Data Representation and Manipulation.

- humans - decimal (base 10) number system
- computers - binary (base 2) " "

∵ signal - high or low, on or off.

3 representations of numbers

1. unsigned encodings (for numbers ≥ 0)
2. 2's complement " (for signed numbers)
3. floating-point " (for real numbers).

INFORMATION STORAGE

* Hexadecimal Notation

1 byte = 8 bits

$\underline{0000}\underline{0000}_2$ to $\underline{1111}\underline{1111}_2$

$0\ 0_{16}$ $F\ F_{16}$

0x or 0X

Symbols: 0, 1, 2, . . . 9, A, B, C, D, E, F.

eg. 0x1A0F = $0001\ 1010\ 0000\ 1111_2$

* Words

word size (w) = 32 bits (in most machines).

If w = 32 bits, a program's virtual address space can access addresses in the range of 0 to $2^{32}-1$ (ie) 2^{32} bytes.

w = 64 bits, common these days. 2^w bytes.

* Data sizes

C type	32-bit	64-bit
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

short int - atleast 16 bits

int - " 16 "

long int = " 32 "

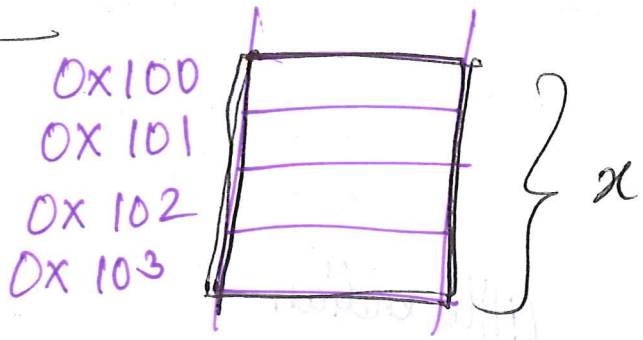
long long int = " 64 "

	Win 16 API	Unix & Unix-like
int	2 bytes	4 bytes.

* Addressing and Byte Ordering

int x;

x = 0x 12 34 56 78



Endian-ness

Little Endian

0x100	78
101	56
102	34
103	12

Least Significant Byte (LSB) comes first. "most" Intel machine

Big Endian

0x100	12
101	34
102	56
103	78

MSB comes first.

"most" IBM, Sun Microsystems = machines.

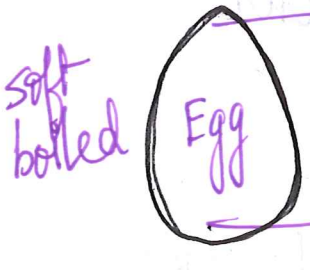
(X) Show code to determine endian-ness of a machine.

Bi-endian – machines than can be configured to work as both little or big endian.

Origin of term "endian-ness"

Gulliver's Travels by Jonathan Swift.

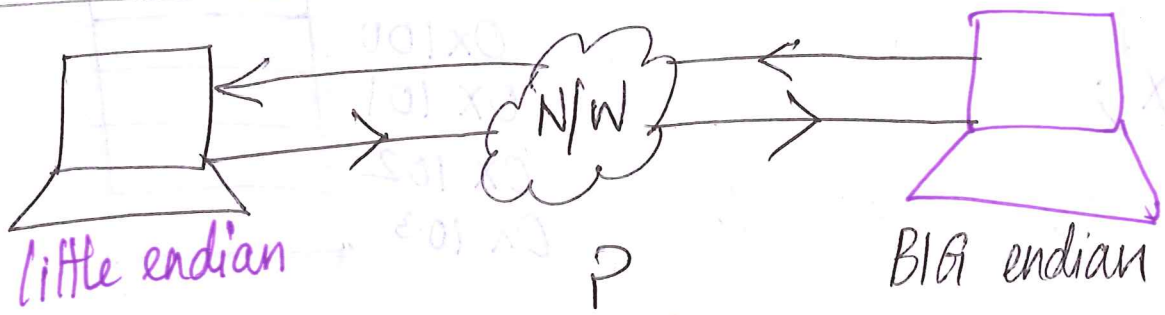
Lilliput and Blefuscu.



On holy wars and a plea for peace. (1981)

– Danny Cohen.
(Networking Protocol).

Problems with endian-ness ④

① 

?

Network Byte Order
(Big-endian)

htons, htonl, ntohs, ntohl.

hton - host to network ; ntoh - network to host.

② Machine-level code

80483bd: 01 05 [64 94 04 08] add %eax,

0x80494 64

[08 04 94 64]

---

* Strings in C

↳ encoded by an array of characters terminated by the NULL character.

eg. "ABC" ⟹ 41 42 43 00

41 → 65 in decimal.

00 → null character.

To see ASCII code: man ascii

Unicode standard for text:

- ASCII - only English
- Unicode - Albanian to Xamtanga.
  - lang. spoken by Xamir people in Ethiopia.
  - 100,000 characters.
  - Universal Character Set - 32-bit representation of chars.
  - UTF-8 - Universal Coded Character Set + Transformation Format - 8 bit.
    - 1,112,064 valid code points.
- Java uses unicode.

