Review

° We represent “things” in computers as particular bit patterns: $N$ bits $\Rightarrow 2^N$
  * numbers, characters, ...
° Decimal for human calculations, binary to understand computers, hexadecimal to understand binary
° 2’s complement universal in computing: cannot avoid, so learn
° Computer operations on the representation correspond to real operations on the real thing

2’s Complement Number “line”: $N = 5$

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

Two’s Complement for $N=32$

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Two's Complement Formula

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

- Example: 1111 1100\text{two}
  \[ = 1 \times -2^7 + 1 \times 2^8 + 1 \times 2^7 + ... + 1 \times 2^1 + 0 \times 2^0 \]
  \[ = -128 + 64 + 32 + 16 + 8 + 4 \]
  \[ = -128 + 124 \]
  \[ = -4_{ten} \]

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_{ten}
  
  Let \( x' \) ⇒ one’s complement representation of \( x \)
  
  Then \( x + x' = -1 \) ⇒ \( x + x' + 1 = 0 \) ⇒ \( x' + 1 = -x \)

- Example: -4 to +4 to -4\text{x}:
  \[
  \begin{align*}
  \text{x:} & & 1111 1111 1111 1111 1111 1111 1111 1100 \text{two} \\
  \text{x':} & & 0000 0000 0000 0000 0000 0000 0000 0011 \text{two} \\
  +1: & & 0000 0000 0000 0000 0000 0000 0000 0100 \text{two} \\
  ()': & & 1111 1111 1111 1111 1111 1111 1111 1011 \text{two} \\
  +1: & & 1111 1111 1111 1111 1111 1111 1111 1100 \text{two}
  \end{align*}
  \]

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\text{ten} to 32-bit:
    \[
    \begin{array}{cccccccc}
    1111 & 1111 & 1111 & 1111 & 1100_{\text{two}} \\
    \end{array}
    \]

Signed vs. Unsigned Variables

- Java just declares integers \text{int}
  - Uses two’s complement

- C has declaration \text{int} also
  - Declares variable as a signed integer
  - Uses two’s complement

- Also, C declaration \text{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
Numbers represented in memory

- Memory is a place to store bits
- A word is a fixed number of bits (e.g., 32) at an address
- Addresses are naturally represented as unsigned numbers in C

Signed v. Unsigned Comparisons

- X = 1111 1111 1111 1111 1111 1111 1100\text{two}
- Y = 0011 1011 1001 1010 1000 1010 0000 0000\text{two}

- Is X > Y?
  - unsigned: YES
  - signed: NO

What if too big?

- Binary bit patterns above are simply representatives of numbers. Strictly speaking they are called “numerals”.
- Numbers really have an infinite 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.

And in Conclusion...

- We represent “things” in computers as particular bit patterns: N bits \( \Rightarrow 2^N \)
  - numbers, characters, ...
- Decimal for human calculations, binary to understand computers, hexadecimal to understand binary
- 2’s complement universal in computing: cannot avoid, so learn
- Computer operations on the representation correspond to real operations on the real thing
- Overflow: numbers infinite but computers finite, so errors occur
**Administrivia: Who gets lab seat?**

**Rank order of seating priority**

1. 61c registered for that section
2. 61c registered for another section
3. 61c waitlisted for that section
4. 61c waitlisted for another section
5. Concurrent enrollment

**Administrivia: You have a question?**

**Tips on getting an answer to your question:**

1. Ask a classmate
2. Ask Brian/John after or before lecture
3. The newsgroup, ucb.class.cs61c
   a) Read it: Has your Q been answered already?
   b) If not, ask it and check back
4. Ask TA in section, lab or OH
5. Ask Brian/John in OH
6. Ask Brian/John in lecture
7. Send your TA email
8. Send Brian/John email

**Administrivia: Near term**

° Get cardkeys from CS main office  
  Soda Hall 3rd floor
° Reading for weekend:  
  - K&R Ch 5 and 6
° Monday holiday
° Wed lecture  
  - C pointers and arrays in detail
° HW  
  - HW1 due Wednesday 9am.
  - HW2 will be posted soon.

