Lecture 2: Number Representation and C Basics

CSE 30: Computer Organization and Systems Programming
Winter 2010

Notes by Ryan Kastner
Dept. of Computer Science and Engineering
University of California, San Diego
Outline

- Review concepts
- Number Representation

Reference: P+H 2.4-2.5
First, let us review concepts

- Computer programs manipulate *information* by means of computer programs.
- A computer hardware that makes this possible is called a *stored program computer*.
- A computer system designer must
  - evaluate applications for which the computer is to be used
  - build a system *architecture* from software to its hardware *organization*
  - (by architecture we often mean how the system appears to the system programmer, though it is not strictly correct..)
“Application” is a form of “problem solving”

- A machine defines a language
  - that accesses the machine functionality for the application
- A language defines a (virtual) machine
  - machine is often virtual since it is too expensive to build it directly
  - even if such a machine could be constructed no one would want it for a given higher-level language (why?)
- Application writing is a form of problem solving
  - so is micro-programming (though it is machine dependent).
Information Representation

- Information = data as characters, numbers
- Represented as binary numbers = a string of binary digits (bits)
- Value of a number is related to the length of the bit string
- Two problems:
  - only a finite numbers can be represented (ordinary arithmetic operations may not be closed)
  - how to represent negative numbers?
Decimal Numbers: Base 10

Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Example:

\[ 3271 = (3 \times 10^3) + (2 \times 10^2) + (7 \times 10^1) + (1 \times 10^0) \]
Numbers: Positional Notation

- Number Base B $\Rightarrow$ B symbols per digit:
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Base 2 (Binary): 0, 1

- Number representation:
  - $d_{31}d_{30} ... d_1d_0$ is a 32 digit number
  - value = $d_{31} \times B^{31} + d_{30} \times B^{30} + ... + d_1 \times B^1 + d_0 \times B^0$

- Binary: 0, 1 (In binary digits called “bits”)
  - 0b11010 = $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
  - = 16 + 8 + 2
  - = 26

- #s often written 0b...
  - Here 5 digit binary # turns into a 2 digit decimal #
  - Can we find a base that converts to binary easily?
Hexadecimal Numbers: Base 16

- Hexadecimal:
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Normal digits + 6 more from the alphabet
- In C, written as 0x… (e.g., 0xFAB5)

- Conversion: Binary ⇔ Hex
  - 1 hex digit represents 16 decimal values
  - 4 binary digits represent 16 decimal values
  ⇒ 1 hex digit replaces 4 binary digits

- One hex digit is a “nibble”. Two is a “byte”
  - 2 bits is a “half-nibble”. Shave and a haircut…

- Example:
  - 1010 1100 0011 (binary) = 0x_____ ?
### Decimal vs. Hexadecimal vs. Binary

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

Examples:

- \(1010\ 1100\ 0011\) (binary) = \(0x\text{AC3}\)
- \(10111\) (binary) = \(0x17\)
- \(0x3F9\) = \(11\ 1111\ 1001\) (binary)

*How do we convert between hex and Decimal?*

**MEMORIZE!**
Which base do we use?

- **Decimal**: great for humans, especially when doing arithmetic.
- **Hex**: if human looking at long strings of binary numbers, it's much easier to convert to hex and look 4 bits/symbol.
  - Terrible for arithmetic on paper.
- **Binary**: what computers use; you will learn how computers do +, -, *, /.
  - To a computer, numbers always binary.
  - Regardless of how number is written:
    - $32_{\text{ten}} == 32_{10} == 0x20 == 100000_2 == 0b100000$
    - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing.
What to do with representations of numbers?

- Just what we do with numbers!
  - Add them
  - Subtract them
  - Multiply them
  - Divide them
  - Compare them

Example: $10 + 7 = 17$
- ...so simple to add in binary that we can build circuits to do it!
- subtraction just as you would in decimal
- Comparison: How do you tell if $X > Y$?
BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters ⇒ 5 bits ($2^5 = 32$)
  - upper/lower case + punctuation ⇒ 7 bits (in 8) (“ASCII”)
  - standard code to cover all the world’s languages ⇒ 8, 16, 32 bits ("Unicode")
    www.unicode.com

- Logical values?
  - 0 ⇒ False, 1 ⇒ True

- colors? Ex:
- locations / addresses? commands?
- MEMORIZE: $N$ bits ⇔ at most $2^N$ things

Red (00)  Green (01)  Blue (11)
How to Represent Negative Numbers?

