Number System
Number Representation

Topics to be Discussed

- How are numeric data items actually stored in computer memory?
- How much space (memory locations) is allocated for each type of data?
  - int, float, char, etc.
- How are characters and strings stored in memory?

Number System :: The Basics

- We are accustomed to using the so-called *decimal number system*.
  - Ten digits :: 0,1,2,3,4,5,6,7,8,9
  - Every digit position has a weight which is a power of 10.
  - Base or radix is 10.
- Example:

  234 = $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

  250.67 = $2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$

Binary Number System

- Two digits:
  - 0 and 1.
  - Every digit position has a weight which is a power of 2.
  - *Base* or *radix* is 2.
- Example:

  110 = $1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

  101.01 = $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Counting with Binary Numbers

0
1
10
11
100
101
110
111
1000
.

Multiplication and Division with base

- Multiplication with 10 (decimal system)
  435 x 10 = 4350

Left Shift and add zero at right end

- Multiplication with 10 (=2) (binary system)
  1101 x 10 = 11010

Right shift and drop right most digit or shift after decimal point

- Division by 10 (decimal system)
  435 / 10 = 43.5

- Division by 10 (=2) (binary system)
  1101 / 10 = 110.1

Adding two bits

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10

carry

Carries

	1	1	1	0	
		1	0	1	1
+		1	1	1	0
	1	1	0	0	1

Binary addition: Another example

The initial carry in is implicitly 0

	1	1	0	0		(Carries)
		1	1	0	1	
+		1	1	0	0	
	1	1	0	0	1	(Sum)

most significant bit (MSB)

least significant bit (LSB)

Binary-to-Decimal Conversion

- Each digit position of a binary number has a weight.
  - Some power of 2.
- A binary number:

  $B = b_{n-1}\ b_{n-2} \ldots\ldots b_1\ b_0\ .\ b_{-1}\ b_{-2} \ldots\ldots b_{-m}$

  Corresponding value in decimal:

  $$D = \sum_{i=-m}^{n-1} b_i\, 2^i$$

Examples

1. 101011 ➔ $1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$

   = 43

   $(101011)_2 = (43)_{10}$

2. .0101 ➔ $0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4}$

   = .3125

   $(.0101)_2 = (.3125)_{10}$

3. 101.11 ➔ $1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$

   5.75

   $(101.11)_2 = (5.75)_{10}$

Decimal-to-Binary Conversion

- Consider the integer and fractional parts separately.
- For the integer part,
  – Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
  – Arrange the remainders *in reverse order*.
- For the fractional part,
  – Repeatedly multiply the given fraction by 2.
    - Accumulate the integer part (0 or 1).
    - If the integer part is 1, chop it off.
  – Arrange the integer parts *in the order* they are obtained.

Example 1 :: 239

2	239	
2	119	--- 1
2	59	--- 1
2	29	--- 1
2	14	--- 1
2	7	--- 0
2	3	--- 1
2	1	--- 1
2	0	--- 1

$(239)_{10} = (11101111)_2$

Example 2 :: 64

2	64	
2	32	--- 0
2	16	--- 0
2	8	--- 0
2	4	--- 0
2	2	--- 0
2	1	--- 0
2	0	--- 1

$(64)_{10} = (1000000)_2$

Example 3 :: .634

.634 x 2 = 1.268
.268 x 2 = 0.536
.536 x 2 = 1.072
.072 x 2 = 0.144
.144 x 2 = 0.288
:
:

$(.634)_{10} = (.10100......)_2$

Example 4 :: 37.0625

$(37)_{10} = (100101)_2$

$(.0625)_{10} = (.0001)_2$

$(37.0625)_{10} = (100101\ .\ 0001)_2$

Hexadecimal Number System

- A compact way of representing binary numbers.
- 16 different symbols (radix = 16).

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

Binary-to-Hexadecimal Conversion

- For the integer part,
  - Scan the binary number from *right to left*.
  - Translate each group of four bits into the corresponding hexadecimal digit.
    - Add *leading* zeros if necessary.
- For the fractional part,
  - Scan the binary number from *left to right*.
  - Translate each group of four bits into the corresponding hexadecimal digit.
    - Add *trailing* zeros if necessary.

