Computational Methods
CMSC/AMSC 460

Ramani Duraiswami,
Dept. of Computer Science

Computer Memory
• Everything on a computer is stored digitally
  – Numbers, Letters, Instructions
• Memory is
  – Limited
  – has two states 0 and 1 ➔ Binary
• How do you represent numbers for scientific computation?
Fixed point representation

- How can we represent a number in a computer’s memory?
- Fixed point is an obvious way:
- Used to represent integers on computers, and real numbers on some DSPs:
- Each **word** (storage location) in a machine contains a fixed number of digits.
- Example: An old style calculator display with 6-digits

\[
\begin{array}{cccccc}
0 & 0 & 2 & 0 & 0 & 5 \\
\end{array}
\]

- This only allows us to represent integers and uses a decimal system

---

Binary/Decimal/Octal/Hexadecimal

- Computer memory usually has two states
  - Assigned to 0 and 1
  - Leads to a binary representation
- Numbers can be represented in different bases
- Usually humans use decimal
  - Perhaps because we have ten fingers
- Octal and Hexadecimal representations arise by considering 3 or 4 memory locations together
  - Lead to $2^3 = 8$ and $2^4 = 16$ numbers
Binary Representation

- **binary (base 2)** representation.
  
  \[0 \ 1 \ 0 \ 1 \ 1 \ 0\]

- Each digit has a value 0 or 1.

- If the number above is binary, its value is
  
  \[1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0.\] (or 22 in base 10)

- Adding numbers in binary

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

  Note the “carry” here!

Bits and Bytes; Hexadecimal

- **Bit**: a single binary digit
  
  – Can take on one of the two values 0 and 1.

- **A byte is a group of eight bits**
  
  – A “nibble” is four bits or half a byte

- **Hexadecimal digit (base 16) == four bits,**
  
  – bytes can be described by pairs of hexadecimal digits.

  \[
  0, \ 1, \ 2, \ 3, \ 4, \ 5, \ 6, \ 7, \ 8, \ 0000, \ 0001, \ 0010, \ 0011, \ 0100, \ 0101, \ 0110, \ 0111, \ 1000, \ 1001, \ 1010, \ 1011, \ 1100, \ 1101, \ 1110, \ 1111
  \]

  \[
  9, \ A(10), \ B(11), \ C(12), \ D(13), \ E(14), \ F(15)
  \]

- **01011110₂ may be represented by the number 5E₁₆,**
Words

- 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
- Internally, by default, Matlab stores all numbers in double words
  - Can specify other types of storage

Unsigned Integers

- Integers can be added, subtracted, multiplied, and divided.
- **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 some 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., $a < b$
  - $a \times b$: 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., $17/3 = 5.6667 \rightarrow 5$
- The exact quotient should be truncated toward zero.

Negative numbers

- One way computers represent negative numbers is using the sign-magnitude representation:
- **Sign magnitude**: if the first bit is zero, then the number is positive. Otherwise, it is negative.
- 0 0 0 1 1 Denotes +11.
- 1 0 0 1 1 Denotes -11.

**Range of fixed point numbers**

Largest 5-digit (5 bit) binary number: 0 1 1 1 1 =15
Smallest: 1 1 1 1 1 =-15
Smallest positive: 0 0 0 0 1 =1
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:
  - Two approaches
    - **Sign magnitude**: if the first bit is zero, then 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 the way that is used in computer processors, as it is easier to implement in hardware.

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

**Fixed point arithmetic:**
- Easy: always get an integer answer.
- Either we get exactly the right answer, or we can detect overflow.
- The numbers that we can store are equally spaced.
- Disadvantage: **very** limited range of numbers.
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

• Use Scientific Notation as a guide

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) the number representation is more limited

Binary Floating Point Representation

- Same basic idea as scientific notation
- Modifications and improvements based on
  - Hardware architecture
  - Efficiency (Space & Time)
  - Additional requirements: Need to represent conditions which arise during calculations
    - Infinity
    - Not a number (NaN)
    - Underflow
Floating point on a computer

• If we wanted to store $15 \times 2^{11}$, we would need 16 bits:
  \[0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0\]

• 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)

\[
\begin{array}{|c|c|c|c|c|}
\hline
0 & 1 & 8 & 9 & 31 \\
\hline
0 & \text{exponent} & \text{mantissa (significand)} & & \\
\hline
\end{array}
\]

\[
(-1)^S \times 1.f \times 2^{E-127}
\]
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 precision. The finiteness of \( e \) is a limitation on 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