- So far, **unsigned numbers**
- Obvious solution: define leftmost bit to be sign!
  - 0 ⇒ +, 1 ⇒ –
  - Rest of bits can be numerical value of number
- Representation called **sign and magnitude**
- MIPS uses 32-bit integers. $+1_{ten}$ would be:

  0000 0000 0000 0000 0000 0000 0000 0001

- And $-1_{ten}$ in sign and magnitude would be:

  1000 0000 0000 0000 0000 0000 0000 0001
Shortcomings of sign and magnitude?

- Arithmetic circuit complicated
  - Special steps depending whether signs are the same or not
- Also, two zeros
  - $0x00000000 = +0_{\text{ten}}$
  - $0x80000000 = -0_{\text{ten}}$
  - What would two 0s mean for programming?

- Therefore sign and magnitude abandoned
Another try: complement the bits

- Example: \( 7_{10} = 00111_2 \) \(-7_{10} = 11000_2 \)
- Called One’s Complement
- Note: positive numbers have leading 0s, negative numbers have leading 1s.

\[ \begin{array}{cccc}
00000 & 00001 & \ldots & 01111 \\
10000 & \ldots & 11110 & 11111
\end{array} \]

- What is \(-00000\) ? Answer: 11111
- How many positive numbers in \(N\) bits?
- How many negative numbers?
Shortcomings of One’s complement?

- Arithmetic still a somewhat complicated.
- Still two zeros
  - $0x00000000 = +0_{\text{ten}}$
  - $0xFFFFFFFF = -0_{\text{ten}}$
- Although used for awhile on some computer products, one’s complement was eventually abandoned because another solution was better.
Binary Arithmetic Using Signed Numbers

- one’s complement requires saving the *end-around carry*
  - examples:
    - 10  1010
    - +(-3)  1100
    - +7  10110
  - two’s complement requires discarding the *end carry*
# 4-bit numbers

<table>
<thead>
<tr>
<th>String</th>
<th>S&amp;M</th>
<th>1’s</th>
<th>2’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
<td>+2</td>
<td>+2</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
<td>+3</td>
<td>+3</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
<td>+5</td>
<td>+5</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
<td>+6</td>
<td>+6</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
<td>+7</td>
<td>+7</td>
</tr>
<tr>
<td>1000</td>
<td>-0</td>
<td>-7</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-1</td>
<td>-6</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-2</td>
<td>-5</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-3</td>
<td>-4</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-5</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-6</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-7</td>
<td>-0</td>
<td>-1</td>
</tr>
</tbody>
</table>
2’s Complement Number “line”: N = 5

- $2^{N-1}$ non-negatives
- $2^{N-1}$ negatives
- one zero
- how many positives?

![Diagram showing 2's complement number line with N = 5.](image-url)
Two’s Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:
  \[ d_{31} \times -(2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0 \]

- Example: \(1101_{\text{two}}\)
  \[
  = 1 \times -(2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
  = -8 + 4 + 0 + 1 \\
  = -3_{\text{ten}}
  \]
Complement Representations

Given \( n \)-bit number, \( N \), its complement is represented as:

One’s Complement \( \overline{N} = -N + (2^n - 1) \)

Two’s Complement \( \tilde{N} = -N + 2^n \)

Examples: Representation of -7.

1’s complement: \( (2^4-1)-7 = (10000 - 0001) - 0111 = (01111) - 0111 = 01000 \)
2’s complement: \( (2^4-7) = 10000 - 0111 = 01001 \)
Two’s Complement shortcut: Negation

- Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result
- Proof*: Sum of number and its (one’s) complement must be 111...111\text{two}
  
  However, 111...111\text{two} = -1\text{ten}

Let \( x’ \) ⇒ one’s complement representation of \( x \)

Then \( x + x’ = -1 \Rightarrow x + x’ + 1 = 0 \Rightarrow -x = x’ + 1 \)

- Example: -3 to +3 to -3

<table>
<thead>
<tr>
<th>( x )</th>
<th>1111 1111 1111 1111 1111 1111 1111 1101\text{two}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x’ )</td>
<td>0000 0000 0000 0000 0000 0000 0000 0010\text{two}</td>
</tr>
<tr>
<td>+1</td>
<td>0000 0000 0000 0000 0000 0000 0000 0011\text{two}</td>
</tr>
<tr>
<td>()’</td>
<td>1111 1111 1111 1111 1111 1111 1111 1100\text{two}</td>
</tr>
<tr>
<td>+1</td>
<td>1111 1111 1111 1111 1111 1111 1111 1101\text{two}</td>
</tr>
</tbody>
</table>

You should be able to do this in your head…
Two’s comp. shortcut: Sign extension

- Convert 2’s complement number rep. using n bits to more than n bits
- Simply replicate the most significant bit (sign bit) of smaller to fill new bits
  - 2’s comp. positive number has infinite 0s
  - 2’s comp. negative number has infinite 1s
  - Binary representation hides leading bits; sign extension restores some of them
- 16-bit $-4_{ten}$ to 32-bit:
  
  \[
  \begin{array}{cccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0
  \end{array}
  \]
What if too big?

