bit word, with 52 bits for \( f \), 11 bits for \( e \), and one bit for the sign of the number. The sign of \( e \) is accommodated by storing \( e + 1023 \), which is between 1 and \( 2^{11} - 2 \). The two
Can be written...

\[ (-1)^S \times 2^{E} \times 1.f \]

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa (significand)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000000000</td>
<td>0000000000000000 ...... 0000000000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E+1023$</th>
<th>$0 &lt; E+1023 &lt; 2047$</th>
<th>$E+1023$ == 2047</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f==0$</td>
<td>0</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$f\sim=0$</td>
<td>Non-normalized typically underflow</td>
<td>Floating point Numbers</td>
</tr>
</tbody>
</table>
• $x = \pm (1+f) \times 2^e$
• $0 \leq f < 1$
• $f = (\text{integer} < 2^{52})/2^{52}$
• $-1022 \leq e \leq 1023$
• $e = \text{integer}$