Lecture 1: Bits, Data, and Operations

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Who Am I?

• Mohamed Zahran (aka Z)
• Computer architecture/OS/Compilers Interaction
• http://www.mzahran.com
• Office hours: Tue 1:00-3:00 pm
• Room: WWH 320
Formal Goals of This Course

• What happens under the hood in computer systems
• How are software and hardware related
• From algorithms to circuits

You will be able to write programs in C and understand what’s going on underneath.
Informal Goals of This Course

• To get more than an A
• To build strong background in computer science
• To use what you have learned in MANY different contexts
• To enjoy the course!
The Course Web Page

- Lecture slides
- Info about mailing list, labs, ...
- Useful links (manuals, tools, ...)
Grading

- Homework : 30%
- Project : 15%
- Midterm Exam : 20%
- Final Exam : 35%
So...What is a computer?

“The Computer is only a fast idiot, it has no imagination; it cannot originate action. It is, and will remain, only a tool to human beings.”

American Library Association’s reaction to UNIVAC computer Exhibit at the 1964 New York World’s fair.

A computer is a symbol-processing machine

Computer: electronic genius?
• NO! Electronic idiot!
• Does exactly what we tell it to, nothing more.
It all starts with a "problem"
Automating Algorithm Execution

• Algorithm *development*
  – A detailed know-how
  – Granularity depends on the machine
  – Done with human brain power

• Algorithm *execution*
  – Sequencing
  – Execution
Two Side Effects

• Algorithm must handle different set of inputs

• Algorithm must be presented to the machine in a *formal way*
Hardware and Software
From Theory to Practice

• In theory, computer can compute anything that’s possible to compute
  – given enough memory and time

• In practice, solving problems involves computing under constraints.
  – time
    • weather forecast, next frame of animation, ...
  – cost
    • cell phone, automotive engine controller, ...
  – power
    • cell phone, handheld video game, ...
Can We Solve Anything With a Computer?

- **Undecidable**
  - Cannot be solved by an algorithm
  - e.g. Halting problem (given a program and inputs for it, decide whether it will run forever or will eventually halt.)

- **Unsolvable**
  - No finite algorithm
  - e.g. Goldbach’s conjecture (Every even number greater than 2 can be written as the sum of two primes.)

- **Intractable**
  - Unreasonable amount of time and resources
Hierarchical View of a Computer System

• A computer system is complicated
• In order to facilitate its study and analysis, it is advisable to divide it into levels
How do we Understand computers?

• Need to understand *abstractions* such:
  - Algorithms
  - Applications software
  - Systems software
  - Assembly Language
  - Machine Language (ISA)
  - Microarchitecture
  - Logic design
  - Device level
  - Semiconductors/Silicon used to build transistors
  - Properties of atoms, electrons, and quantum dynamics
Two Recurring Themes

• Abstraction
  – Productivity enhancer - don’t need to worry about details...
    You can drive a car without knowing how the internal combustion engine works.
  – ...until something goes wrong!
    Where’s the dipstick? What’s a spark plug?
  – Important to understand the components and how they work together.

• Hardware vs. Software
  – It’s not either/or - both are components of a computer system.
  – Even if you specialize in one, you should understand capabilities and limitations of both.
Problem → Algorithm Development → Programmer

High Level Language

Assembly Language

Machine Language

Microarchitecture

Logic Level

Device Level → Semiconductors → Quantum
Problem Definition Level

- Taking a complex real-life problem and formulating it so as to be solved by a computer (abstraction/modeling)
- Requires simplification (which details to remove?)
- Using mathematical model, graph theory, etc.
Algorithm Level

- Precise step-by-step procedure
- Steps must be well defined, to be executed by a machine (no ambiguity)
- Algorithm development is a creative process
- Finite number of steps
- Pseudocode or flowchart
High-Level Language Level

- e.g. C/C++/C#, Java, Fortran, Lisp, etc.
- Used by application programmers and systems programmers
- Can we build machines executing HLL right away?
- Compiler’s job is not only translating
Assembly Language Level

