Computer Organization & Assembly Languages

Introduction

Pu-Jen Cheng
2008/09/15
Course Administration

- **Instructor:** Pu-Jen Cheng (CSIE R323)
  
  pjcheng@csie.ntu.edu.tw
  
  http://www.csie.ntu.edu.tw/~pjcheng

- **Class Hours:** 2:00pm-5:00pm, Monday

- **Classroom:** CSIE R102

- **TA(s):** 戴瑋彥 b93705014@ntu.edu.tw

- **Course Information:**
  
  Announce: http://www.csie.ntu.edu.tw/~pjcheng/course/asm2008/
  
  Q&A: bbs://ptt.cc → CSIE_ASM
Textbook

- http://www.asmirvine.com
References

Computer Systems: A Programmer's Perspective
By Randal E. Bryant and David R. O'Hallaron,
Prentice Hall
http://csapp.cs.cmu.edu/

The Art of Assembly Language
By Randy Hyde,
http://webster.cs.ucr.edu/AoA/Windows/PDFs/0_PDFIndexWin.html

System Software: An Introduction to Systems Programming
By Leland L. Beck
Addison-Wesley
Pre-requisite

- Experiences in writing programs in a high-level language such as C, C++, and Java
Course Grading (tentative)

- Assignments (55%)
- Class participation (5%)
- Midterm exam (20%)
- Final exam (20%)
Some materials used in this course are adapted from:

- The slides prepared by Kip Irvine for the book, Assembly Language for Intel-Based Computers, 5th Ed.
- The slides prepared by S. Dandamudi for the book, Introduction to Assembly Language Programming, 2nd Ed.
- Introduction to Computer Systems, CMU (http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f05/www/)
What is Assembly Language

- First Glance at Assembly Language
Translating Languages

**English:** Display the sum of A times B plus C.

**C++:**

```cpp
cout << (A * B + C);
```

**Assembly Language:**

```assembly
mov eax,A
mul B
add eax,C
call WriteInt
```

**Intel Machine Language:**

```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```
A Simple Example in VC++

```cpp
#include <stdio.h>

int main(void)
{
    int a, b, c, d;
    a = 1;
    b = 2;
    c = 3;
    d = a * b + c;
    printf("a * b + c = %d\n", d);
    return 0;
}
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-858993460</td>
</tr>
<tr>
<td>b</td>
<td>-858993460</td>
</tr>
<tr>
<td>c</td>
<td>-858993460</td>
</tr>
<tr>
<td>d</td>
<td>-858993460</td>
</tr>
</tbody>
</table>
3: int main(void)
4: {
00401010  push    ebp
00401011  mov     ebp,esp
00401013  sub     esp,50h
00401016  push    ebx
00401017  push    esi
00401018  push    edi
00401019  lea     edx,[ebp-50h]
0040101c  mov     ecx,14h
00401021  mov     eax,[GCCGCCCh]
00401026  rep stos  dword ptr [edx]
5:     int a, b, c, d;
6:
7:     a = 1;
00401028  mov     dword ptr [ebp-4],1
8:     b = 2;
0040102f  mov     dword ptr [ebp-8],2
9:     c = 3;
00401036  mov     dword ptr [ebp-0Ch],3
10:    d = a + b + c;
0040103d  mov     eax,dword ptr [ebp-4]
00401040  imul    eax,dword ptr [ebp-8]
00401044  add     eax,dword ptr [ebp-0Ch]
00401047  mov     dword ptr [ebp-10h],eax
11:    printf("a * b + c = %d\n", d);
0040104a  mov     ecx,dword ptr [ebp-10h]
0040104d  push    ecx
0040104e  push    offset string "a * b + c = %d\\n" (0042001c)
00401053  call    printf (00401090)
00401058  add     esp,8
12:    return 0;
gcc -s prog.c

.file "prog1.c"
.section .rodata

.string "a * b + c = %d\n"
.text
.globl main
.type main, @function

main:
  leal 4(%esp), %ecx
  andl $-16, %esp
  pushl -4(%ecx)
  pushl %ebp
  movl %esp, %ebp
  pushl %ecx
  subl $80, %esp
  movl $1, -20(%ebp)
  movl $2, -16(%ebp)
  movl $3, -12(%ebp)
  movl -20(%ebp), %eax
  imull -16(%ebp), %eax
  addl -12(%ebp), %eax
  movl %eax, -8(%ebp)
  movl -8(%ebp), %eax
  movl %eax, 4(%esp)
  movl $.LCO, (%esp)
  call printf
  movl $0, %eax
  addl $36, %esp
  popl %ecx
  popl %ebp
  leal -4(%ecx), %esp
  ret

