

#### The Design of C: A Rational Reconstruction

Jennifer Rexford

#### **Goals of this Lecture**



- Number systems
  - Binary numbers
  - Finite precision
  - Binary arithmetic
  - Logical operators
- Design rationale for C
  - Decisions available to the designers of C
  - Decisions made by the designers of C



# **Number Systems**

## Why Bits (Binary Digits)?



- Computers are built using digital circuits
  - · Inputs and outputs can have only two values
  - True (high voltage) or false (low voltage)
  - Represented as 1 and 0
- Can represent many kinds of information
  - Boolean (true or false)
  - Numbers (23, 79, ...)
  - Characters ('a', 'z', ...)
  - Pixels, sounds
  - Internet addresses
- Can manipulate in many ways
  - Read and write
  - Logical operations
  - Arithmetic

#### Base 10 and Base 2



- Decimal (base 10)
  - Each digit represents a power of 10
  - **4173** = **4** x  $10^3$  + **1** x  $10^2$  + **7** x  $10^1$  + **3** x  $10^0$
- Binary (base 2)
  - Each bit represents a power of 2
  - 10110 = 1 x  $2^4$  + 0 x  $2^3$  + 1 x  $2^2$  + 1 x  $2^1$  + 0 x  $2^0$  = 22

Convert decimal to binary: divide by 2, keep remainders

12/2 = 6 R = 0 6/2 = 3 R = 0 3/2 = 1 R = 1 1/2 = 0 R = 1 Result = 1100



## Writing Bits is Tedious for People

- Octal (base 8)
  - Digits 0, 1, ..., 7
- Hexadecimal (base 16)
  - Digits 0, 1, ..., 9, A, B, C, D, E, F

0000 = 0	1000 = 8
0001 = 1	1001 = 9
0010 = 2	1010 = A
0011 = 3	1011 = B
0100 = 4	1100 = C
0101 = 5	1101 = D
0110 = 6	1110 = E
0111 = 7	1111 = F

Thus the 16-bit binary number

**1011 0010 1010 1001** 

converted to hex is

**B2A9** 



# **Representing Colors: RGB**

- Three primary colors
  - Red
  - Green
  - Blue
- Strength
  - 8-bit number for each color (e.g., two hex digits)
  - So, 24 bits to specify a color
- In HTML, e.g. course "Schedule" Web page
  - Red: <span style="color:#FF0000">De-Comment Assignment Due</span>
  - Blue: <span style="color:#0000FF">Reading Period</span>
- Same thing in digital cameras
  - Each pixel is a mixture of red, green, and blue



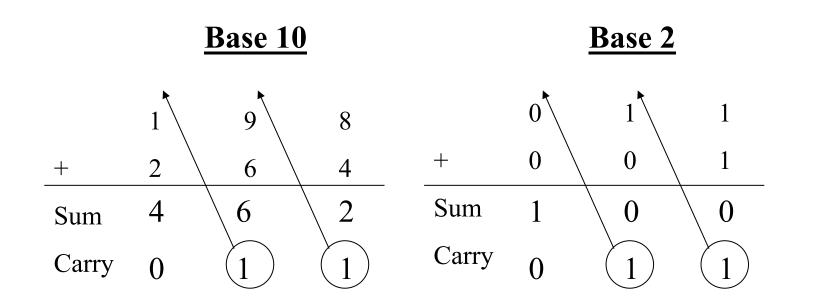
### **Finite Representation of Integers**

- Fixed number of bits in memory
  - Usually 8, 16, or 32 bits
  - (1, 2, or 4 bytes)
- Unsigned integer
  - No sign bit
  - Always 0 or a positive number
  - All arithmetic is modulo 2<sup>n</sup>
- Examples of unsigned integers
  - 00000001 **→** 1
  - 00001111 **→** 15
  - 00010000 **→** 16
  - 00100001 **→** 33
  - 11111111 → 255

