Representing Information within a Computer System

We will be concerned with representing three basic types of information in computer system

- Integer numbers (signed and unsigned)
- Floating point numbers (scientific notation)
- The “Latin” character set in which English is written

Before turning to the details of the representation, we need to consider the computer memory system in which the information is stored.

Computer Memory

Computer memory is a comprised of a large collection of bi-state (*off/on)* electrical devices called *bits (binary digits).* A single bit can assume two distinct values which are commonly called {0 , 1}.

Because a single bit encodes so little information, it is necessary to aggregate bits into larger elements.

With two bits we can encode 4 distinct values, {00, 01, 10, 11}. This might lead one to think that the number of distinct values that may be encoded is 2 x the number of bits, but that would be incorrect.

With three bits we can encode not 6 but 8 distinct values {000, 001, 010, 011, 100, 101, 110, 111}. In general, with *n* bits we may repesent 2^n distinct values or “elements of information”.

Encoding schemes used in computer programs use completely arbitrary encodings to represent information in different domains.

For example, common house pets might be encoded:

00	dog
01	cat
10	Vietnamese pot bellied pig
11	bird

Modern memory organization

A computer memory is one dimensional array of individually addressable storage elements (analogous to the boxes of the human computer).

For reasons discussed on the previous page, it was decided in the design of the very earliest computers that each *"box" must contain more than 1 bit* of information.

The basic addressable unit of memory

The "optimal" number of bits in the "box" is arbitrary and various values have been tried through out computer history. Using the term *byte* to mean '*the basic addressable unit of memory*', 5, 6, 7, 8, and 9-bit *bytes* have all been used.

Based upon

- the success of the IBM System 360 (1960's) computer system,
- and the recognition that life was good when the number of bits in byte *is a power of 2*

in virtually all modern computer systems a *byte* is comprised of 8 bits.

A single 8-bit byte can encode 256 distinct values or elements of information:

```
00000000  - 0
00000001  - 1
00000010  - 2
00000011  - 3
    :
    :
11111110  - 255
11111111  - 256
```

Addresses contrasted to contents

It is *extremely* important to understand and distinguish

the *address* of a storage element
the *contents* of a storage element

Addresses -

begin at *0* and increase in unit steps to *N-1* where *N* is the total number of *bytes* in the memory space.

Contents -

Since each basic storage element consists of only 8 bits, there are only $2^8 = 256$ different values that can be contained in a single byte storage element.

Aggregation of basic memory elements

8 bit bytes (or “mailboxes”) are useful for encoding

- the identity of domestic animals commonly found in the home
- characters of the Latin alphabet in which English is written
- ... and many other things, but ....

Numerical problems

More than 8 bits are needed for useful application to numerical problems.   Thus it is common to group adjacent bytes into units commonly called *words.*

- Multiple word lengths are common, and common word lengths include 2 bytes, 4 bytes, and 8 bytes.

- In some architectures (Sun Sparc) it is required that the address of each word be a multiple of word length. That is, the only valid addresses of 4-byte words are 0, 4, 8, 12, 16, ...)

- In other architechtures, words can reside on any byte boundary, but the use of unaligned words often causes a performance problem.

- Common word lengths now include 16, 32, 64 and 128 bits which correspond to 2, 4, 8, and 16 bytes per word.

- Word lengths may also differ between integer and floating point data types.

- Even when bytes are aggregated into words, addresses remain *byte oriented* in modern computer systems.

- The addresses of 4 byte words are thus 0, 4, 8, 12, 16, ...

Mathematical integer arithmetic

Mathematically, the integers consist of a countably *infinite* set of values:

$$-\infty, \ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots, \infty$$

The integers comprise what is called a *ring.*

- The sum of two integers is an integer
- The difference of two integers is an integer
- The product of two integers is an integers
- The division of one integer by another is *not defined.*

Computer integer arithmetic

Computer arithmetic is an approximation of mathematical arithmetic

- The number of distinct integers is 2^{word_length} where word length = 8, 16, 32, 64, bits
- Unlike true integer arithmetic on the computer the number of distinct values is *finite.*
- If word length is 8 bits you can represent only the values -128, ... -1, 0, 1, ... 127
- If you compute 100 + 100 *you get the wrong answer* because the maximum positive number that you can represent with 7 bits of information is 127
- A computer instruction set supports *integer division.*
- If you compute 100 / 9 *you get an approximate answer.* Although computer systems do support integer arithmetic, they do so by discarding any remaineder remainder
- So 100 / 9 = 10 (instead of 10.111111..........), and 5 / 2 = 2 (instead of 2.5).