- Binary bit patterns above are simply representatives of numbers. Strictly speaking they are called “numerals”.
- Numbers really have an ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don’t normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, overflow is said to have occurred.
Recap: Arithmetic Using S&M Numbers

- Sign & Magnitude Addition
  - same sign operations: add the magnitudes
  - different sign operations: subtract the smaller magnitude from the larger one, copy the sign of the number with larger magnitude.

- Sign & Magnitude Subtraction
  - same as addition by using negative of the subtracted number.
Arithmetic using 1’s Complement

- Convert negative numbers into 1’s comp representation
- Perform ADDition operations.
- Addition of two numbers may lead to a carry out of the MSBs
  - in such cases, correct answer is obtained by adding the carry-out into the LSB of the result
- **End-around carry**
- Examples.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>+(-3)</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>-7</td>
<td>10111</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
End-Around Carry

- Equivalent of subtracting $2^n$ and adding 1.

\[
M - N = M + \overline{N} = M + (2^n - 1 - N)
\]

\[
= (M - N) + (2^n - 1)
\]
2’s Complement Arithmetic

- Similar to 1’s complement except that
  - No end-around carry
  - Instead, Ignore the carry-out of MSB.
  - Ignoring carry-out is similar to subtracting $2^n$
- Simple addition, however, negation requires bit-wise complementation and an addition.
- In contrast, 1’s complement, negation is simple but addition requires end-around carry.
Overflow & Underflow

- When a resulting number requires more bits to represent the number correctly.
- Recall, in 2’s complement we drop the carry out of the MSB (in case there is one)
- However, if there is no carry into the MSB but there is a carry out, this means there is an underflow (two negative numbers added, but the result is negative).
- Similarly, if there is no carry out of MSB but there is a carry into the MSB, then there is overflow (two positive numbers added, but the result a negative number).
Decimal Coding

- Representation of decimal numbers using binary digits
  - Each decimal number is represented by a binary string. For example: 6 = 0110
- The coded form can be used to directly implement the decimal operations.
- Need a minimum of 4 bits to represent 10 decimal digits.
BCD Coding & Operations

(185)_{10} = (0001 1000 0101)_{BCD} = (10111001)_{2}

\[
\begin{array}{ccccccc}
4 & 0100 & +5 & 0101 & 12 & 1100 & +9 & 1001 \\
+8 & 1000 & +8 & 1000 & 17 & 10001 & 9 & 0110 \\
--- & ------ & --- & ------ & --- & ------ & --- & ------ \\
+9 & 1001 & +9 & 1001 & 17 & 10001 & +0110 & + 0110 \\
--- & ------ & --- & ------ & --- & ------ & --- & ------ \\
\end{array}
\]

1 0010

1 0111

Question: When is a correction needed? Why does addition by 6 work? Hint: 16 - 10 = 6.
Example

184 0001 1000 0100
+576 0101 0111 0110
------ ------ ------ ------
1010
Correct +0110
------

1 0000
------

1 0000
Correct + 0110
------
0110
------
0111
ASCII Coding

- Used to represent *alpha-numeric* characters
  - plus some *control characters*

- Commonly used as 7-bit codes
  - 128 characters

- Byte-wise encoding leaves room for 1 parity bit
Signed vs. Unsigned Variables

- Java and C declare integers `int`
  - Use two’s complement (signed integer)
- Also, C declaration `unsigned int`
  - Declares a `unsigned` integer
  - Treats 32-bit number as unsigned integer, so most significant bit is part of the number, not a sign bit
Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

physics.nist.gov/cuu/Units/binary.html

- Common use prefixes (all SI, except K [= k in SI])