.size main, .-main
.ident "GCC: (GNU) 4.1.2 20060901 (prerelease) [Debian 4.1.1-13]"
.section .note.GNU-stack,"",@progbits
The Compilation System

Diagram:

1. **Hello.c** (source program (text)) → **Pre-processor (cpp)** → **Hello.i** (modified source program (text)) → **Compiler (cc1)** → **Hello.s** (assembly program (text)) → **Assembler (as)** → **Hello.o** (relocatable object programs (binary)) → **Linker (ld)** → **Hello** (executable object program (binary))
First Glance at Assembly Language

- Low-level language
  - Each instruction performs a much lower-level task compared to a high-level language instruction
  - Most high-level language instructions need more than one assembly instruction

- One-to-one correspondence between assembly language and machine language instructions
  - For most assembly language instructions, there is a machine language equivalent

- Directly influenced by the instruction set and architecture of the processor (CPU)
Comparisons with High-level Languages

- Advantages of Assembly Languages
  - Space-efficiency
    (e.g. hand-held device softwares, etc)
  - Time-efficiency
    (e.g. Real-time applications, etc)
  - Accessibility to system hardwares
    (e.g., Network interfaces, device drivers, video games, etc)

- Advantages of High-level Languages
  - Development
  - Maintenance (Readability)
  - Portability (compiler, virtual machine)
Comparisons with High-level Languages (cont.)

<table>
<thead>
<tr>
<th>Type of Application</th>
<th>High-Level Languages</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business application software, written for single platform, medium to large size.</td>
<td>Formal structures make it easy to organize and maintain large sections of code.</td>
<td>Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.</td>
</tr>
<tr>
<td>Hardware device driver.</td>
<td>Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties.</td>
<td>Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.</td>
</tr>
<tr>
<td>Business application written for multiple platforms (different operating systems).</td>
<td>Usually very portable. The source code can be recompiled on each target operating system with minimal changes.</td>
<td>Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain.</td>
</tr>
<tr>
<td>Embedded systems and computer games requiring direct hardware access.</td>
<td>Produces too much executable code, and may not run efficiently.</td>
<td>Ideal, because the executable code is small and runs quickly.</td>
</tr>
</tbody>
</table>
Why Taking the Course?

This Course

Basic Concepts of Computer Organization

Computer Design

Computer Organization

Computer Architecture

System Software

Assembly Language

Assembler, Linker, Loader

Compiler, Operating System, …
“I really don’t think that you can write a book for serious computer programmers unless you are able to discuss low-level details.”

Donald Knuth (高德納)
The Art of Computer Programming

http://en.wikipedia.org/wiki/Donald_Knuth
Course Coverage

- Basic Concepts
- IA-32 Processor Architecture
- Assembly Language Fundamentals
- Data Transfers, Addressing, and Arithmetic
- Procedures
- Conditional Processing
- Integer Arithmetic
- Advanced Procedures
- Strings and Arrays
- Structures and Macros
- High-Level Language Interface
- Assembler, Linker, and Loader
- Other Advanced Topics (optional)
What You Will Learn

- Basic principles of computer architecture
- IA-32 processors and memory management
- Basic assembly programming skills
- How high-level language is translated to assembly
- How assembly is translated to machine code
- How application program communicates with OS
- Interface between assembly to high-level language
Performance: Multiword Arithmetic

- Longhand multiplication
  - Final 128-bit result in P:A

- P := 0; count := 64
- A := multiplier; B := multiplicand
- while (count > 0)
  - if (LSB of A = 1)
  - then   P := P+B
  -     CF := carry generated by P+B
  - else   CF := 0
  - end if
- shift right CF:P:A by one bit position
- count := count-1
- end while
Example

- A = $1101_2$ (13)
- B = $0101_2$ (5)

<table>
<thead>
<tr>
<th>Initial state</th>
<th>After P+B</th>
<th>After the shift</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CF  P  A</td>
<td>CF  P  A</td>
</tr>
<tr>
<td>Initial state</td>
<td>?  0000  1101</td>
<td>--  ----  ----</td>
</tr>
<tr>
<td>Iteration 1</td>
<td>0  0101  1101</td>
<td>?  0010  1110</td>
</tr>
<tr>
<td>Iteration 2</td>
<td>0  0010  1110</td>
<td>?  0001  0111</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>0  0110  0111</td>
<td>?  0011  0011</td>
</tr>
<tr>
<td>Iteration 4</td>
<td>0  1000  0011</td>
<td>?  0100  0001</td>
</tr>
</tbody>
</table>
Time Comparison