### **Adding Two Integers**

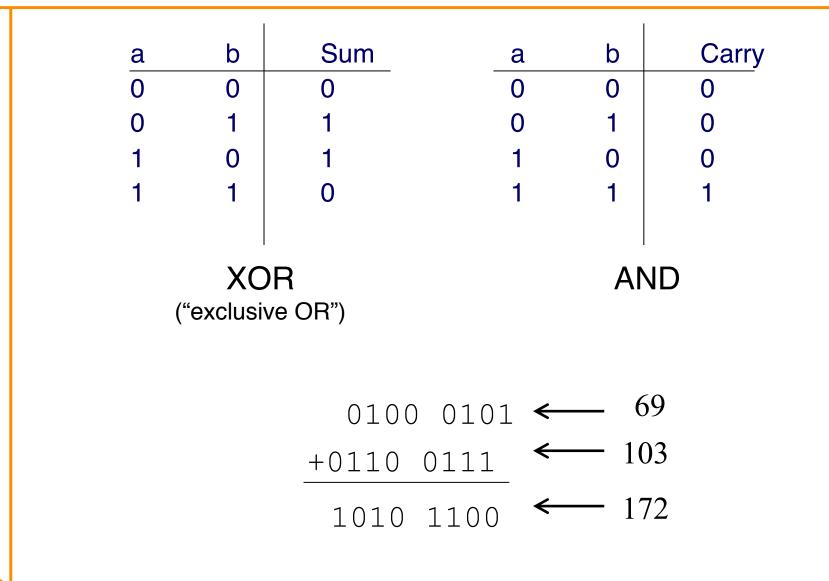


- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column



#### **Binary Sums and Carries**





#### **Modulo Arithmetic**



- Consider only numbers in a range
  - E.g., five-digit car odometer: 0, 1, ..., 99999
  - E.g., eight-bit numbers 0, 1, ..., 255
- Roll-over when you run out of space
  - E.g., car odometer goes from 99999 to 0, 1, ...
  - E.g., eight-bit number goes from 255 to 0, 1, ...
- Adding 2<sup>n</sup> doesn't change the answer
  - For eight-bit number, n=8 and 2<sup>n</sup>=256
  - E.g., (37 + 256) mod 256 is simply 37
- This can help us do subtraction...
  - a b: equals a + (256 -1 b) + 1



# **One's and Two's Complement**

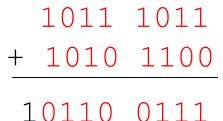
- One's complement: flip every bit
  - E.g., b is 01000101 (i.e., 69 in decimal)
  - One's complement is 10111010
  - That's simply 255-69
- Subtracting from 11111111 is easy (no carry needed!)

$$\frac{\begin{array}{c}1111 \\ - \\ 0100 \\ 0101\end{array}}{\begin{array}{c}1011 \\ 1010\end{array}} \xleftarrow{} b$$

- Two's complement
  - Add 1 to the one's complement
  - E.g., (255 69) + 1 → 1011 1011

### **Putting it All Together**

- Computing "a b"
  - Same as "a + 256 b"
  - Same as "a + (255 b) + 1"
  - Same as "a + onesComplement(b) + 1"
  - Same as "a + twosComplement(b)"
- Example: 172 69
  - The original number 69: 0100 0101
  - One's complement of 69: 1011 1010
  - Two's complement of 69: 1011 1011
  - Add to the number 172: 1010 1100
  - The sum comes to: 0110 0111
  - Equals: **103** in decimal





### **Signed Integers**

- Sign-magnitude representation
  - Use one bit to store the sign
    - Zero for positive number
    - One for negative number
  - Examples
    - E.g., 0010 1100 → 44
    - E.g., 1010 1100 → -44
  - · Hard to do arithmetic this way, so it is rarely used
- Complement representation
  - One's complement
    - Flip every bit
    - E.g., 1101 0011 → -44
  - Two's complement
    - Flip every bit, then add 1
    - E.g., 1101 0100 → -44



# **Overflow: Running Out of Room**



- Adding two large integers together
  - Sum might be too big for the number of bits available
  - What happens?
- Unsigned integers
  - All arithmetic is "modulo" arithmetic
  - Sum would just wrap around
- Signed integers
  - Can get nonsense values
  - Example with 16-bit integers
    - Sum: 10000+20000+30000
    - Result: -5536

## **Bitwise Operators: AND and OR**

• Bitwise AND (&)

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

- Mod on the cheap!
  - E.g., 53 % 16
  - ... is same as 53 & 15;

