Binary Numbers

COS 217

Goals of Today's Lecture

- Binary numbers
  - Why binary?
  - Converting base 10 to base 2
  - Octal and hexadecimal
- Integers
  - Unsigned integers
  - Integer addition
  - Signed integers
- C bit operators
  - And, or, not, and xor
  - Shift-left and shift-right
  - Function for counting the number of 1 bits
  - Function for XOR encryption of a message

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
    - Sound
- Can manipulate in many ways
    - Read and write
    - Logical operations
    - Arithmetic
    - …

Base 10 and Base 2

- Base 10
  - Each digit represents a power of 10
  - $4173 = 4 \times 10^3 + 1 \times 10^2 + 7 \times 10^1 + 3 \times 10^0$
- Base 2
  - Each bit represents a power of 2
  - $10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22$

Divide repeatedly by 2 and 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
  - In C: 00, 01, …, 07
- Hexadecimal (base 16)
  - Digits 0, 1, …, 9, A, B, C, D, E, F
  - In C: 0x0, 0x1, …, 0xf

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

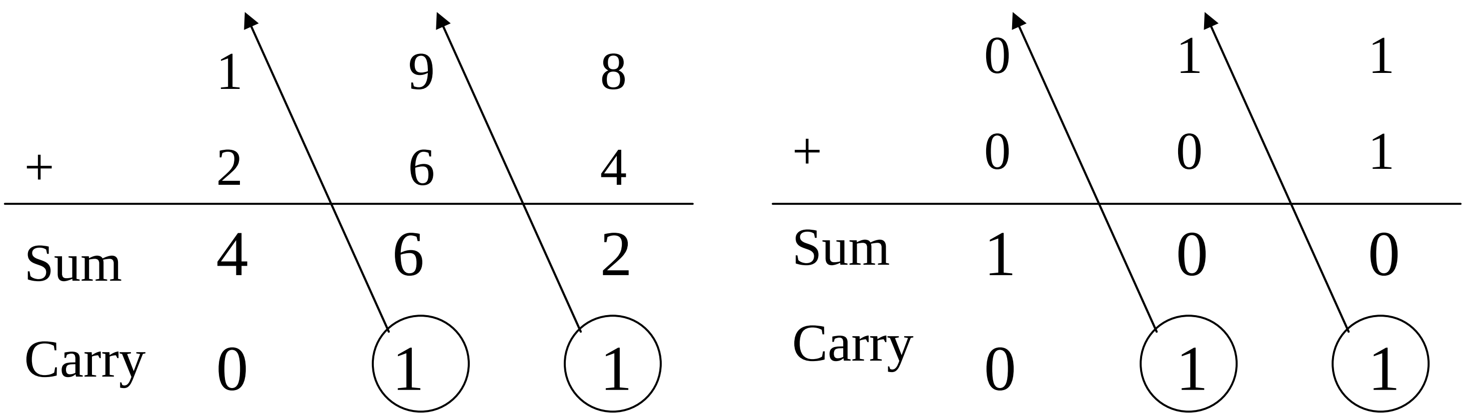
- Three primary colors
  - Red
  - Green
  - Blue
- Strength
  - 8-bit number for each color (e.g., two hex digits)
  - So, 24 bits to specify a color
- In HTML, on the course Web page
  - Red: <i>Symbol Table Assignment Due</i>
  - Blue: <i>Fall Recess</i>
- Same thing in digital cameras
  - Each pixel is a mixture of red, green, and blue

Storing Integers on the Computer

- Fixed number of bits in memory
  - Short: usually 16 bits
  - Int: 16 or 32 bits
  - Long: 32 bits
- Unsigned integer
  - No sign bit
  - Always positive or 0
  - All arithmetic is modulo 2n
- Example of unsigned int
  - 00000001 → 1
  - 00001111 → 15
  - 00010000 → 16
  - 00100001 → 33
  - 11111111 → 255

Adding Two Integers: Base 10

- 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

a	b	Sum
0	0	0
0	1	1
1	0	1
1	1	0

XOR

a	b	Carry
0	0	0
0	1	0
1	0	0
1	1	1

AND

```
  0100 0101   ←  69
+ 0110 0111   ←  103
-----------
  1010 1100   ←  172
```

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^n doesn't change the answer
  - For eight-bit number, n=8 and 2^n=256
  - E.g., (37 + 256) mod 256 is simply 37
- This can help us do subtraction…
  - Suppose you want to compute a – b
  - Note that this 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 base 10)
  - One’s complement is 10111010
  - That’s simply 255-69
- Subtracting from 11111111 is easy (no carry needed!)

```
  1111 1111
- 0100 0101   ←— b
-----------
  1011 1010   ←— one’s complement
```

- Two’s complement
  - Add 1 to the one’s complement
  - E.g., (255 – 69) + 1 ➔ 1011 1011