Multiplication time comparison on a 2.4-GHz Pentium 4 system
Chapter 1: Basic Concept

- Virtual Machine Concept
- Data Representation
- Boolean Operations
Translating Languages

**English:** Display the sum of A times B plus C.

**C++:** `cout << (A * B + C);`

**Assembly Language:**
```
mov eax, A
mul B
add eax, C
call WriteInt
```

**Intel Machine Language:**
```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```
Virtual Machines

Abstractions for computers

- High-Level Language
- Assembly Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Digital Logic

Levels:
- Level 5: Machine-independent
- Level 4: High-Level Language
- Level 3: Operating System
- Level 2: Instruction Set Architecture
- Level 1: Microarchitecture
- Level 0: Digital Logic
High-Level Language

- Level 5
- Application-oriented languages
  - C++, Java, Pascal, Visual Basic . . .
- Programs compile into assembly language (Level 4)
Assembly Language

- Level 4
- Instruction mnemonics that have a one-to-one correspondence to machine language
- Calls functions written at the operating system level (Level 3)
- Programs are translated into machine language (Level 2)
Operating System

- Level 3
- Provides services to Level 4 programs
- Translated and run at the instruction set architecture level (Level 2)
Instruction Set Architecture

- Level 2
- Also known as conventional machine language
- Executed by Level 1 (microarchitecture) program
Microarchitecture

- Level 1
  - Interprets conventional machine instructions (Level 2)
  - Executed by digital hardware (Level 0)
Digital Logic

- Level 0
- CPU, constructed from digital logic gates
- System bus
- Memory

next: Data Representation
Data Representation

- Binary Numbers
  - Translating between binary and decimal
- Binary Addition
- Integer Storage Sizes
- Hexadecimal Integers
  - Translating between decimal and hexadecimal
  - Hexadecimal subtraction
- Signed Integers
  - Binary subtraction
- Fractional Binary Numbers
- Character Storage
- Machine Words
Binary Representation

- **Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

![Graph showing binary representation with voltage levels 0.0V to 3.3V with 0s and 1s marked at 2.8V and 0.5V, respectively.](image-url)
Binary Numbers

- Digits are 1 and 0
  - 1 = true
  - 0 = false
- MSB – most significant bit
- LSB – least significant bit

Bit numbering:
Binary Numbers

- Each digit (bit) is either 1 or 0
- Each bit represents a power of 2:

Every binary number is a sum of powers of 2

<table>
<thead>
<tr>
<th>$2^n$</th>
<th>Decimal Value</th>
<th>$2^n$</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>1</td>
<td>$2^8$</td>
<td>256</td>
</tr>
<tr>
<td>$2^1$</td>
<td>2</td>
<td>$2^9$</td>
<td>512</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4</td>
<td>$2^{10}$</td>
<td>1024</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8</td>
<td>$2^{11}$</td>
<td>2048</td>
</tr>
<tr>
<td>$2^4$</td>
<td>16</td>
<td>$2^{12}$</td>
<td>4096</td>
</tr>
<tr>
<td>$2^5$</td>
<td>32</td>
<td>$2^{13}$</td>
<td>8192</td>
</tr>
<tr>
<td>$2^6$</td>
<td>64</td>
<td>$2^{14}$</td>
<td>16384</td>
</tr>
<tr>
<td>$2^7$</td>
<td>128</td>
<td>$2^{15}$</td>
<td>32768</td>
</tr>
</tbody>
</table>
Translating Binary to Decimal

Weighted positional notation shows how to calculate the decimal value of each binary bit:

\[ dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0) \]

\( D = \) binary digit

binary 00001001 = decimal 9:

\[ (1 \times 2^3) + (1 \times 2^0) = 9 \]
Translating Unsigned Decimal to Binary

- Repeatedly divide the decimal integer by 2.
- Each remainder is a binary digit in the translated value:

<table>
<thead>
<tr>
<th>Division</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>37 / 2</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>18 / 2</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>9 / 2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4 / 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2 / 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 / 2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

37 = 100101
Binary Addition

- Starting with the LSB, add each pair of digits, include the carry if present.