• More primitive instructions than HLL
• English version of the machine language + some more
• User mode and kernel mode
• Can we go from this level to HLL?
ISA (Instruction Set Architecture) level

- A very important abstraction
  - interface between hardware and low-level software
  - advantage: different implementations of the same architecture
  - disadvantage: sometimes prevents using new innovations

- Modern instruction set architectures:
  - x86_64, IA-32, PowerPC, MIPS, SPARC, ARM, and others
Instructions

• Language of the Machine
• Platform-specific
• A limited set of machine language commands "understood" by hardware (e.g. ADD, LOAD, STORE, RET)
• We’ll study MIPS instruction set architecture and x86 instruction set architecture
From HLL to ISA: an Example
Microarchitecture Level

• Resources and techniques used to implement the ISA
  – Pentium IV implements the x86 ISA
  – Motorola G4 implements the Power PC ISA
• Register files, ALU, Fetch unit, etc.
• Realize intended cost/performance goals
• Interpretation done by the control unit
Logic-Design Level

• Gates
• Multiplexers, decoders, PLA, etc.
• Synchronous (i.e. clocked): the most widely used
• Asynchronous
Device Level

- Transistors and wires
- Implement the digital logic gates
- Lower level:
  - Solid state physics
  - Machine looks more analog than digital!
Many Choices at Each Level

Solve a system of equations

- Red-black SOR
- Gaussian elimination
- Jacobi iteration
- Multigrid

- FORTRAN
- C
- C++
- Java

- PowerPC
- Intel x86
- Atmel AVR

- Centrino
- Pentium 4
- Xeon

- Ripple-carry adder
- Carry-lookahead adder

- CMOS
- Bipolar
- GaAs

Tradeoffs:
- cost
- performance
- power
- (etc.)
Our First Steps...
How do we represent data in a computer?

• How do we represent information using electrical signals?
• At the lowest level, a computer is an electronic machine.
• Easy to recognize two conditions:
  – presence of a voltage - we call this state “1”
  – absence of a voltage - we call this state “0”
A Computer is a Binary Digital Machine

- Basic unit of information is the binary digit, or bit.
- Values with more than two states require multiple bits.
  - A collection of two bits has four possible states: 00, 01, 10, 11
  - A collection of three bits has eight possible states: 000, 001, 010, 011, 100, 101, 110, 111
  - A collection of \( n \) bits has \( 2^n \) possible states.
What kinds of data do we need to represent?

- **Numbers** - signed, unsigned, integers, floating point, complex, rational, irrational, ...
- **Text** - characters, strings, ...
- **Images** - pixels, colors, shapes, ...
- **Sound**
- **Logical** - true, false
- **Instructions**
- ...

• **Data type:**
  - *representation* and *operations* within the computer
Unsigned Integers

• Non-positional notation
  – could represent a number (“5”) with a string of ones (“11111”)
  – problems?

• Weighted positional notation
  – like decimal numbers: “329”
  – “3” is worth 300, because of its position, while “9” is only worth 9

\[
\begin{align*}
329 & = 3 \times 10^2 + 2 \times 10^1 + 9 \times 10^0 \\
101 & = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0
\end{align*}
\]
Unsigned Integers (cont.)

- An $n$-bit unsigned integer represents $2^n$ values: from 0 to $2^n-1$.

<table>
<thead>
<tr>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
Unsigned Binary Arithmetic

- Base-2 addition – just like base-10!
  - add from right to left, propagating carry

\[
\begin{align*}
10010 & + 1001 & + 1011 & + 1 \\
11011 & & 11101 & & 10000 \\
\end{align*}
\]

\[
\begin{align*}
10111 \\
+ 111 \\
\end{align*}
\]
How About Negative Numbers

<table>
<thead>
<tr>
<th>Sign Magnitude:</th>
<th>One's Complement</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = +0</td>
<td>000 = +0</td>
<td>000 = +0</td>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
</tr>
<tr>
<td>100 = -0</td>
<td>100 = -3</td>
<td>100 = -4</td>
</tr>
<tr>
<td>101 = -1</td>
<td>101 = -2</td>
<td>101 = -3</td>
</tr>
<tr>
<td>110 = -2</td>
<td>110 = -1</td>
<td>110 = -2</td>
</tr>
<tr>
<td>111 = -3</td>
<td>111 = -0</td>
<td>111 = -1</td>
</tr>
</tbody>
</table>

