# Representing and Manipulating Information

## 2.1 Information Storage

Rather than accessing individual bits in memory, most computers use blocks of 8 bits, or bytes, as the smallest addressable unit of memory.

Every byte of memory is identified by a unique number, known as its address.

The C compiler also associates type information with each pointer, so that it can generate different machine-level code to access the value stored at the location designated by the pointer depending on the type of that value.

### 2.1.1 Hexadecimal Notation

Binary notation is too verbose, while with decimal notation it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base-16, or hexadecimal numbers. Hexadecimal (or simply "hex")

### 2.1.2 Data Size

Every computer has a word size, indicating the nominal size of pointer data.

Computers and compilers support multiple data formats using different ways to encode data, such as integers and floating point, as well as different lengths.

#### Typical sizes(in bytes) of basic C data types

| C declaration | | Bytes | |
|---|---|---|---|
| Signed | Unsigned | 32-bit | 64-bit |
| [signed] char | unsigned char | 1 | 1 |
| short | unsigned short | 2 | 2 |
| int | unsigned | 4 | 4 |
| long | unsigned long | 4 | 8 |
| int32_t | uint32_t | 4 | 4 |
| int64_t | uint64_t | 8 | 8 |
| char* | | 4 | 8 |
| float | | 4 | 4 |
| double | | 8 | 8 |

### 2.1.3 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what the address of the object will be, and how we will order the bytes in memory.

#### Two ways of convention

- **Little endian**: where the least significant byte comes first–is referred to as little endian.
- **Big endian**: where the most significant byte comes first–is referred to as big endian.

### 2.1.4 Representing Strings

A string in C is encoded by an array of characters terminated by the null (having value 0) character. Each character is represented by some standard encoding, with the most common being the ASCII character code.

### 2.1.5 Representing Code

Different machine tgyes use different and incompatible instructions and encodings. Even identical proces- sors running different operating systems have differences in their osding conven- tions and hence are not binary compatible. Binary code is seldom portable across different combinations of machine and operating system.

A fundamental concept of computer systems is that a program, from the perspective of the machine, is simply a sequence of bytes.

### 2.1.6 Introduction to Boolean Algebra

Boole observed that by encoding logic values true and false as binary values 1 and 0, he could formulate an algebra that captures the basic principles of logical reasoning.

#### Operations of Boolean algebra

| & | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

**Exclusive-Or('^') can be considered as add up without carry.**

We can extend the four Boolean operations to also operate on bit vectors, strings of zeros and ones of some fixed length w.

#### One useful application

One useful application of bit vectors is to represent finite sets.

We can encode any subset A ⊂ {0, 1, . . . , w − 1} with a bit vector (aw−1, . . . , a1 , a0), where ai= 1 if and only if i ∈ A.

### 2.1.7 Bit-Level Operations in C

One useful feature of C is that it supports bitwise Boolean operations. In fact, the symbols we have used for the Boolean operations are exactly those used by C:

#### Masking operations

One common use of bit-level operations is to implement masking operations

### 2.1.8 Logical Operations in C

C also provides a set of logical operators ||, &&, and !, which correspond to the or, and, and not operations of logic.

#### Two distinctions between logical operators and bit-level operations

- **Not '0' is TRUE**: The logical operations treat any nonzero argument as representing true and argument 0 as representing false. They return either 1 or 0, indicating a result of either true or false, respectively.
- **Logical short circuit**: the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument.

#### Avoid nullpointer

p && *p++ will never cause the dereferencing of a null pointer.

### 2.1.9 Shift Operation in C

C also provides a set of shift operations for shifting bit patterns to the left and to the right.

#### Left Shift

That is, x is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k zeros. The shift amount should be a value between 0 and w − 1.

#### Undefined Behavior

Say, X(8 bits) << 8.
**On most machine you will get X. Because the way machine deal with the situation when shift more bits or equal bits than the original bits. It will shift the most of the length of original bits.In this case, shift 8 mod 8 (0).**

#### Right Shift

Generally, machines support two forms of right shift:

- **Logical**: Logical. A logical right shift fills the left end with k zeros
- **Arithmetic**: Arithmetic. An arithmetic right shift fills the left end with k repetitions of the most significant bit

## 2.2 Integer Representations

In this section, we describe two different ways bits can be used to encode integers–one that can only represent nonnegative numbers, and one that can represent negative, zero, and positive numbers.