```
  0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1
+   0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1
________________________
  0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1
```

bit position: 7 6 5 4 3 2 1 0

(4) (7) (11)
Integer Storage Sizes

Standard sizes:

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Range (low–high)</th>
<th>Powers of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned byte</td>
<td>0 to 255</td>
<td>0 to ((2^8 - 1))</td>
</tr>
<tr>
<td>Unsigned word</td>
<td>0 to 65,535</td>
<td>0 to ((2^{16} - 1))</td>
</tr>
<tr>
<td>Unsigned doubleword</td>
<td>0 to 4,294,967,295</td>
<td>0 to ((2^{32} - 1))</td>
</tr>
<tr>
<td>Unsigned quadword</td>
<td>0 to 18,446,744,073,709,551,615</td>
<td>0 to ((2^{64} - 1))</td>
</tr>
</tbody>
</table>

What is the largest unsigned integer that may be stored in 20 bits?
Large Measurements

- Kilobyte (KB), $2^{10}$ bytes
- Megabyte (MB), $2^{20}$ bytes
- Gigabyte (GB), $2^{30}$ bytes
- Terabyte (TB), $2^{40}$ bytes
- Petabyte, $2^{50}$ bytes
- Exabyte, $2^{60}$ bytes
- Zettabyte, $2^{70}$ bytes
- Yottabyte, $2^{80}$ bytes
- Googol, $10^{100}$
Binary values are represented in hexadecimal.

**Table 1-5** Binary, Decimal, and Hexadecimal Equivalents.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</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>10</td>
<td>A</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>1011</td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>1100</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>1101</td>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>1110</td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>1111</td>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>
Translating Binary to Hexadecimal

- Each hexadecimal digit corresponds to 4 binary bits.

Example: Translate the binary integer 000101101010011110010100 to hexadecimal:

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>A</th>
<th>7</th>
<th>9</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0110</td>
<td>1010</td>
<td>0111</td>
<td>1001</td>
<td>0100</td>
</tr>
</tbody>
</table>
Converting Hexadecimal to Decimal

- Multiply each digit by its corresponding power of 16:
  \[ \text{dec} = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0) \]

- Hex 1234 equals \((1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)\), or decimal 4,660.

- Hex 3BA4 equals \((3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0)\), or decimal 15,268.
Powers of 16

- Used when calculating hexadecimal values up to 8 digits long:

<table>
<thead>
<tr>
<th>$16^n$</th>
<th>Decimal Value</th>
<th>$16^n$</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16^0$</td>
<td>1</td>
<td>$16^4$</td>
<td>65,536</td>
</tr>
<tr>
<td>$16^1$</td>
<td>16</td>
<td>$16^5$</td>
<td>1,048,576</td>
</tr>
<tr>
<td>$16^2$</td>
<td>256</td>
<td>$16^6$</td>
<td>16,777,216</td>
</tr>
<tr>
<td>$16^3$</td>
<td>4096</td>
<td>$16^7$</td>
<td>268,435,456</td>
</tr>
</tbody>
</table>
Converting Decimal to Hexadecimal

<table>
<thead>
<tr>
<th>Division</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>422 / 16</td>
<td>26</td>
<td>6</td>
</tr>
<tr>
<td>26 / 16</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1 / 16</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

decimal 422 = 1A6 hexadecimal
Hexadecimal Addition

- Divide the sum of two digits by the number base (16).
- The quotient becomes the carry value, and the remainder is the sum digit.

\[
\begin{array}{cccc}
36 & 28 & 28 & 6A \\
42 & 45 & 58 & 4B \\
\hline \\
78 & 6D & 80 & B5 \\
\end{array}
\]

\[
21 / 16 = 1, \text{ rem } 5
\]

Important skill: Programmers frequently add and subtract the addresses of variables and instructions.
Hexadecimal Subtraction

- When a borrow is required from the digit to the left, add 16 (decimal) to the current digit's value:

\[
\begin{array}{c}
16 + 5 = 21 \\
\end{array}
\]

\[
\begin{array}{c|c}
C6 & 75 \\
A2 & 47 \\
\hline
24 & 2E \\
\end{array}
\]

Practice: The address of var1 is 00400020. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?
Signed Integers

- The highest bit indicates the sign.
- 1 = negative, 0 = positive

If the highest digit of a hexadecimal integer is > 7, the value is negative. Examples: 8A, C5, A2, 9D
## Forming the Two's Complement

- Bitwise NOT of the number and add 1

<table>
<thead>
<tr>
<th>Starting value</th>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: reverse the bits</td>
<td>11111110</td>
</tr>
<tr>
<td>Step 2: add 1 to the value from Step 1</td>
<td>11111110 +00000001</td>
</tr>
<tr>
<td>Sum: two’s complement representation</td>
<td>11111111</td>
</tr>
</tbody>
</table>

Note that 00000001 + 11111111 = 00000000
8-bit Two's Complement Integers

<table>
<thead>
<tr>
<th>Sign Bit</th>
<th>0 1 1 1 1 1 1 1</th>
<th>= 127</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 1 0</td>