Binary integer arithmetic

We have seen that integers are stored in a computer *words* using a *fixed number* of *binary digits* or bits to encode each value.

Accordingly, computers perform integer arithmetic in *base 2*.

Binary or base 2 arithmetic is a positional system that works just like base 10 but uses 2 rather than 10 distinct "digits". (The choice of base 10 in "human" arithmetic was arbitrary and based upon the number of fingers (digits) possessed by the average human.)

In base 10 the number 1234 means 1 x 10^3 + 2 x 10^2 + 3 x 10 ^ 1 + 4 x 10 ^0 or

```
1000
 200
  30
+  4
1234
```

Thus a digit at position *n* (*n = 0, 1, 2, 3, ...)* is implicitly understood to be multiplied by *10 ^ n.*

In the binary system, a bit at position *n* is implicitly understood to be multiplied by *2 ^ n.*

Thus, the binary number 10110 means

```
16  = 1 * 2 ^ 4
 0  = 0 * 2 ^ 3
 4 =  1 * 2 ^ 2
 2 =  1 * 2 ^ 1
+0 =  0 * 2 ^ 0
22  base 10 = 10110 base 2
```

and we have just devised an algorithm for converting from base 2 to base 10!

Exercise: Convert 1101 and 11011101 from base 2 to base 10.

Converting from base 10 to base 2.

One can convert from base 10 to base to by a sequence of divisions by 2.

- In each successive division the *remainder* is a bit in the base 2 representation.
- The *quotient* becomes the dividend in the next stage of the operation.
- The bits are produced from least significant (low order) to most significant.
- The procedure ends when the quotient becomes 0.

Example: Convert 67 base 10 to base 2.

```
67 / 2  =  33 r 1  <--  least significant ( 2 ^ 0) bit
33 / 2  =  16 r 1
16 / 2  =   8 r 0
 8 / 2  =   4 r 0
 4 / 2  =   2 r 0
 2 / 2   =  1 r 0
 1 / 2   =- 0 r 1   <-- most significant (2 ^ 6 = 64) bit
```

Thus 67 base 10 = 1 0 0 0 0 1 1 base 2

One can verify the accuracy of the computation by converting the result back to base 10.

Here: 64 + 2 + 1 = 67

Exercise: Convert 96 base 10 to base 2.

Bases other than 2 and 10

Any positive integer can be used as a base. For example, in base 3

- The "digits" are {0, 1, 2}
- In a number, such as 12012, each digit is implicitly multiplied by a positional power of 3:

```
  2 x 3 ^ 0 =   2
  1 x 3 ^ 1 =   3
  0 x 3 ^ 2 =   0
  2 x 3 ^ 3 = 54
+1 x 3 ^ 4 = 81
             140 base 10
```

Conversion from base 10 to base 3 is accomplished by a sequence of divisions by 3

Exercise: Convert 134 base 5 to base 10. Convert 79 base 10 to base 4.

Bases that are powers of 2

Binary numbers are very difficult for humans to write and remember:
Consider the 32 bit number:

10110101101110111101010110111110

Bases which are powers of 2 are useful in simplifying notation.

A single digit of a base *2^n* system represents exactly *n* bits of a base 2 system.

Base	# of bits
4	2
8	3
16	4

Converting from base 2 to base 4: (valid digits (0, 1, 2, 3))

```
Base 2:  10 11 10  01
Base 4:    2   3   2    1
```

Converting from base 2 to base 8 (valid digits (0, 1, 2, 3, 4, 5, 6, 7))

```
Base 2: 101 110 001 010
Base 8:   5     6     1     2
```

Base 16 – Hexadecimal

Since 16 is greater than 10, there are not enough digits in the base 10 system to encode base 16 numbers. Thus we augment the digits with the first 6 letters of the alphabet.

Base 16 digits: {0, 1, 2, 3, 4, 5, 6, 7, 8, 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

```
1011 0101 1011 1011 1101 0101 1011 1110
  B    5    B    B    D    5    B    E
```

A single 8 bit byte can always be encoded in exactly 2 hexadecimal digits:

```
  Base 2      Base 16
10010011  =   93
```

Unsigned and signed integer arithmetic

We will use 8 bit arithmetic as an example, but 16 and 32 bit arithmetic operates in essentially the same way.

A mathematical integer consists of an *unbounded* number of bits ==> overflow can't happen.

A computer integer consists of a *finite* number of bits ==> overflow can happen.

Unsigned 8 bit arithmetic:

Possible values in base 10 are 0, 1, 2, ... 255

Consider what happens when we try to compute 240 + 32

```
   1111 0000     240
   0010 0000      32
   0001 0000      16
```

The 1 bit that carries out of the left end of the operation will be discarded and the answer we compute will be 16 which is (240 + 32) modulo 256 in mathematical terms. Computer systems typically provide low level mechanisms for detecting when these situations occur, but these mechanisms typically *are not* available in high level programming languages!!