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?
Signed Integers

- With \( n \) bits, we have \( 2^n \) distinct values.
  - assign about half to positive integers and about half to negative

- Positive integers
  - just like unsigned - zero in most significant (MS) bit
  \( 00101 = 5 \)

- Negative integers
  - sign-magnitude - set MS bit to show negative, other bits are the same as unsigned
  \( 10101 = -5 \)
  - one’s complement - flip every bit to represent negative
  \( 11010 = -5 \)
  - in either case, MS bit indicates sign: 0=positive, 1=negative
Two’s Complement

- Problems with sign-magnitude and 1’s complement
  - two representations of zero (+0 and -0)
  - arithmetic circuits are complex
    - How to add two sign-magnitude numbers?
      - e.g., try 2 + (-3)
    - How to add to one’s complement numbers?
      - e.g., try 4 + (-3)
- Two’s complement representation developed to make circuits easy for arithmetic.
  - for each positive number (X), assign value to its negative (-X),
    such that X + (-X) = 0 with “normal” addition, ignoring carry out

```
  00101   (5)   01001   (9)
+ 11011    (-5) + 10111    (-9)
  00000    (0)   00000    (0)
```
Two's Complement Signed Integers

- **MS bit is sign bit.**
- **Range of an n-bit number: \(-2^{n-1}\) through \(2^{n-1} - 1\).**
  - The most negative number \((-2^{n-1})\) has no positive counterpart.

<table>
<thead>
<tr>
<th>(-2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
<th>(-2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0</td>
<td>1 0 0 0</td>
<td>-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1</td>
<td>1 0 0 1</td>
<td>-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>2</td>
<td>1 0 1 0</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>3</td>
<td>1 0 1 1</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>4</td>
<td>1 1 0 0</td>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>5</td>
<td>1 1 0 1</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>6</td>
<td>1 1 1 0</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>7</td>
<td>1 1 1 1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Converting Binary (2’s C) to Decimal

1. If leading bit is one, take two’s complement to get a positive number.

2. Add powers of 2 that have “1” in the corresponding bit positions.

3. If original number was negative, add a minus sign.

<table>
<thead>
<tr>
<th>n</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>
Examples

\[ X = 00100111_{\text{two}} \]
\[ = 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1 \]
\[ = 39_{\text{ten}} \]

\[ X = 11100110_{\text{two}} \]
\[ -X = 00011010 \]
\[ = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 \]
\[ = 26_{\text{ten}} \]
\[ X = -26_{\text{ten}} \]

Assuming 8-bit 2’s complement numbers.
Converting Decimal to Binary (2's C)

- First Method: Division

1. Find magnitude of decimal number. (Always positive.)
2. Divide by two - remainder is least significant bit.
3. Keep dividing by two until answer is zero, writing remainders from right to left.
4. Append a zero as the MS bit; if original number was negative, take two's complement.

<table>
<thead>
<tr>
<th>X = (104_{ten} )</th>
<th>104/2 = 52 r0</th>
<th>bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>52/2 = 26 r0</td>
<td>bit 1</td>
<td></td>
</tr>
<tr>
<td>26/2 = 13 r0</td>
<td>bit 2</td>
<td></td>
</tr>
<tr>
<td>13/2 = 6 r1</td>
<td>bit 3</td>
<td></td>
</tr>
<tr>
<td>6/2 = 3 r0</td>
<td>bit 4</td>
<td></td>
</tr>
<tr>
<td>3/2 = 1 r1</td>
<td>bit 5</td>
<td></td>
</tr>
</tbody>
</table>

| X = \(01101000_{two} \) | 1/2 = 0 r1 | bit 6 |
Converting Decimal to Binary (2's C)

- **Second Method: Subtract Powers of Two**
  1. Find magnitude of decimal number.
  2. Subtract largest power of two less than or equal to number.
  3. Put a one in the corresponding bit position.
  4. Keep subtracting until result is zero.
  5. Append a zero as MS bit; if original was negative, take two's complement.