### 2.2.1 Integral Data Types

Integral Data Types

C supports a variety of integral data types–ones that represent finite ranges of integers.

Each type can specify a size with keyword char, short, long, as well as an indication of whether the represented numbers are all nonnegative (declared as unsigned), or possibly negative (the default.)

### 2.2.1 Unsigned Encodings

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

#### The rang of values

The unsigned binary representation has the important property that every number between 0 and 2w − 1has a unique encoding as a w-bit value.

### 2.2.3 Two's-Complement Encodings

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

#### The rang of values

The least representable value is given by bit vector [10. . . 0] (set the bit with negative weight but clear all others), having integer value TMinw = −2w−1. The greatest value is given by bit vector [01. . . 1] (clear the bit with negative weight but set all others), having integer value TMaxw.=2w−2 =(02)= 2w−1− 1.

### 2.2.4 Conversions between Signed and Unsigned

C allows casting between different numeric data types.

The effect of casting is to keep the bit values identical but change how these bits are interpreted.

### 2.2.5 Signed versis Unsigned in C

Signed versus Unsigned in C

Although the C standard does not spec- ify a particular representation of signed numbers, almost all machines use two's complement.

#### Conversions can happen in three ways

- **Explict casting**: Conversions can happen due to explicit casting,
- **Assignment**: they can happen implicitly when an expression of one type is as- signed to a variable of another
- **Printf**: When printing numeric values with printf, the directives %d, %u, and %x are used to print a number as a signed decimal, an unsigned decimal, and in hexadecimal format
- **Two different type oprand**: When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations assuming the numbers are nonnegative.

**sizeof()**

**The type of the return value of sizeof() is unsigned, so be very careful when you combine the sizeof() with some logical or numerical operations.**

### 2.2.6 Expanding the Bit Representation of a Number

One common operation is to convert between integers having different word sizes while retaining the same numeric value. This may not be possible when the destination data type is too small to represent the desired value. Converting from a smaller to a larger data type, however, should always be possible.

#### Zero extension

To convert an unsigned number to a larger data type, we can simply add leading zeros to the representation;

#### Sign extension

the rule is to perform a sign extension, adding copies of the most significant bit to the representation, expressed by the following principle.

#### Provement

The key property we exploit is that 2w−2w−1= 2w−1. Thus, the combined effect of adding a bit of weight −2wand of converting the bit having weight −2w−1to be one with weight 2w−1is to preserve the original numeric value.

#### Effect of relative order of conversion

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program.

### 2.2.7 Truncating Numbers

Suppose that, rather than extending a value with extra bits, we reduce the number of bits representing a number.

#### Truncation of an unsigned number (mod operation)

The result is the origin number mod 2^k.
The intuition behind this principle is simply that all of the bits that were truncated have weights of the form 2i, where i ≥ k, and therefore each of these weights reduces to zero under the modulus operation. This is formalized by the following derivation

#### Truncation of a Two's-complement number (similar to mod operation)

In this formulation, x mod 2kwill be a number between 0 and 2k−1 .Applying function U2Tk to it will have the effect of converting the most significant bit xk−1 from having weight 2k−1to having weight −2k−1.

## 2.3 Integer Arithmetic

Integer Arithmetic

### 2.3.1 Unsigned Addition

If we were to maintain the sum as a (w + 1)-bit number and add it to another value, we may require w + 2 bits.

#### Actual arithmetic

This can be characterized as a form of modular arithmetic, computing the sum modulo 2wby simply discarding any bits with weight greater than 2w−1in the bit-level representation of x + y.
Result = (x + y) mod 2^w;

### 2.3.2 Two's-Complement Addition

With two's-complement addition, we must decide what to do when the result is either too large (positive) or too small (negative) to represent.

#### Actual arithmetic

The w-bit two's-complement sum of two numbers has the exact same bit-level representation as the unsigned sum. In fact, most computers use the same machine instruction to perform either unsigned or signed addition.