Example

1. $(\underline{1011}\ \underline{0100}\ \underline{0011})_2 = (B43)_{16}$
2. $(\underline{10}\ \underline{1010}\ \underline{0001})_2 = (2A1)_{16}$
3. $(.\underline{1000}\ \underline{010})_2 = (.84)_{16}$
4. $(\underline{101}\ .\ \underline{0101}\ \underline{111})_2 = (5.5E)_{16}$

Hexadecimal-to-Binary Conversion

- Translate every hexadecimal digit into its 4-bit binary equivalent.
- Examples:

$(3A5)_{16} = (\underline{0011}\ \underline{1010}\ \underline{0101})_2$

$(12.3D)_{16} = (\underline{0001}\ \underline{0010}\ .\ \underline{0011}\ \underline{1101})_2$

$(1.8)_{16} = (\underline{0001}\ .\ \underline{1000})_2$

Unsigned Binary Numbers

- An n-bit binary number

  $B = b_{n-1}b_{n-2} \ldots b_2b_1b_0$

  - 2^n distinct combinations are possible, 0 to 2^n-1.
- For example, for n = 3, there are 8 distinct combinations.
  - 000, 001, 010, 011, 100, 101, 110, 111
- Range of numbers that can be represented

  n=8 ➔ 0 to 2^8-1 (255)

  n=16 ➔ 0 to $2^{16}-1$ (65535)

  n=32 ➔ 0 to $2^{32}-1$ (4294967295)

Signed Integer Representation

- Many of the numerical data items that are used in a program are signed (positive or negative).
  - Question:: How to represent sign?
- Three possible approaches:
  - Sign-magnitude representation
  - One's complement representation
  - Two's complement representation

Sign-magnitude Representation

- For an n-bit number representation
  - The most significant bit (MSB) indicates sign
    - 0 → positive
    - 1 → negative
  - The remaining n-1 bits represent magnitude.

Representation and ZERO

- Range of numbers that can be represented:

  Maximum :: $+ (2^{n-1} – 1)$

  Minimum :: $– (2^{n-1} – 1)$

- A problem:

  Two different representations of zero.

  +0 → 0 000….0

  -0 → 1 000….0

One's Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form.
  - Negative numbers are represented in 1's complement form.

- How to compute the 1's complement of a number?
  - Complement every bit of the number (1→0 and 0→1).
  - MSB will indicate the sign of the number.

    0 → positive

    1 → negative

Example :: n=4

0000 → +0	1000 → -7
0001 → +1	1001 → -6
0010 → +2	1010 → -5
0011 → +3	1011 → -4
0100 → +4	1100 → -3
0101 → +5	1101 → -2
0110 → +6	1110 → -1
0111 → +7	1111 → -0

To find the representation of -4, first note that

+4 = 0100

-4 = 1's complement of 0100 = 1011

One's Complement Representation

- Range of numbers that can be represented:
  - Maximum :: $+ (2^{n-1} – 1)$
  - Minimum :: $– (2^{n-1} – 1)$
- A problem:
  - Two different representations of zero.
    - +0 → 0 000….0
    - -0 → 1 111….1
- Advantage of 1's complement representation
  - Subtraction can be done using addition.
  - Leads to substantial saving in circuitry.

Two's Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form.
  - Negative numbers are represented in 2's complement form.

- How to compute the 2's complement of a number?
  - Complement every bit of the number (1→0 and 0→1), and then ***add one*** to the resulting number.
  - MSB will indicate the sign of the number.

    0 → positive

    1 → negative

Example :: n=4

0000 → +0	1000 → -8
0001 → +1	1001 → -7
0010 → +2	1010 → -6
0011 → +3	1011 → -5
0100 → +4	1100 → -4
0101 → +5	1101 → -3
0110 → +6	1110 → -2
0111 → +7	1111 → -1

To find the representation of, say, -4, first note that

+4 = 0100

-4 = 2's complement of 0100 = 1011+1 = 1100

Storage and number system in Programming

- In C
  - short int
    - 16 bits → $+(2^{15}-1)$ to -2^{15}
  - int
    - 32 bits → $+(2^{31}-1)$ to -2^{31}
  - long int
    - 64 bits → $+(2^{63}-1)$ to -2^{63}

Storage and number system in Programming

- Range of numbers that can be represented:

  Maximum :: $+(2^{n-1}-1)$

  Minimum :: -2^{n-1}

- Advantage:
  - *Unique representation of zero*.
  - Subtraction can be done using addition.
  - Leads to substantial saving in circuitry.

- Almost all computers today use the 2's complement representation for storing negative numbers.

Subtraction Using Addition :: 1's Complement

- How to compute A – B ?
  - Compute the 1's complement of B (say, B_1).
  - Compute $R = A + B_1$
  - If the carry obtained after addition is '1'
    - Add the carry back to R (called *end-around carry*).
    - That is, R = R + 1.
    - The result is a positive number.

  Else
    - The result is negative, and is in 1's complement form.