<table>
<thead>
<tr>
<th>n</th>
<th>(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>

\[ X = 104_{\text{ten}} \]

\[ \begin{align*}
104 - 64 &= 40 & \text{bit 6} \\
40 - 32 &= 8 & \text{bit 5} \\
8 - 8 &= 0 & \text{bit 3}
\end{align*} \]

\[ X = 01101000_{\text{two}} \]
Operations: Arithmetic and Logical

- We now have a good representation for signed integers, so let’s look at some arithmetic operations:
  - Addition
  - Subtraction
  - Sign Extension
- We’ll also look at overflow conditions for addition.
- Multiplication, division, etc., can be built from these basic operations.
- Logical operations are also useful:
  - AND
  - OR
  - NOT
Addition

• As we've discussed, 2's comp. addition is just binary addition.
  – assume all integers have the same number of bits
  – ignore carry out
  – for now, assume that sum fits in n-bit 2's comp. representation

\[
\begin{align*}
01101000 & \quad (104) & + & 11110110 & \quad (-10) \\
+ & 11110000 & \quad (-16) & + & \quad \text{_________} & \quad (-9) \\
01011000 & \quad (98) & & & & \quad (-19)
\end{align*}
\]
Subtraction

- Negate subtrahend (2nd no.) and add.
  - Assume all integers have the same number of bits
  - Ignore carry out
  - For now, assume that difference fits in $n$-bit 2's comp. representation

\[
\begin{array}{c}
\text{01101000} \quad \text{(104)} \\
- \text{00010000} \quad \text{(16)} \\
\text{01101000} \quad \text{(104)} \\
+ \text{11110000} \quad \text{(-16)} \\
\text{01011000} \quad \text{(88)}
\end{array}
\begin{array}{c}
\text{11110110} \quad \text{(-10)} \\
- \text{11110110} \quad \text{(-10)} \\
\text{01011000} \quad \text{(-1)}
\end{array}
\]
Sign Extension

• To add two numbers, we must represent them with the same number of bits.

• If we just pad with zeroes on the left:

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100</td>
</tr>
<tr>
<td>1100</td>
<td>00001100</td>
</tr>
</tbody>
</table>

• Instead, replicate the MS bit -- the sign bit:

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100</td>
</tr>
<tr>
<td>1100</td>
<td>11111100</td>
</tr>
</tbody>
</table>
Detecting Overflow

• No overflow when adding a positive and a negative number
• No overflow when signs are the same for subtraction
• Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
Logical Operations

- **Operations on logical TRUE or FALSE**
  - two states -- takes one bit to represent: TRUE=1, FALSE=0

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- View \( n \)-bit number as a collection of \( n \) logical values
  - operation applied to each bit independently
Examples of Logical Operations

- **AND**
  - useful for clearing bits
    - AND with zero = 0
    - AND with one = no change

- **OR**
  - useful for setting bits
    - OR with zero = no change
    - OR with one = 1

- **NOT**
  - unary operation -- one argument
  - flips every bit

\[
\begin{align*}
\text{11000101} & \quad \text{AND} \quad \text{00001111} \\
\text{00000101} & \\
\text{11000101} & \quad \text{OR} \quad \text{00001111} \\
\text{11001111} & \\
\text{11000101} & \quad \text{NOT} \quad \text{11000101} \\
\text{00111010} & 
\end{align*}
\]
Hexadecimal Notation

- It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.
  - fewer digits -- four bits per hex digit
  - less error prone -- easy to corrupt long string of 1’s and 0’s

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

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>
Converting from Binary to Hexadecimal

- Every four bits is a hex digit.
  - start grouping from right-hand side

011101010001111010011010111

\[ \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \]

3 A 8 F 4 D 7

This is not a new machine representation, just a convenient way to write the number.
Fractions: Fixed-Point

