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

Decimal to binary conversion:
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
- 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: De-Comment Assignment Due
  - Blue: Reading Period
- 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
- Examples of unsigned integers
  - 00000001 → 1
  - 00001111 → 15
  - 00100001 → 33
  - 11111111 → 255 ($2^8 – 1$)
- All arithmetic is modulo 2^n
- Signed integers, negative numbers: soon

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

Base 10

	1	9	8
+	2	6	4
Sum	4	6	2
Carry	0	1	1

Base 2

	0	1	1
+	0	0	1
Sum	1	0	0
Carry	0	1	1

Binary Sums and Carries

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

XOR
("exclusive OR")

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
  - Turn subtraction into addition: a – b into a + x
  - Let x be easily computable from b
  - Use properties of modulo arithmetic and number complements

Subtraction made easy

- Turn subtraction into addition
  - Suppose you want to compute a – b, in eight-bit representation
  - This equals (a – b) + 256 (modulo arithmetic)
  - This equals a + (256 – b) [generally, $a + (2^n - b)$]
  - This equals a + (256 -1 - b) + 1 [$a + (2^n - 1 - b) + 1$]

  - $2^n - 1 - b$ is easy to compute
    - $2^n - 1$ is all 1s: 1111 1111 for 2^8 (256 – 1)
    - So $(2^n - 1) - b$ is just b with all the bits flipped
    - This is called the one's complement of b

  - Therefore $(2^n - 1 - b) + 1$ is also easy to compute (just add 1)
    - This is called the two's complement of b

  - The rest is just an addition with a

One's and Two's Complement

- Example: 172 – 69 (in eight bit arithmetic)
  - $172 + (2^8 - 1 - 69) + 1$
- Compute the one's complement of b (here b = 69)
  - That's simply 255 – 69

```
   1111 1111
-  0100 0101   ←  b
 ___________
   1011 1010   ←  one's complement of b
```

  - Flip every bit of 69 to get the one's complement $(2^8 - 1 - 69)$
- Compute the two's complement of b
  - Add 1 to the one's complement
  - E.g., (255 – 69) + 1 → 1011 1011

Putting it All Together

- Computing "a – b"
  - a + $(2^n - 1 - b) + 1$
  - Same as "a + twosComplement(b)"
  - Same as "a + onesComplement(b) + 1"
- 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

```
    1010 1100
+   1011 1011
-------------
   10110 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
  - -b can be represented as the One's complement of b
    - Flip every bit
    - E.g., 1101 0011 → -44
  - -b can be represented as the Two's complement of b
    - 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 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 (&)

&	0	1
0	0	0
1	0	1

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

- 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

- AND (&)

&	0	1
0	0	0
1	0	1

- OR (|)

|	0	1
0	0	1
1	1	1

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	1
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 negative integers?
    - Can vary from one machine to another

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

Example: Counting the 1’s

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

0	0	1	1	0	1	0	1

  - 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
    - All bits but the last in 1 are zeros, so this n & 1 is either 0 or 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