CSCI-GA.1144-001

PAC II

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 2:00-4:00 pm
• Room: WWH 320
Main 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.
My wishlist for 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 NYU classes, slides, ...
• Useful links (manuals, tools, ... )
Grading

- Homework : 15%
- Project : 25%
- Midterm Exam : 20%
- Final Exam : 40%
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.
High Level Language → Assembly Language → Machine Language

Compiler (translator) → Assembler (translator) → Control Unit (Interpreter) → Microsequencer (Interpreter) → 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 that can execute 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, 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

High-level language program (in C)

swap(int v[], int k)
int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;

Compiler

Assembly language program (for MIPS)

swap:
    mli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31

Assembler

Binary machine language program (for MIPS)

000000000101000010000000000000011000
000000000011000011000000000000100001
1001110011100101000000000000000000
1001110111101000000000000000000000
110111101111010000000000000000000
101011101111100100000000000000000
100000111111110000000000000000000
000001111000000000000000000000000
000000011110000000000000000000000
000000000000000000000000000000000
Microarchitecture Level

• Resources and techniques used to implement the ISA
  – Intel i7 implements the x86 ISA
  – IBM Power 8 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 at that level!
Many Choices at Each Level

Solve a system of equations

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

Programming languages:
- FORTRAN
- C
- C++
- Java

Processor types:
- PowerPC
- Intel x86
- Atmel AVR
- Centrino
- Pentium 4
- Xeon

Adder types:
- Ripple-carry adder
- Carry-lookahead adder

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

This is the one used in computers

\[
\begin{align*}
3 \times 100 + 2 \times 10 + 9 \times 1 &= 329 \\
1 \times 4 + 0 \times 2 + 1 \times 1 &= 5
\end{align*}
\]
Unsigned Integers (cont.)

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

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

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

\[
\begin{array}{c}
10010 \\
+ 1001 \\
\hline
11011
\end{array}
\quad
\begin{array}{c}
10010 \\
+ 1011 \\
\hline
11101 \\
+ 1 \\
\hline
10000
\end{array}
\quad
\begin{array}{c}
1111 \\
+ 1 \\
\hline
10111 \\
+ 111 \\
\hline
101111
\end{array}
\]
### 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

\[
\begin{array}{c c c}
00101 & 01001 & 00000 \\
(5) & (9) & (0) \\
+ 11011 & + 10111 & + 00000 \\
(-5) & (-9) & (0)
\end{array}
\]
**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.

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | | $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|--------|--------|--------|--------|--|--------|--------|--------|--------| |
| 0      | 0      | 0      | 0      | 0 | 1      | 0      | 0      | 0      | -8 |
| 0      | 0      | 0      | 1      | 1 | 1      | 0      | 0      | 1      | -7 |
| 0      | 0      | 1      | 0      | 2 | 1      | 0      | 1      | 0      | -6 |
| 0      | 0      | 1      | 1      | 3 | 1      | 0      | 1      | 1      | -5 |
| 0      | 1      | 0      | 0      | 4 | 1      | 1      | 0      | 0      | -4 |
| 0      | 1      | 0      | 1      | 5 | 1      | 1      | 0      | 1      | -3 |
| 0      | 1      | 1      | 0      | 6 | 1      | 1      | 1      | 0      | -2 |
| 0      | 1      | 1      | 1      | 7 | 1      | 1      | 1      | 1      | -1 |
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

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

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

\[
\begin{array}{c|c|c}
\hline
n & 2^n & \\
\hline
0 & 1 & \\
1 & 2 & 16 \\
2 & 4 & 16 \\
3 & 8 & 16 \\
4 & 16 & 16 \\
5 & 32 & 16 \\
6 & 64 & 16 \\
7 & 128 & 16 \\
8 & 256 & 16 \\
9 & 512 & 16 \\
10 & 1024 & 16 \\
\hline
\end{array}
\]

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></td>
<td>52/2 = 26 r0</td>
<td>bit 1</td>
</tr>
<tr>
<td></td>
<td>26/2 = 13 r0</td>
<td>bit 2</td>
</tr>
<tr>
<td></td>
<td>13/2 =  6 r1</td>
<td>bit 3</td>
</tr>
<tr>
<td></td>
<td>6/2 =  3 r0</td>
<td>bit 4</td>
</tr>
<tr>
<td></td>
<td>3/2 =  1 r1</td>
<td>bit 5</td>
</tr>
<tr>
<td>X = 01101000_{two}</td>
<td>1/2 = 0 r1</td>
<td>bit 6</td>
</tr>
</tbody>
</table>
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.