Example 1 :: 6 – 2

A = 6 (0110)
B = 2 (0010)
6 – 2 = A - B

1's complement of 2 = 1101

```
       6  ::   0110      A
      -2  ::   1101      B1
              1 0011     R
End-around carry  1
               0100  →  +4
```

Assume 4-bit representations.

Since there is a carry, it is added back to the result.

The result is positive.

Example 2 :: 3 – 5

1's complement of 5 = 1010

3	::	0011	A
-5	::	1010	B_1
		1101	R
		⇩	
		-2	

Assume 4-bit representations.

Since there is no carry, the result is negative.

1101 is the 1's complement of 0010, that is, it represents –2.

Subtraction Using Addition :: 2's Complement

- How to compute A – B ?
  - Compute the 2's complement of B (say, B_2).
  - Compute $R = A + B_2$
  - Ignore carry if it is there.
  - The result is in 2's complement form.

Example 1 :: 6 – 2

2's complement of 2 = 1101 + 1 = 1110

6	::	0110	A
-2	::	1110	B_2
		1 0100	R

Ignore carry

⇓ +4

Example 2 :: 3 – 5

2's complement of 5 = 1010 + 1 = 1011

3	::	0011	A
-5	::	1011	B_2
		1110	R

⇓ -2

Example 3 :: -3 – 5

2's complement of 3 = 1100 + 1 = 1101

2's complement of 5 = 1010 + 1 = 1011

```
-3  ::   1101
-5  ::   1011
        ------
        1 1000
```

Ignore carry

-8

Floating-point Numbers

- The representations discussed so far applies only to integers.
  - Cannot represent numbers with fractional parts.
- We can assume a decimal point before a 2's complement number.
  - In that case, pure fractions (without integer parts) can be represented.
- We can also assume the decimal point somewhere in between.
  - This lacks flexibility.
  - Very large and very small numbers cannot be represented.

Representation of Floating-Point Numbers

- A floating-point number F is represented by a doublet <M,E> :

  $F = M \times B^E$
  - B → exponent base (usually 2)
  - M → mantissa
  - E → exponent
  - M is usually represented in 2's complement form, with an implied decimal point before it.

- For example,

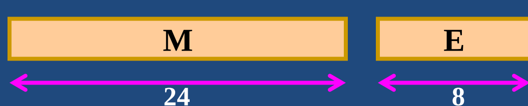
  In decimal,

  0.235×10^6

  In binary,

  0.101011×2^{0110}

Example :: 32-bit representation



- M represents a 2's complement fraction

  $1 > M > -1$
- E represents the exponent (in 2's complement form)

  $127 > E > -128$

- Points to note:
  - The number of *significant digits* depends on the number of bits in M.
    - 6 significant digits for 24-bit mantissa.
  - The *range* of the number depends on the number of bits in E.
    - 10^{38} to 10^{-38} for 8-bit exponent.

Floating point number: IEEE Standard 754

- Storage Layout

	Sign	Exponent	Fraction / Mantissa
Single Precision	1 [31]	8 [30–23]	23 [22–00]
Double Precision	1 [63]	11 [62–52]	52 [51–00]

Single: SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM
Double: SEEEEEEE EEEEMMMM MMMMMMMM MMMMMMMM
MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM

IEEE Standard 754

1. The sign bit is 0 for positive, 1 for negative.
2. The exponent base is two.
3. The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
4. The first bit of the mantissa is typically assumed to be 1.*f*, where *f* is the field of fraction bits.

- Ranges of Floating-Point Numbers

Since every floating-point number has a corresponding, negated value (by toggling the sign bit), the ranges above are symmetric around zero.

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$	$\pm \approx 10^{-44.85}$ to $\approx 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$	$\pm \approx 10^{-323.3}$ to $\approx 10^{308.3}$

IEEE Standard 754

There are five distinct numerical ranges that single-precision floating-point numbers are **not** able to represent:

1. Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (*negative overflow*)
2. Negative numbers greater than -2^{-149} (*negative underflow*)
3. Zero
4. Positive numbers less than 2^{-149} (*positive underflow*)
5. Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (*positive overflow*)

Special Values

- Zero
  −0 and +0 are distinct values, though they both compare as equal.
- Denormalized
  If the exponent is all 0s, but the fraction is non-zero, then the value is a *denormalized* number, which now has an assumed leading *0* before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where *s* is the sign bit and *f* is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.
- Infinity
  The values +∞ and −∞ are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. *Operations with infinite values are well defined in IEEE floating point.*
- Not A Number
  The value NaN (*Not a Number*) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction.

Representation of Characters

- Many applications have to deal with non-numerical data.
  - Characters and strings.
  - There must be a standard mechanism to represent alphanumeric and other characters in memory.