### 2.3.3 Two's-Complement Negation

$$-_w^t x = \begin{cases} TMin_w & x = TMin_w \\ -x & x > TMin_w \end{cases}$$

#### Trick to calculate the Two's complement Negation

One technique for performing two's-complement negation at the bit level is to complement the bits and then increment the result. In C, we can state that for any integer value x, computing the expressions ~x and ~x + 1 will give identical results.

### 2.3.4 Unsigned Multiplication

Unsigned Multiplication

### 2.3.5 Two's-Complement Multiplication

Two's-Complement Multiplication

#### Summary

**Basically, Computer just take both unsigned integer or two's-complement all as unsigned while calculating, and truncate the result to keep the lower w-bits. Then interpret the result with whether it is unsigned or two's complement.**

### 2.3.6 Multiplying by Constants

One important optimization used by compilers is to attempt to replace multiplications by constant factors with combinations of shift and addition operations.

#### Multiplication by a power of 2

Shifting a value left is equivalent to performing unsigned multiplication by a power of 2:

#### Break constant into sum or subtraction of power of 2

The compiler can express the binary representation of K as an alternating sequence of zeros and ones:

Consider a run of ones from bit position n down to bit position m (n ≥ m). (For the case of 14, we have n = 3 and m = 1.) We can compute the effect of these bits on the product using either of two different forms: Form A: (x<<n) + (x<<(n − 1)) + ... + (x<<m) Form B: (x<<(n + 1)) - (x<<m) By adding together the results for each run, we are able to compute x * K with- out any multiplications.

#### Multiplying by constant

Given that integer multiplication is more costly than shifting and adding, many C compilers try to remove many cases where an integer is being multiplied by a constant with combinations of shifting, adding, and subtracting.

### 2.3.7 Dividing by Powers of 2

Integer division on most machines is even slower than integer multiplication–requiring 30 or more clock cycles. Dividing by a power of 2 can also be performed using shift operations, but we use a right shift rather than a left shift, arithmetic right shift of course.

#### Two's-complement division by a power of 2, rounding down

Let C variables x and k have two's-complement value x and unsigned value k, respectively, such that 0 ≤ k <w. The C expression x >> k, when the shift is performed arithmetically, yields the value floor(x/2^k).

#### Two's-complement division by a power of 2, rounding up

Let C variables x and k have two's-complement value x and unsigned value k, respectively, such that 0 ≤ k <w. The C expression (x + (1 << k) - 1) >> k, when the shift is performed arithmetically, yields the value ceil(x/2^k).

#### Counting Down with Unsigned

Proper way to use unsigned as loop index:
unsigned i;
for(i = cnt - 2; i < cnt; i--){
a[i] += a[i+1];
}

#### Why Should I Use Unsigned ?

**After all it's easy to make mistake and can be very subtle.**

#### Do Use When Performing Modular Arithmetic

#### Do Use When Using Bits to Represent Sets

## 2.4 Floating Point

In this section, we will see how numbers are represented in the IEEE floating- point format.

### 2.4.1 Fractional Binary Numbers

By analogy, consider a notation of the form

$$a_m a_{m-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots a_{-n+1} a_{-n}$$

This notation represents a number b defined as

$$b = \sum_{i=-n}^{m} 2^i \times b_i$$

#### Limitation #1

Fractional binary notation can only represent numbers that can be written x × 2y. Other values can only be approximated.

#### Limitation #2

**Just one setting of binary point within the w bits.Limited range of numbers(very small values ? Very large)**

### 2.4.2 IEEE Floating-Point Representation

The IEEE floating-point standard represents a number in a form V = (−1)^s× M × 2^E

The sign s determines whether the number is negative (s = 1) or positive (s = 0), where the interpretation of the sign bit for numeric value 0 is handled as a special case.
The significand M is a fractional binary number that ranges either between 1 and 2 or between 0 and 1.
The exponent E weights the value by a (possibly negative) power of 2.

#### Standard floating point formats

The single sign bit s directly encodes the sign s.
The k-bit exponent field exp = ek−1 . . . e1e0encodes the exponent E.
The n-bit fraction field frac = fn−1 . . . f1f0encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.