T



- & 15 0 0 0 ()
  - 5 0 | 0 |0 0 ()



	0	1
0	0	1
1	1	1



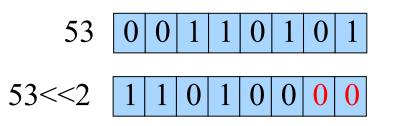
# **Bitwise Operators: Not and XOR**

- One's complement (~)
  - Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    - x = x & ~7;
- XOR (^)
  - 0 if both bits are the same
  - 1 if the two bits are different

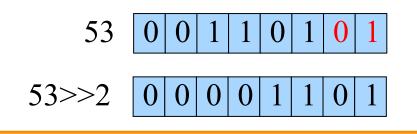
## **Bitwise Operators: Shift Left/Right**



- Shift left (<<): Multiply by powers of 2</li>
  - Shift some # of bits to the left, filling the blanks with 0



- Shift right (>>): Divide by powers of 2
  - Shift some # of bits to the right
  - For unsigned integer, fill in blanks with 0
  - What about signed negative integers?
    - · Can vary from one machine to another!



### **Example: Counting the 1's**



- How many 1 bits in a number?
  - E.g., how many 1 bits in the binary representation of 53?



- Four 1 bits
- How to count them?
  - · Look at one bit at a time
  - Check if that bit is a 1
  - Increment counter
- How to look at one bit at a time?
  - Look at the last bit: n & 1
  - Check if it is a 1: (n & 1) == 1, or simply (n & 1)