<td>= 2</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 1</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
<td>= 0</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 1 1 1 1</td>
<td>= -1</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 1 1 0</td>
<td>= -2</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 0 0 0 1</td>
<td>= -127</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 0 0 0 0</td>
<td>= -128</td>
</tr>
</tbody>
</table>
Binary Subtraction

- When subtracting $A - B$, convert $B$ to its two's complement.
- Add $A$ to $(-B)$

$$
\begin{array}{c}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
- & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1
\end{array}
$$

Advantages for 2’s complement:
- No two 0’s
- Sign bit
- Remove the need for separate circuits for add and sub
## Ranges of Signed Integers

- The highest bit is reserved for the sign. This limits the range:

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Range (low–high)</th>
<th>Powers of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed byte</td>
<td>−128 to +127</td>
<td>−2⁷ to (2⁷ − 1)</td>
</tr>
<tr>
<td>Signed word</td>
<td>−32,768 to +32,767</td>
<td>−2¹⁵ to (2¹⁵ − 1)</td>
</tr>
<tr>
<td>Signed doubleword</td>
<td>−2,147,483,648 to 2,147,483,647</td>
<td>−2³¹ to (2³¹ − 1)</td>
</tr>
<tr>
<td>Signed quadword</td>
<td>−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807</td>
<td>−2⁶³ to (2⁶³ − 1)</td>
</tr>
</tbody>
</table>
Fractional Binary Numbers

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: \( \sum_{k=-j}^{i} b_k \cdot 2^k \)
Examples of Fractional Binary Numbers

- **Value** | **Representation**
- 5-3/4 | 101.11₂
- 2-7/8 | 10.111₁₂
- 63/64 | 0.111111₁₂

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form 0.111111...₂ just below 1.0
    - \( \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^i} + \ldots \rightarrow 1.0 \)
  - Use notation 1.0 – ε
Representable Numbers

- **Limitation**
  - Can only exactly represent numbers of the form $x \times 2^y$
  - Other numbers have repeating bit representations

- **Value** | **Representation**
  - $1/3$ | $0.0101010101[01]..._2$
  - $1/5$ | $0.001100110011[0011]..._2$
  - $1/10$ | $0.0001100110011[0011]..._2$
Converting Real Numbers

- Binary real to decimal real

$$110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375$$

- Decimal real to binary real

$$0.5625 \times 2 = 1.125$$
$$0.125 \times 2 = 0.25$$
$$0.25 \times 2 = 0.5$$
$$0.5 \times 2 = 1.0$$

The first bit = 1, second bit = 0, third bit = 0, fourth bit = 1

$$4.5625 = 100.1001_2$$
True or False

- If $x > 0$ then $x + 1 > 0$
- If $x < 0$ then $x \times 2 < 0$
- If $x > y$ then $-x < -y$
- If $x \geq 0$ then $-x \leq 0$
- If $x < 0$ then $-x > 0$
- If $x \geq 0$ then $(( !x - 1 ) \& x ) == x$
- If $x < 0$ && $y > 0$ then $x \times y < 0$
- If $x < 0$ then $((x ^ x >> 31) + 1) > 0$
Character Storage

- Character sets
  - Standard ASCII (0 – 127)
  - Extended ASCII (0 – 255)
  - ANSI (0 – 255)
  - Unicode (0 – 65,535)

- Null-terminated String
  - Array of characters followed by a null byte

- Using the ASCII table
  - back inside cover of book
Machine Words

Machine Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
    - Users can access 3GB
    - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space \( \approx 1.8 \times 10^{19} \) bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
## Data Representations

- **Sizes of C Objects (in Bytes)**

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

- Or any other pointer
Byte Ordering

- How should bytes within multi-byte word be ordered in memory?

Conventions

- Big Endian: Sun, PPC Mac
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address
**Byte Ordering Example**

- **Big Endian**
  - Least significant byte has highest address

- **Little Endian**
  - Least significant byte has lowest address

- **Example**
  - Variable $x$ has 4-byte representation $0x01234567$
  - Address given by `&x` is $0x100$

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x100$</td>
<td>$0x100$</td>
</tr>
<tr>
<td>$0x101$</td>
<td>$0x101$</td>
</tr>
<tr>
<td>$0x102$</td>
<td>$0x102$</td>
</tr>
<tr>
<td>$0x103$</td>
<td>$0x103$</td>
</tr>
<tr>
<td>$01$</td>
<td>$67$</td>
</tr>
<tr>
<td>$23$</td>
<td>$45$</td>
</tr>
<tr>
<td>$45$</td>
<td>$67$</td>
</tr>
<tr>
<td>$67$</td>
<td>$01$</td>
</tr>
</tbody>
</table>
Representing Integers

- `int A = 15213;`
- `int B = -15213;`
- `long int C = 15213;`

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>0011 1011 0110 1101</td>
<td>3B 6D</td>
</tr>
</tbody>
</table>

Two’s complement representation
Representing Strings

Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character “0” has code 0x30
    - Digit $i$ has code 0x30 + $i$
- String should be null-terminated
  - Final character = 0

Compatibility

- Byte ordering not an issue