#### Case 1 : Normalized Values

This is the most common case. It occurs when the bit pattern of exp is neither all zeros (numeric value 0) nor all ones (numeric value 255 for single precision, 2047 for double).

- **Exponent coded as a biased value**: That is, the exponent value is E = e − Bias, where e is the unsigned number having bit representation ek−1 . . . e1e0and Bias is a bias value equal to 2k−1 − 1 (127 for single precision and 1023 for double). This yields exponent ranges from −126 to +127 for single precision and −1022 to +1023 for double precision.
- **Significant coded with implied leading 1**: we can view M to be the number with binary representation 1.fn−1fn−2 . . .f0.
  **Minimum when frac = 00. . .00(M = 1.0)**
  **Maximum when frac = 11 . . .11(M = 2.0 - ε)**
  **Get extra leading bit for "free".**

#### Case 2 : Denormalized Values

When the exponent field is all zeros, the represented number is in denormalized form.

- **Exponent**: The exponent value is E = 1 − Bias,NOT 0 - Bias.
  Smooth transition between the denormalized number and normalized number
- **Significant value**: the significand value is M = f , that is, the value of the fraction field WITHOUT an implied leading 1.
- **Two purposes of denormalized values**: First, they provide a way to represent numeric value 0,
  A second function of denormalized numbers is to represent numbers that are very close to 0.0.

#### Case 3 : Special Values

A final category of values occurs when the exponent field is all ones.

- **Infinity**: When the fraction field is all zeros, the resulting value represent infinity, either +∞ when s = 0 or −∞ when s = 1.
- **NaN**: When the fraction field is nonzero, the resulting value is called a NaN, short for "not a number."

### 2.4.3 Visualization of the representable values for 6-bit floating-point format

#### Special Properties of the IEEE Encoding

#### Smooth transition between the denormalized number and normalized number

Observe the smooth transition between the largest denormalized number 7 512 and the smallest normalized number 8 512. This smoothness is due to our definition of E for denormalized values. By making it 1 − Bias rather than −Bias, we com- pensate for the fact that the significand of a denormalized number does not have an implied leading 1.

Exponent

- **Floating-point Zero Same as Integer Zero**: The value +0.0 always has a bit representation of all zeros.
- **Can (Almost) Use Unsigned Integer Comparison**: The IEEE format was designed so that floating-point numbers could be sorted using an integer sorting routine. A minor difficulty occurs when dealing with negative numbers, since they have a leading 1 and occur in descending order, but this can be overcome without requiring floating-point operations to perform comparisons

**Principles**

1) Must first consider sign bits
2) Must consider -0 = +0
3) NaNs problematic:
   Will be greater than any other values:
   What should comparison yield?
4)Otherwise OK:
   Denormalized vs. normalized
   Normalized vs. Infinity

### 2.4.4 Rounding

The IEEE floating-point format defines four different rounding modes.

#### Round-to-even(default)

Round-to-even (also called round-to-nearest) is the default mode. It attempts to find a closest match.

Round-to-even mode adopts the convention that it rounds the number either upward or downward such that the least significant digit of the result is even.

- **Statistical reason**: Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.
- **Rounding Binary Numbers**: In general, the rounding mode is only significant when we have a bit pattern of the form XX . . .XYY. . .YYZ00. . . , where X and Y denote arbitrary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results.

#### The Other Three

The other three modes produce guaranteed bounds on the actual value. These can be useful in some numerical applications.

***Round-toward-zero*** mode rounds positive numbers downward and negative numbers upward,

***Round-down mode*** rounds both positive and negative numbers downward,

***Round-up mode*** rounds both positive and negative numbers upward,

### 2.4.5 Floating-Point Operations

This lack of associativity(结合律) and distributivity(分配律) is of serious concern to scientific programmers and to compiler writers.

### 2.4.6 Floating-point in C

All versions of C provide two different floating point data types: float and double.

#### Casting values between int, float and double

- **From int to float**: The number cannot overflow, but it may be rounded.
- **From int or float to double**: The exact numeric value can be preserved because double has both greater range, as well as greater precision.
- **From double to float**: The value can overflow to +∞ or −∞, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
- **From float or double to int**: The value will be rounded toward zero.