\[
\begin{array}{c|c}
 n & 2^n \\
\hline
 0 & 1 \\
 1 & 2 \\
 2 & 4 \\
 3 & 8 \\
 4 & 16 \\
 5 & 32 \\
 6 & 64 \\
 7 & 128 \\
 8 & 256 \\
 9 & 512 \\
10 & 1024 \\
\end{array}
\]

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

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

\[
X = 01101000_{\text{two}}
\]
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

\[
\begin{array}{c}
01101000 \quad \text{(104)} \quad 11110110 \quad \text{(-10)} \\
+ \quad 11110000 \quad \text{(-16)} \\
\hline
01011000 \quad \text{(98)} \quad + \quad \underline{\text{[ ]}} \quad \text{(-9)}
\end{array}
\]

\[
\begin{array}{c}
01011000 \quad \text{(98)} \quad + \quad 11110110 \quad \text{(-19)}
\end{array}
\]
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

\[
\begin{array}{c}
01101000 \quad (104) \quad 11110110 \quad (-10) \\
-00010000 \quad (16) \quad - \quad \quad \quad \quad (9) \\
01101000 \quad (104) \quad 11110110 \quad (-10) \\
+11110000 \quad (-16) \quad + \quad \quad \quad \quad (9) \\
01011000 \quad (88) \quad 01\text{ - other digits }
\end{array}
\]
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>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>(4)</td>
<td>00000100 (still 4)</td>
</tr>
<tr>
<td>1100</td>
<td>(-4)</td>
<td>00001100 (12, not -4)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>(4)</td>
<td>00000100 (still 4)</td>
</tr>
<tr>
<td>1100</td>
<td>(-4)</td>
<td>11111100 (still -4)</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
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>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<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

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & & & & & & & \\
\end{array}
\]

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

\[
\begin{align*}
00101000.101 & \text{ (40.625)} \\
+ 11111110.110 & \text{ (-1.25)} \\
\hline
00100111.011 & \text{ (39.375)}
\end{align*}
\]
Fractional Binary Numbers

- Value: \[ \sum_{k=-j}^{i} b_k \times 2^k \]
How does the machine store floating points?

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., $3.15576 \times 10^9$
- **IEEE Standard 754**
  - Supported by all major CPUs
  - Standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard
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
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: 10111111101000000000000000000000
Floating Point Example

what is the decimal equivalent of

1 01110110 10110000...0
Precisions

- **Single precision: 32 bits**
  - \(s\) \(\text{exp}\) \(\frac{1}{2}\) bits \(\frac{1}{2}\) bits
  - 1 8-bits 23-bits

- **Double precision: 64 bits**
  - \(s\) \(\text{exp}\) \(\frac{1}{2}\) bits \(\frac{1}{2}\) bits
  - 1 11-bits 52-bits

- **Extended precision: 80 bits (Intel only)**
  - \(s\) \(\text{exp}\) \(\frac{1}{2}\) bits \(\frac{1}{2}\) bits
  - 1 15-bits 63 or 64-bits
Based on $\exp$ we have 3 encoding schemes

- $\exp \neq 0\ldots0$ or $11\ldots1 \rightarrow$ normalized encoding
- $\exp = 0\ldots000 \rightarrow$ denormalized encoding
- $\exp = 1111\ldots1 \rightarrow$ special value encoding
  - $\text{frac} = 000\ldots0$
  - $\text{frac} = \text{something else}$
1. Normalized Encoding

• **Condition:** exp \( \neq \) 000...0 and exp \( \neq \) 111...1

  referred to as Bias

• **Exponent is:** 
  \[ E = \text{Exp} - (2^{k-1} - 1), \text{ k is the # of exponent bits} \]
  – Single precision: \( E = \exp - 127 \)
  – Double precision: \( E = \exp - 1023 \)

• **Significand is:** 
  \[ M = 1.xxx...x_2 \]
  – Range(\( M \)) = [1.0, 2.0-\( \epsilon \)]
  – Get extra leading bit for free
Normalized Encoding Example

• Value: \( \text{Float } F = 15213.0; \)
  \[ 15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13} \]

• Significand
  \[ M = 1.1101101101101_2 \]
  \[ \frac{\text{frac}}{1101101101101000000000000_2} \]

• Exponent
  \[ E = \text{exp} - \text{Bias} = \text{exp} - 127 = 13 \]
  \[ \rightarrow \text{exp} = 140 = 10001100_2 \]