# **Counting the Number of '1' Bits**

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  unsigned int n;
  unsigned int count;
  printf("Number: ");
  if (scanf("%u", &n) != 1) {
      fprintf(stderr, "Error: Expect unsigned int.\n");
     exit(EXIT FAILURE);
   for (count = 0; n > 0; n >>= 1)
      count += (n \& 1);
  printf("Number of 1 bits: %u\n", count);
  return 0;
```

### Summary



- Computer represents everything in binary
  - Integers, floating-point numbers, characters, addresses, ...
  - Pixels, sounds, colors, etc.
- Binary arithmetic through logic operations
  - Sum (XOR) and Carry (AND)
  - Two's complement for subtraction
- Bitwise operators
  - AND, OR, NOT, and XOR
  - Shift left and shift right
  - · Useful for efficient and concise code, though sometimes cryptic



# The Design of C

#### Goals of C



Designers wanted C to support:

- Systems programming
  - Development of Unix OS
  - Development of Unix programming tools

But also:

- Applications programming
  - · Development of financial, scientific, etc. applications

Systems programming was the primary intended use

#### The Goals of C (cont.)



#### The designers of wanted C to be:

- Low-level
  - Close to assembly/machine language
  - Close to hardware

#### But also:

- Portable
  - Yield systems software that is easy to port to differing hardware

#### The Goals of C (cont.)

#### The designers wanted C to be:

- Easy for **people** to handle
  - Easy to understand
  - Expressive
    - High (functionality/sourceCodeSize) ratio

#### But also:

- Easy for **computers** to handle
  - Easy/fast to compile
  - Yield efficient machine language code

#### Commonality:

Small/simple

#### **Design Decisions**



In light of those goals...

- What design decisions did the designers of C have?
- What design decisions did they make?

Consider programming language features,

from simple to complex...

#### Feature 1: Data Types



- Previously in this lecture:
  - · Bits can be combined into bytes
  - Our interpretation of a collection of bytes gives it meaning
    - A signed integer, an unsigned integer, a RGB color, etc.
- Data type: well-defined interpretation of collection of bytes
- A high-level language should provide primitive data types
  - Facilitates abstraction
  - Facilitates manipulation via associated well-defined operators
  - Enables compiler to check for mixed types, inappropriate use of types, etc.

### **Primitive Data Types**

- Thought process
  - C should handle:
    - Integers
    - Characters
    - Character strings
    - Logical (alias Boolean) data
    - Floating-point numbers
  - C should be small/simple

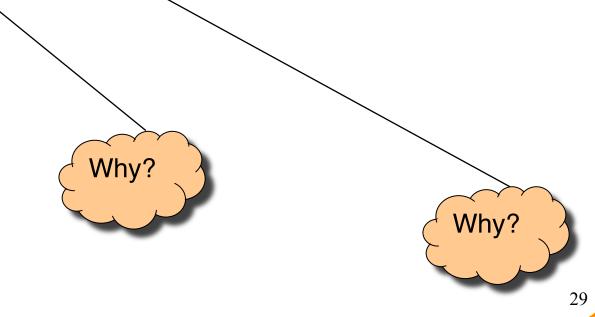
#### Decisions

- Provide integer, character, and floating-point data types
- Do not provide a character string data type (More on that later)
- Do not provide a logical data type (More on that later)

#### **Integer Data Types**



- Thought process
  - For flexibility, should provide integer data types of various sizes
  - For portability at application level, should specify size of each data type
  - For portability at systems level, should define integral data types in terms of natural word size of computer
  - Primary use will be systems programming



### Integer Data Types (cont.)



- Decisions
  - Provide three integer data types: short, int, and long
  - Do not specify sizes; instead:
    - int is natural word size
    - 2 <= bytes in **short** <= bytes in **int** <= bytes in **long**
- Incidentally, on hats using gcc217
  - Natural word size: 4 bytes
  - short: 2 bytes
  - int: 4 bytes
  - long: 4 bytes

#### **Integer Constants**



Was that a good

decision?

- Thought process
  - People naturally use decimal
  - Systems programmers often use binary, octal, hexadecimal
- Decisions
  - · Use decimal notation as default
  - Use "0" prefix to indicate octal notation<sup>4</sup>
  - Use "0x" prefix to indicate hexadecimal notation
  - Do not allow binary notation; too verbose, error prone
  - Use "L" suffix to indicate long constant
  - Do not use a suffix to indicate short constant; instead must use cast .

#### • Examples

- int: 123, -123, 0173, 0x7B
- long: 123L, -123L, 0173L, 0x7BL
- short: (short) 123, (short) -123, (short) 0173, (short) 0x7B

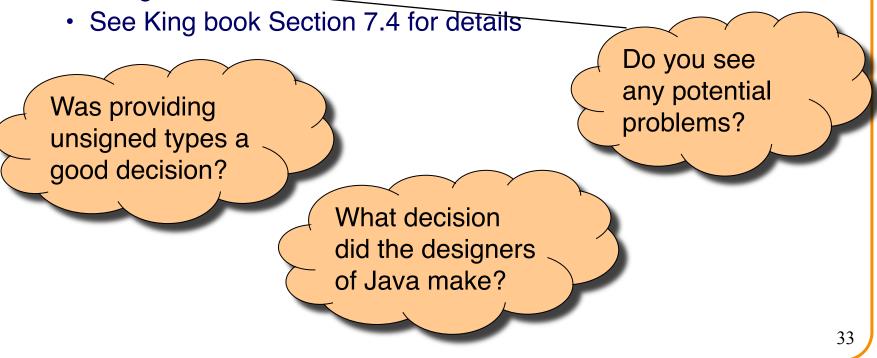


# **Unsigned Integer Data Types**

- Thought process
  - Must represent positive and negative integers
    - Signed types are essential
  - Unsigned data can be twice as large as signed data
    - Unsigned data could be useful
  - Unsigned data are good for bit-level operations
    - Bit-level operations are common in systems programming
  - Implementing both signed and unsigned data types is complex
    - Must define behavior when an expression involves both

# Unsigned Integer Data Types (cont.)

- Decisions
  - Provide unsigned integer types: unsigned short, unsigned int, and unsigned long
  - Conversion rules in mixed-type expressions are complex
    - Generally, mixing signed and unsigned converts signed to unsigned



### **Unsigned Integer Constants**



- Thought process
  - "L" suffix distinguishes long from int; also could use a suffix to distinguish signed from unsigned
  - Octal or hexadecimal probably are used with bit-level operators

#### Decisions

- Default is signed
- Use "U" suffix to indicate unsigned
- Integers expressed in octal or hexadecimal automatically are unsigned

#### Examples

- unsigned int: 123U, 0173, 0x7B
- unsigned long: 123UL, 0173L, 0x7BL
- unsigned short: (short)123U, (short)0173, (short)0x7B