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr</th>
<th>Factor</th>
<th>SI size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kilo</td>
<td>K</td>
<td>$2^{10} = 1,024$</td>
<td>$10^3 = 1,000$</td>
</tr>
<tr>
<td>Mega</td>
<td>M</td>
<td>$2^{20} = 1,048,576$</td>
<td>$10^6 = 1,000,000$</td>
</tr>
<tr>
<td>Giga</td>
<td>G</td>
<td>$2^{30} = 1,073,741,824$</td>
<td>$10^9 = 1,000,000,000$</td>
</tr>
<tr>
<td>Tera</td>
<td>T</td>
<td>$2^{40} = 1,099,511,627,776$</td>
<td>$10^{12} = 1,000,000,000,000$</td>
</tr>
<tr>
<td>Peta</td>
<td>P</td>
<td>$2^{50} = 1,125,899,906,842,624$</td>
<td>$10^{15} = 1,000,000,000,000,000$</td>
</tr>
<tr>
<td>Exa</td>
<td>E</td>
<td>$2^{60} = 1,152,921,504,606,846,976$</td>
<td>$10^{18} = 1,000,000,000,000,000,000$</td>
</tr>
<tr>
<td>Zetta</td>
<td>Z</td>
<td>$2^{70} = 1,180,591,620,717,411,303,424$</td>
<td>$10^{21} = 1,000,000,000,000,000,000,000$</td>
</tr>
<tr>
<td>Yotta</td>
<td>Y</td>
<td>$2^{80} = 1,208,925,819,614,629,174,706,176$</td>
<td>$10^{24} = 1,000,000,000,000,000,000,000,000$</td>
</tr>
</tbody>
</table>

- Confusing! Common usage of “kilobyte” means 1024 bytes, but the “correct” SI value is 1000 bytes
- Hard Disk manufacturers & Telecommunications are the only computing groups that use SI factors, so what is advertised as a 30 GB drive will actually only hold about $28 \times 2^{30}$ bytes, and a 1 Mbit/s connection transfers $10^6$ bps.
kibi, mebi, gibi, tebi, pebi, exbi, zebi, yobi

en.wikipedia.org/wiki/Binary_prefix

- New IEC Standard Prefixes [only to exbi officially]

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>kibi</td>
<td>Ki</td>
<td>$2^{10} = 1,024$</td>
</tr>
<tr>
<td>mebi</td>
<td>Mi</td>
<td>$2^{20} = 1,048,576$</td>
</tr>
<tr>
<td>gibi</td>
<td>Gi</td>
<td>$2^{30} = 1,073,741,824$</td>
</tr>
<tr>
<td>tebi</td>
<td>Ti</td>
<td>$2^{40} = 1,099,511,627,776$</td>
</tr>
<tr>
<td>pebi</td>
<td>Pi</td>
<td>$2^{50} = 1,125,899,906,842,624$</td>
</tr>
<tr>
<td>exbi</td>
<td>Ei</td>
<td>$2^{60} = 1,152,921,504,606,846,976$</td>
</tr>
<tr>
<td>zebi</td>
<td>Zi</td>
<td>$2^{70} = 1,180,591,620,717,411,303,424$</td>
</tr>
<tr>
<td>yobi</td>
<td>Yi</td>
<td>$2^{80} = 1,208,925,819,614,629,174,706,176$</td>
</tr>
</tbody>
</table>

As of this writing, this proposal has yet to gain widespread use…

- International Electrotechnical Commission (IEC) in 1999 introduced these to specify binary quantities.
- Names come from shortened versions of the original SI prefixes (same pronunciation) and bi is short for “binary”, but pronounced “bee” :-(
- Now SI prefixes only have their base-10 meaning and never have a base-2 meaning.
Logical Operators

- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1

- In general, can define them to accept >2 inputs, but in the case of MIPS assembly, both of these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware
Logical Operators

- Truth Table: standard table listing all possible combinations of inputs and resultant output for each.

- Truth Table for AND and OR

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Uses for Logical Operators

- Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.
- This can be used to create a mask.
  - Example:

    | 1011 0110 1010 0100 0011 1101 1001 1010 |
    | 0000 0000 0000 0000 0000 1111 1111 1111 |

  - The result of anding these:

    | 0000 0000 0000 0000 0000 1101 1001 1010 |

    mask last 12 bits
Uses for Logical Operators

- Similarly, note that or ing a bit with 1 produces a 1 at the output while or ing a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, 0x12345678 OR 0x0000FFF results in 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).
Summary

- We represent “things” in computers as particular bit patterns: \( N \) bits \( \Rightarrow 2^N \) things.

- Decimal for human calculations, binary for computers, hex to write binary more easily.

- 1’s complement - mostly abandoned.

- 2’s complement universal in computing: cannot avoid, so learn.

- Overflow: numbers \( \infty \); computers finite, errors!