Signed 8 bit arithmetic -

Most computer systems now use a *2's complement representation* for negative numbers.

- The 1's complement of a value is obtained by inverting all the bits.
- The 2's complement is then obtained by adding 1 to the 1's complement

00101101	-- base integer
11010010	-- 1's complement
11010011	-- 2's complement

- Since the negative of any number is its two's complement, the sum of a number and its two's complement is always 0.
- The difference *a – b* is computed as *a + twos_complement(b)*
- Unlike “sign bit” systems, the twos complement system has only a single 0

Dec	*Binary*	*Hexadecimal*
-128	10000000	80
-127	10000001	81
-126	10000010	82
:		
-2	11111110	FE
-1	11111111	FF
0	00000000	00
1	00000001	01
2	00000010	02
:		
126	01111110	7E
127	01111111	7F

Overflow remains a problem

```
  96      01100000
+68       01001000
          10101000  --- which is a negative number!
```

Encoding the alphabet

Its useful to encode text in computer systems and files:

- Names of account holders in financial records
- Text in word processors
- Part names in inventory systems, etc.

Although the encoding is arbitrary there are some useful characteristics of a code:

- The letters are encoded sequentially: 'A' + 1 is 'B'
- Inverting a single bit can convert between upper and lower case

The code that we will be using is called ASCII (American Standard Code for the Interchange of Information). Printable characters start with the value hex 20 = 32 which is the code for blank space. Following space are many of the special characters that you find on the keyboard.

```
32 20
33 21 !
34 22 "
35 23 #
36 24 $
37 25 %
38 26 &
39 27 '
40 28 (
41 29 )
42 2a *
43 2b +
44 2c ,
45 2d -
46 2e .
47 2f /
```

The encoding of the digits is in the range hex 30 to hex 39

```
48 30 0
49 31 1
 :
56 38 8
57 39 9
```

More special characters follow in the range hex 3A to 40

```
58 3a :
59 3b ;
60 3c <
61 3d =
62 3e >
63 3f ?
64 40 @
```

The upper case letters are next

```
65 41 A
66 42 B
67 43 C
    :
88 58 X
89 59 Y
90 5a Z
```

The ASCII encoding of the letter A is the byte having the value

$$0100\ 0001 = 2^{6} + 2^{0}$$

This value is written in decimal as 65 and in hexadecimal as 41

Another block of special characters occupy hex 5B to 60

```
91 5b [
92 5c \
93 5d ]
94 5e ^
95 5f _
96 60 `
```

The lower case characters occupy hex 61 to 7A.

Upper case A	0100 0001
Lower case a	0110 0001

Case conversion may be accomplished by inverting the bit in the red position

```
 97 61 a
 98 62 b
 99 63 c
   :

120 78 x
121 79 y
122 7a z
```

There are also a few special characters that follow z.

Control characters:

The ASCII encodings between decimal 0 and 31 are used for to encode what are commonly called *control characters.* Control characters having decimal values in the range 1 through 26 can be entered from the keyboard by holding down the *ctrl* key while typing a letter in the set *a* through *z*.

Some control characters have "escaped code" respresentations in C, but all may be written in octal.

Dec	Keystroke	Name	Escaped code
4	ctrl-D	end of file	'\004'
8	ctrl-H	backspace	'\b'
9	ctrl-I	tab	'\t'
10	ctrl-J	newline	'\n'
12	ctrl-L	page eject	'\f'
13	ctrl-M	carriage return	'\r'

The EOF character has no \letter representation.

It may be expressed as '\004' or for that matter simply 4.

Creating signed and unsigned integers in the C language

To create a C variable write a statement of the following form:

modifier type_name variable_name;

The following type names create integer values:

char	-	*8 bit*
short	-	*16 bit*
int	-	*32 bit (sometimes 16)*
long	-	*32 bit (sometimes 64)*

The default modifier is *signed* . To create an *unsigned* integer the *unsigned* modifier must be explicitly provided.

```
unsigned char value1;
int           xyz;
short         has16bits;
```

Variable names:

- Comprised of upper and lower case letters, digits, and _
- Must start with a letter or _
- Must not be a *reserved word (e.g. int, char, unsigned ....) See appendix A of text*

Any signed or unsigned integer variable can hold

- values used in arithmetic computation
- ASCII encoded representation of characters

Neither the computer nor a human who examines the contents of memory can determine if a value <= hex 7F is being used as a number or as encoded character.