```
char S[6] = "15213";
```
Boolean Operations

- NOT
- AND
- OR
- Operator Precedence
- Truth Tables
Boolean Algebra

- Based on **symbolic logic**, designed by George Boole
- Boolean expressions created from:
  - NOT, AND, OR

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg X )</td>
<td>NOT X</td>
</tr>
<tr>
<td>( X \land Y )</td>
<td>X AND Y</td>
</tr>
<tr>
<td>( X \lor Y )</td>
<td>X OR Y</td>
</tr>
<tr>
<td>( \neg (X \lor Y) )</td>
<td>( NOT X ) OR Y</td>
</tr>
<tr>
<td>( \neg (X \land Y) )</td>
<td>NOT ( X AND Y )</td>
</tr>
<tr>
<td>( X \land \neg Y )</td>
<td>X AND ( NOT Y )</td>
</tr>
</tbody>
</table>
Inverts (reverses) a boolean value

Truth table for Boolean NOT operator:

<table>
<thead>
<tr>
<th>x</th>
<th>( \neg x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Digital gate diagram for NOT:
AND

- Truth table for Boolean AND operator:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ∧ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Digital gate diagram for AND:
Truth table for Boolean OR operator:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ∨ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Digital gate diagram for OR:
Operator Precedence

- NOT > AND > OR
- Examples showing the order of operations:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Order of Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg X \lor Y )</td>
<td>NOT, then OR</td>
</tr>
<tr>
<td>( \neg(X \lor Y) )</td>
<td>OR, then NOT</td>
</tr>
<tr>
<td>( X \lor (Y \land Z) )</td>
<td>AND, then OR</td>
</tr>
</tbody>
</table>

- Use parentheses to avoid ambiguity
A Boolean function has one or more Boolean inputs, and returns a single Boolean output.

A truth table shows all the inputs and outputs of a Boolean function.

Example: $\neg X \lor Y$

<table>
<thead>
<tr>
<th></th>
<th>$\neg X$</th>
<th>$Y$</th>
<th>$\neg X \lor Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Truth Tables (2 of 3)

- Example: $X \land \neg Y$

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>$\neg Y$</th>
<th>$X \land \neg Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
### Truth Tables (3 of 3)

- **Example:** \((Y \land S) \lor (X \land \neg S)\)

![Two-input multiplexer diagram]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(S)</th>
<th>(Y \land S)</th>
<th>(\neg S)</th>
<th>(X \land \neg S)</th>
<th>((Y \land S) \lor (X \land \neg S))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
</tr>
<tr>
<td>(F)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
</tr>
<tr>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
</tr>
<tr>
<td>(T)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
</tr>
<tr>
<td>(F)</td>
<td>(F)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
</tr>
<tr>
<td>(F)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
<td>(T)</td>
</tr>
<tr>
<td>(T)</td>
<td>(F)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
</tr>
<tr>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
<td>(F)</td>
<td>(F)</td>
<td>(T)</td>
</tr>
</tbody>
</table>