Putting it All Together

- Computing "a – b" for unsigned integers
  - Same as "a + 256 – b"
  - Same as "a + (255 – b) + 1"
  - Same as "a + onecomplement(b) + 1"
  - Same as "a + twocomplement(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 base 10

```
  1010 1100
+ 1011 1011
-----------
1 0110 0111
```

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 large to store in the number of bits allowed
  - What happens?
- Unsigned numbers
  - 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
  - In this case, fixable by using “long”…

Bitwise Operators: AND and OR

- Bitwise AND (&)

&	0	1
0	0	0
1	0	1

  - Mod on the cheap!
    - E.g., h = 53 & 15;

53	0	0	1	1	0	1	0	1
& 15	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	0	1

- Bitwise OR (|)

|	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

^	0	1
0	0	1
1	1	0

Bitwise Operators: Shift Left/Right

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

53	0	0	1	1	0	1	0	0
53<<2	1	1	0	1	0	0	0	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 integers? Varies across machines…
      - Can vary from one machine to another!

53	0	0	1	1	0	1	0	0
53>>2	0	0	0	0	1	1	0	1

Count Number of 1s in an Integer

- Function bitcount(unsigned x)
  - Input: unsigned integer
  - Output: number of bits set to 1 in the binary representation of x
- Main idea
  - Isolate the last bit and see if it is equal to 1
  - Shift to the right by one bit, and repeat

```
int bitcount(unsigned x) {
    int b;
    for (b = 0; x != 0; x >>= 1)
        if (x & 1)
            b++;
    return b;
}
```

XOR Encryption

- Program to encrypt text with a key
  - Input: original text in stdin
  - Output: encrypted text in stdout
- Use the same program to decrypt text with a key
  - Input: encrypted text in stdin
  - Output: original text in stdout
- Basic idea
  - Start with a key, some 8-bit number (e.g., 0110 0111)
  - Do an operation that can be inverted
    - E.g., XOR each character with the 8-bit number

```
    0100 0101              0010 0010
  ^ 0110 0111            ^ 0110 0111
  -----------            -----------
    0010 0010              0100 0101
```

XOR Encryption, Continued

- But, we have a problem
  - Some characters are control characters
  - These characters don't print
- So, let's play it safe
  - If the encrypted character would be a control character
  - … just print the original, unencrypted character
  - Note: the same thing will happen when decrypting, so we're okay
- C function **`iscntrl()`**
  - Returns true if the character is a control character

XOR Encryption, C Code

```
#define KEY '&'
int main() {
   int orig_char, new_char;

   while ((orig_char = getchar()) != EOF) {
      new_char = orig_char ^ KEY;
      if (iscntrl(new_char))
         putchar(orig_char);
      else
         putchar(new_char);
   }
   return 0;
}
```

Next Week

- Wednesday lecture time
  - Midterm exam
  - Open book and open notes
  - Practice exams online

Stupid Programmer Tricks

- Where do I use bitwise & most?
  - Bit vectors
- What's a bit vector?
  - Lots of booleans packed into an int/long
  - Often used to indicate some condition(s)
  - Less storage space than lots of fields
  - More explicit storage than compiled-defined bit fields
- Your compiler can do this?

```
typedef struct blah {
  int b_onoff:1;
  int b_temperature:7;
  char b_someChar;
}
```

Example From Real Code

- #define DONTCACHE_REQNOSTORE 0x000001
- #define DONTCACHE_AUTHORIZED 0x000002
- #define DONTCACHE_MISSINGVARIANTHDR 0x000004
- #define DONTCACHE_USERORPASS 0x000008
- #define DONTCACHE_BYPASSFILTER 0x000010
- #define DONTCACHE_NONCACHEMETHOD 0x000020
- #define DONTCACHE_CTLPRIVATE 0x000040
- #define DONTCACHE_CTLNOSTORE 0x000080
- #define DONTCACHE_ISQUERY 0x000100
- #define DONTCACHE_EARLYEXPIRE 0x000200
- #define DONTCACHE_NOLASTMOD 0x000400
- #define DONTCACHE_NONEGCACHING 0x000800
- #define DONTCACHE_INSTANTEXPIRE 0x001000
- #define DONTCACHE_FILETOOBIG 0x002000
- #define DONTCACHE_FILEGREWTOOBIG 0x004000
- #define DONTCACHE_ICPPROXYONLY 0x008000
- #define DONTCACHE_LARGEFILEBLAST 0x010000
- #define DONTCACHE_PERSISTLOGLOADING 0x020000
- #define DONTCACHE_NEWERCOPYEXISTS 0x040000
- #define DONTCACHE_BADVARYFIELDS 0x080000
- #define DONTCACHE_SETCOOKIE 0x100000
- #define DONTCACHE_HTTPSTATUSCODE 0x200000
- #define DONTCACHE_OBJECTINCOMPLETE 0x400000

Conclusions

- 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
- Binary operations in C
  - AND, OR, NOT, and XOR
  - Shift left and shift right
  - Useful for efficient and concise code, though sometimes cryptic