• How can we represent fractions?
  – Use a “binary point” to separate positive from negative powers of two -- just like “decimal point.”
  – 2’s comp addition and subtraction still work.
    • if binary points are aligned

\[
\begin{align*}
00101000.101 & \quad (40.625) \\
+11111110.110 & \quad (-1.25) \\
\hline
00100111.011 & \quad (39.375)
\end{align*}
\]
Floating Point (a brief look)

• We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., $3.15576 \times 10^9$

• Representation:
  - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range

• IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand
IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit (called hidden 1 technique, except when exp = -127)
- Exponent is “biased” to make sorting easier
  - all 0s is smallest exponent
  - all 1s is largest exponent
  - bias of 127 for single precision and 1023 for double precision
- summary: \((-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}\)

- Example:
  - decimal: 
    
    \[-.75 = - \left( \frac{1}{2} + \frac{1}{4} \right)\]

  - binary: 
    
    \[-.11 = -1.1 \times 2^{-1}\]

  - floating point: exponent = 126 = 01111110

  - IEEE single precision: 
    
    10111111010000000000000000000000
More about IEEE floating Point Standard

Single Precision:

\[
(-1)^{\text{sign}} \times (1 + \text{significand}) \times 2^{\text{exponent} - 127}
\]

The variables shown in red are the numbers stored in the machine

Important! Significant is always 0.XXXX
Floating Point Example

what is the decimal equivalent of

1 01110110 10110000...0
Text: ASCII Characters

- **ASCII**: Maps 128 characters to 7-bit code.
  - both printable and non-printable (ESC, DEL, ...) characters

```
00 nul 10 dle 20 sp 30 0 40 @ 50 P 60 ` 70 p
01 soh 11 dc1 21 ! 31 1 41 A 51 Q 61 a 71 q
02 stx 12 dc2 22 " 32 2 42 B 52 R 62 b 72 r
03 etx 13 dc3 23 # 33 3 43 C 53 S 63 c 73 s
04 eot 14 dc4 24 $ 34 4 44 D 54 T 64 d 74 t
05 enq 15 nak 25 % 35 5 45 E 55 U 65 e 75 u
06 ack 16 syn 26 & 36 6 46 F 56 V 66 f 76 v
07 bel 17 etb 27 ' 37 7 47 G 57 W 67 g 77 w
08 bs 18 can 28 ( 38 8 48 H 58 X 68 h 78 x
09 ht 19 em 29 ) 39 9 49 I 59 Y 69 i 79 y
0a nl 1a sub 2a * 3a : 4a J 5a Z 6a j 7a z
0b vt 1b esc 2b + 3b ; 4b K 5b [ 6b k 7b { 0c np 1c fs 2c , 3c < 4c L 5c \ 6c l 7c | 0d cr 1d gs 2d - 3d = 4d M 5d ] 6d m 7d } 0e so 1e rs 2e . 3e > 4e N 5e ^ 6e n 7e ~ 0f si 1f us 2f / 3f ? 4f O 5f _ 6f o 7f del
```
Interesting Properties of ASCII Code

• What is relationship between a decimal digit ('0', '1', ...) and its ASCII code?

• What is the difference between an upper-case letter ('A', 'B', ...) and its lower-case equivalent ('a', 'b', ...)?

• Given two ASCII characters, how do we tell which comes first in alphabetical order?
Other Data Types

- **Text strings**
  - sequence of characters, terminated with NULL (0)

- **Image**
  - array of pixels
    - monochrome: one bit (1/0 = black/white)
    - color: red, green, blue (RGB) components (e.g., 8 bits each)
    - other properties: transparency
  - hardware support:
    - typically none, in general-purpose processors
    - MMX -- multiple 8-bit operations on 32-bit word

- **Sound**
  - sequence of fixed-point numbers
Conclusions

• In this lecture we made our first steps toward understanding bits, data, and operations on them.
• Computers understand only binary
• Binary presentation is enough to deal with many different type of data (signed/unsigned numbers, floating points, ASCII, ... )