**Disclaimer**

° **Important:** You will not learn how to fully code in C in these lectures!  
  You’ll still need your C reference for this course.
  
  - K&R is a must-have reference.
  - Check online for more sources.
  
  - “JAVA in a Nutshell,” O’Reilly.
  - Chapter 2, “How Java Differs from C”.
Compilation: Overview

C compilers take C and convert it into an architecture specific machine code (string of 1s and 0s).
- Unlike Java which converts to architecture independent bytecode.
- Unlike most Scheme environments which interpret the code.
- Generally a 2 part process of compiling .c files to .o files, then linking the .o files into executables.

Compilation: Advantages

° Great run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
° OK compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled

Compilation: Disadvantages

° All compiled files (including the executable) are architecture specific, depending on both the CPU type and the operating system.
° Executable must be rebuilt on each new system.
  - Called “porting your code” to a new architecture.
° The “change-compile-run [repeat]” iteration cycle is slow

C vs. Java™ Overview (1/2)

Java
- Object-oriented (OOP)
- “Methods”
- Class libraries of data structures
- Automatic memory management

C
- No built-in object abstraction. Data separate from methods.
- “Functions”
- C libraries are lower-level
- Manual memory management
- Pointers
C vs. Java™ Overview (2/2)

Java
- High memory overhead from class libraries
- Relatively Slow
- Arrays initialize to zero
- Syntax: //comment
  System.out.print

C
- Low memory overhead
- Relatively Fast
- Arrays initialize to garbage
- Syntax: /* comment */
  printf

C Syntax: Variable Declarations
- Very similar to Java, but with a few minor but important differences
- All variable declarations must go before they are used (at the beginning of the block).
- A variable may be initialized in its declaration.
- Examples of declarations:
  - correct: 
    int a = 0, b = 10;
    ...
  - incorrect: for (int i = 0; i < 10; i++)

C Syntax: True or False?
- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (pointer: more on this later)
  - no such thing as a Boolean
- What evaluates to TRUE in C?
  - everything else...
  - (same idea as in scheme: only #f is false, everything else is true!)

C syntax : flow control
- Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
  - if-else
  - switch
  - while and for
  - do-while
C Syntax: main

° To get the main function to accept arguments, use this:
  ```c
  int main (int argc, char *argv[])
  ```
° What does this mean?
  • argc will contain the number of strings on the command line (the executable counts as one, plus one for each argument).
    - Example: `unix% sort myFile`
  • argv is a pointer to an array containing the arguments as strings (more on pointers later).

Address vs. Value

° Consider memory to be a single huge array:
  • Each cell of the array has an address associated with it.
  • Each cell also stores some value.
° Don’t confuse the address referring to a memory location with the value stored in that location.

Pointers

° An address refers to a particular memory location. In other words, it points to a memory location.
° Pointer: A variable that contains the address of a variable.

How to create a pointer:

& operator: get address of a variable

```c
int *x, y;  
x = &y;
```  

Note the `*` gets used 2 different ways in this example. In the declaration to indicate that `x` is going to be a pointer, and in the `printf` to get the value pointed to by `x`.

How get a value pointed to?

* "dereference operator": get value pointed to

```c
printf("%d\n", *x);
```
**Pointers**

- How to change a variable pointed to?
  - Use dereference * operator on left of =

  \[ \text{x} \quad \text{y} \quad 3 \]

  \[ \text{*x} = 5; \quad \text{x} \quad \text{y} \quad 5 \]

**Pointers and Parameter Passing**

- Java and C pass a parameter “by value”
  - procedure/function gets a copy of the parameter, so changing the copy cannot change the original

```c
void addOne (int x) {
    x = x + 1;
}
int y = 3;
addOne(y);
```

- *y is still = 3*

**Pointers**

- Normally a pointer can only point to one type (int, char, a struct, etc.).
  - `void *` is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs!

```c
void addOne (int *x) {
    *x = *x + 1;
}
int y = 3;
addOne(&y);
```

- *y is now = 4*
And in conclusion…

- All declarations go at the beginning of each function.
- Only 0 and NULL evaluate to FALSE.
- All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
- A pointer is a C version of the address.
  - * “follows” a pointer to its value
  - & gets the address of a value