- Three standards in use:
  - Extended Binary Coded Decimal Interchange Code (EBCDIC)
    - Used in older IBM machines.
  - American Standard Code for Information Interchange (ASCII)
    - Most widely used today.
  - UNICODE
    - Used to represent all international characters.
    - Used by Java.

ASCII Code

- Each individual character is numerically encoded into a unique 7-bit binary code.
  - A total of 2^7 or 128 different characters.
  - A character is normally encoded in a byte (8 bits), with the MSB not been used.

- The binary encoding of the characters follow a regular ordering.
  - Digits are ordered consecutively in their proper numerical sequence (0 to 9).
  - Letters (uppercase and lowercase) are arranged consecutively in their proper alphabetic order.

Some Common ASCII Codes

‘A’ :: 41 (H) 65 (D)	‘0’ :: 30 (H) 48 (D)
‘B’ :: 42 (H) 66 (D)	‘1’ :: 31 (H) 49 (D)
………..	………..
‘Z’ :: 5A (H) 90 (D)	‘9’ :: 39 (H) 57 (D)
‘a’ :: 61 (H) 97 (D)	‘(‘ :: 28 (H) 40 (D)
‘b’ :: 62 (H) 98 (D)	‘+’ :: 2B (H) 43 (D)
………..	‘?’ :: 3F (H) 63 (D)
‘z’ :: 7A (H) 122 (D)	‘\n’ :: 0A (H) 10 (D)
	‘\0’ :: 00 (H) 00 (D)

Character Strings

- Two ways of representing a sequence of characters in memory.
  - The first location contains the number of characters in the string, followed by the actual characters.

5	H	e	l	l	o

  - The characters follow one another, and is terminated by a special delimiter.

H	e	l	l	o	⊥

String Representation in C

- In C, the second approach is used.
  - The '\0' character is used as the string delimiter.
- Example:

  "Hello" ➔

H	**e**	**l**	**l**	**o**	**'\0'**

- A null string "" occupies one byte in memory.
  - Only the '\0' character.

Problem 7

Given 2 positive numbers *n* and *r*, *n>=r* , write a C function to compute the number of combinations($^{n}C_{r}$) and the number of permutations($^{n}P_{r}$).

Permutations formula is P(n,r)=n!/(n-r)!

Combinations formula is C(n,r)=n!/(r!(n-r)!)

Problem 8

Scope of variable:
What is the output of the following code snippet?

```
#include <stdio.h>

int main(){
        int i = 10;
        for(int i = 5; i < 15; i++)
                printf("i is %d\n", i);
         return 0;
}
```

Problem 9

Scope of variable: What is the output of the following code snippet?

```
#include <stdio.h>
int a = 20;
int sum(int a, int b) {
        printf ("value of a in sum() = %d\n",  a);
        printf ("value of b in sum() = %d\n",  b);
        return a + b;
}
int main ()
{
        int a = 10;  int b = 20;  int c = 0;
        printf ("value of a in main() = %d\n",  a);
        c = sum( a, b);
        printf ("value of c in main() = %d\n",  c);
        return 0;
}
```

Problem 10

Write a C program which display the entered number in words.

Example:

Input:
Enter a number: 7

Output:
Seven

Problem 11

Write a C program to delete duplicate elements in an array without using another auxiliary array.

Example:

Input:
5 8 5 5 6 9 8 2 1 1 3 3

Output:
5 8 6 9 2 1 3

Problem 12

Write a C program to print PASCAL's triangle.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
```

Problem 13

Given 2 numbers ***a*** and ***b***, write a C program to compute the Greatest Common Divisor(GCD) of the 2 numbers.

The GCD of 2 numbers is the largest positive integer that divides the numbers without a remainder.

Example: GCD(2,8)=2; GCD(3,7)=1

Problem 14

Given 2 arrays of integers ***A*** and ***B*** of size ***n*** each, write a C program to calculate the dot product of the 2 arrays.

If ***A***=[a_0, a_1, a_2,….,a_{n-1}] and
B= [b_0, b_1, b_2,….,b_{n-1}],
the dot product of ***A*** and ***B*** is given by
$A.B=[a_0*b_0 + a_1*b_1 + a_2*b_2 + \ldots\ldots + a_{n-1}*b_{n-1}]$

Problem 15

Given a non negative integer *n*, write a C function to output the decimal integer(base 10) in its binary representation (base 2).

Example: Binary representation of

3	is	11
8	is	1000
15	is	1111

Problem 16

Given two array of sorted numbers *A* and *B*, both are of arbitrary sizes, write a C function named ***merge_arrays*** that merges both the arrays in sorted order and returns the sorted array C.