• Result:
  \[
  \begin{array}{cccc}
  s & \text{exp} & \text{frac} \\
  0 & 10001100 & 1101101101101000000000000 \\
  \end{array}
  \]
2. Denormalized Encoding

- **Condition**: \( \text{exp} = 000\ldots0 \)

- **Exponent value**: \( E = 1 - \text{Bias} \) (instead of \( E = 0 - \text{Bias} \))
- **Significand is**: \( M = 0.xxx\ldots x_2 \) (instead of \( M=1.xxx_{2} \))

- **Cases**
  - \( \text{exp} = 000\ldots0, \frac{}{\text{frac}} = 000\ldots0 \)
    - Represents zero
    - Note distinct values: +0 and -0
  - \( \text{exp} = 000\ldots0, \frac{}{\text{frac}} \neq 000\ldots0 \)
    - Numbers very close to 0.0
3. Special Values Encoding

- **Condition:** \( \text{exp} = 111...1 \)

- **Case:** \( \text{exp} = 111...1, \frac{\text{frac}}{\text{frac}} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -\infty \)

- **Case:** \( \text{exp} = 111...1, \frac{\text{frac}}{\text{frac}} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty - \infty, \infty \times 0 \)
# ASCII Characters

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

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 nul</td>
<td>0</td>
</tr>
<tr>
<td>01 soh</td>
<td>!</td>
</tr>
<tr>
<td>02 stx</td>
<td>&quot;</td>
</tr>
<tr>
<td>03 etx</td>
<td>#</td>
</tr>
<tr>
<td>04 eot</td>
<td>$</td>
</tr>
<tr>
<td>05 enq</td>
<td>%</td>
</tr>
<tr>
<td>06 ack</td>
<td>&amp;</td>
</tr>
<tr>
<td>07 bel</td>
<td>'</td>
</tr>
<tr>
<td>08 bs</td>
<td>(</td>
</tr>
<tr>
<td>09 ht</td>
<td>)</td>
</tr>
<tr>
<td>0a nl</td>
<td>*</td>
</tr>
<tr>
<td>0b vt</td>
<td>+</td>
</tr>
<tr>
<td>0c np</td>
<td>,</td>
</tr>
<tr>
<td>0d cr</td>
<td>-</td>
</tr>
<tr>
<td>0e so</td>
<td>.</td>
</tr>
<tr>
<td>0f si</td>
<td>/</td>
</tr>
<tr>
<td>10 dle</td>
<td>1</td>
</tr>
<tr>
<td>11 dc1</td>
<td>2</td>
</tr>
<tr>
<td>12 dc2</td>
<td>3</td>
</tr>
<tr>
<td>13 dc3</td>
<td>4</td>
</tr>
<tr>
<td>14 dc4</td>
<td>5</td>
</tr>
<tr>
<td>15 nak</td>
<td>6</td>
</tr>
<tr>
<td>16 syn</td>
<td>7</td>
</tr>
<tr>
<td>17 etb</td>
<td>8</td>
</tr>
<tr>
<td>18 can</td>
<td>9</td>
</tr>
<tr>
<td>19 em</td>
<td>a</td>
</tr>
<tr>
<td>1a sub</td>
<td>b</td>
</tr>
<tr>
<td>1b esc</td>
<td>c</td>
</tr>
<tr>
<td>1c fs</td>
<td>d</td>
</tr>
<tr>
<td>1d gs</td>
<td>e</td>
</tr>
<tr>
<td>1e rs</td>
<td>f</td>
</tr>
<tr>
<td>1f us</td>
<td>g</td>
</tr>
<tr>
<td>20 sp</td>
<td>h</td>
</tr>
<tr>
<td>21 !</td>
<td>i</td>
</tr>
<tr>
<td>22 &quot;</td>
<td>j</td>
</tr>
<tr>
<td>23 #</td>
<td>k</td>
</tr>
<tr>
<td>24 $</td>
<td>l</td>
</tr>
<tr>
<td>25 %</td>
<td>m</td>
</tr>
<tr>
<td>26 &amp;</td>
<td>n</td>
</tr>
<tr>
<td>27 '</td>
<td>o</td>
</tr>
<tr>
<td>28 (</td>
<td>p</td>
</tr>
<tr>
<td>29 )</td>
<td>q</td>
</tr>
<tr>
<td>30 0</td>
<td>r</td>
</tr>
<tr>
<td>31 1</td>
<td>s</td>
</tr>
<tr>
<td>32 2</td>
<td>t</td>
</tr>
<tr>
<td>33 3</td>
<td>u</td>
</tr>
<tr>
<td>34 4</td>
<td>v</td>
</tr>
<tr>
<td>35 5</td>
<td>w</td>
</tr>
<tr>
<td>36 6</td>
<td>x</td>
</tr>
<tr>
<td>37 7</td>
<td>y</td>
</tr>
<tr>
<td>38 8</td>
<td>z</td>
</tr>
<tr>
<td>39 9</td>
<td>{</td>
</tr>
<tr>
<td>40 @</td>
<td>}</td>
</tr>
<tr>
<td>41 A</td>
<td></td>
</tr>
<tr>
<td>42 B</td>
<td>_</td>
</tr>
<tr>
<td>43 C</td>
<td>^</td>
</tr>
<tr>
<td>44 D</td>
<td>_</td>
</tr>
<tr>
<td>45 E</td>
<td>`</td>
</tr>
<tr>
<td>46 F</td>
<td>a</td>
</tr>
<tr>
<td>47 G</td>
<td>b</td>
</tr>
<tr>
<td>48 H</td>
<td>c</td>
</tr>
<tr>
<td>49 I</td>
<td>d</td>
</tr>
<tr>
<td>50 P</td>
<td>e</td>
</tr>
<tr>
<td>51 Q</td>
<td>f</td>
</tr>
<tr>
<td>52 R</td>
<td>g</td>
</tr>
<tr>
<td>53 S</td>
<td>h</td>
</tr>
<tr>
<td>54 T</td>
<td>i</td>
</tr>
<tr>
<td>55 U</td>
<td>j</td>
</tr>
<tr>
<td>56 V</td>
<td>k</td>
</tr>
<tr>
<td>57 W</td>
<td>l</td>
</tr>
<tr>
<td>58 X</td>
<td>m</td>
</tr>
<tr>
<td>59 Y</td>
<td>n</td>
</tr>
<tr>
<td>60 `</td>
<td>o</td>
</tr>
<tr>
<td>61 a</td>
<td>p</td>
</tr>
<tr>
<td>62 b</td>
<td>q</td>
</tr>
<tr>
<td>63 c</td>
<td>r</td>
</tr>
<tr>
<td>64 d</td>
<td>s</td>
</tr>
<tr>
<td>65 e</td>
<td>t</td>
</tr>
<tr>
<td>66 f</td>
<td>u</td>
</tr>
<tr>
<td>67 g</td>
<td>v</td>
</tr>
<tr>
<td>68 h</td>
<td>w</td>
</tr>
<tr>
<td>69 i</td>
<td>x</td>
</tr>
<tr>
<td>70 j</td>
<td>y</td>
</tr>
<tr>
<td>71 k</td>
<td>z</td>
</tr>
<tr>
<td>72 l</td>
<td>{</td>
</tr>
<tr>
<td>73 m</td>
<td>}</td>
</tr>
<tr>
<td>74 n</td>
<td></td>
</tr>
<tr>
<td>75 o</td>
<td>_</td>
</tr>
<tr>
<td>76 p</td>
<td>^</td>
</tr>
<tr>
<td>77 q</td>
<td>`</td>
</tr>
<tr>
<td>78 r</td>
<td>a</td>
</tr>
<tr>
<td>79 s</td>
<td>b</td>
</tr>
<tr>
<td>80 t</td>
<td>c</td>
</tr>
<tr>
<td>81 u</td>
<td>d</td>
</tr>
<tr>
<td>82 v</td>
<td>e</td>
</tr>
<tr>
<td>83 w</td>
<td>f</td>
</tr>
<tr>
<td>84 x</td>
<td>g</td>
</tr>
<tr>
<td>85 y</td>
<td>h</td>
</tr>
<tr>
<td>86 z</td>
<td>i</td>
</tr>
<tr>
<td>87 {</td>
<td>j</td>
</tr>
<tr>
<td>88 }</td>
<td>k</td>
</tr>
<tr>
<td>89</td>
<td></td>
</tr>
<tr>
<td>90 }</td>
<td>m</td>
</tr>
<tr>
<td>91 }</td>
<td>n</td>
</tr>
<tr>
<td>92 ~</td>
<td>o</td>
</tr>
<tr>
<td>93 del</td>
<td>p</td>
</tr>
</tbody>
</table>
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, ... )