BOOLEAN ALGEBRA

- George Boole around 1850.

TRUE - 1
FALSE - 0

op.	Logical op.	boolean op.
NOT	$\neg P$	$\sim p$
AND	$P \wedge Q$	$p \& q$
OR	$P \vee Q$	$p \mid q$
EXOR	$P \oplus Q$	$p \wedge q$

Bit Vectors

strings of 0s and 1s with some fixed length w.

a = 10101

b = 01100

a & b = 00100

Bit-Level Operations in C

~0x00 (or) ~[0000 0000] = [1111 1111] (or) 0xFF

0x01 | 0x02 = 0000 0001 | 0000 0010 = 0000 0011 = 0x03

Masking Operations

x = 0x 89 AB CD EF

x & 0xFF = 0x 00 00 00 EF

↓

Masking of the LSB.

* Logical Operations in C

OR - ||

AND - &&

NOT - !

TRUE - any non-zero value

FALSE - 0.

eg. !0x41 = 0x00

!0x00 = 0x01

0x01 && 0x02 = 0x01

0x01 || 0x03 = 0x01

* short circuit in logical operations.

a && 5/a ⇒ never cause a division by zero.

p && *p++ ⇒ " " " null ptr dereference

* Shift operations in C

x = 01100011

x<<4 = 0011 0000

x>>4 (logical) = 0000 0110

x>>4 (arithmetic) = 0000 0110

y = 1001 0101

y<<4 = 0101 0000

y>>4 (logical) = 0000 1001

y>>4 (arithmetic) = 1111 1001

In C, no precise definition for right shifts.

In Java, x>>k means arithmetic and x>>>k means logical.

Integer Representations

I. Unsigned encoding.

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

$$\underset{2^3}{0}\ \underset{2^2}{1}\ \underset{2^1}{0}\ \underset{2^0}{1} = 1\times 2^2 + 1\times 2^0 = 5.$$

$w = 4$

I. 2's - Complement Encodings

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

$w = 2$

$0000 = 0$
$0001 = 1$

$00 = 0$
$01 = 1$
$10 = -2$
$11 = -1$

$$B2T_4([1011]) = -1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0$$

$$= -8 + 2 + 1 = \underline{-5}$$

$w = 3$

$000 = 0$
$001 = 1$
$010 = 2$
$011 = 3$
$100 = -4$
$101 = -3$
$110 = -2$
$111 = -1$

Note: 1's complement encodings ASIDE

000 = +0	100 = −1
001 = +1	101 = −2
010 = +2	110 = −3
011 = +3	111 = −0

$(-2^{3-1}-1)$ to $(2^{3-1}-1)$.

Important Numbers

Word size w = 32 bits

unsigned ↓

$UMax_{32}$ = 0x FF FF FF FF = 4,294,967,295

$TMax_{32}$ = 0x 7F FF FF FF = 2,147,483,647

↓ Two's complement

$TMin_{32}$ = 0x 80 00 00 00 = −2,147,483,648.

0 = 0x 00 00 00 00 = 0

−1 = 0x FF FF FF FF = −1

$|TMin| = |TMax| + 1$

⇒ There is no positive counterpart for TMin.

$UTMax = 2TMax + 1$

<limits.h> - defines constants with the limits of fundamental integral types for the specific system and compiler implementation used.

eg.
INT_MAX = +2,147,483,647
INT_MIN = -2,147,483,648
UINT_MAX = 4,294,967,295

} on 32-bit unix-like machines

ISO C99 standard - stdint.h

intN_t and uintN_t

↓ ↓

eg. int32_t uint32_t

MACROS: INTN_MIN, INTN_MAX, and UINTN_MAX

eg. INT32_MIN

---

* Signed-Magnitude form

$$B2S_w(\vec{x}) = (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i \right)$$

00 = +0
01 = +1
10 = -0
11 = -1

Two's complement of $x = 2^w - x$ (a single two).
(or)
$-x$ in two's complement

Ones' complement of $x = [1111\ldots1] - x$ (multiple ones).
(or)
$-x$ in ones' complement

---

* Conversions between Signed and Unsigned

```
unsigned u = 4,294,967,295u;   /*U32 Max 32 */
int tu = (int) u;
printf("u = %u, tu = %d\n", u, tu);
```

u = 4294967295, tu = -1

u and tu in hex = 0x FF FF FF FF

- when signed ⟷ unsigned (of same word size)
  ⇒ numeric values might change
  but
  the bit patterns do not.

Signed vs unsigned in C

- most numbers are signed by default.

  eg. 12345 or 0x1A2B

- unsigned : 12345U or 0x1A2Bu.

> if signed and unsigned appear in an operation,
> signed ⟶ unsigned.

problems:

$-1 < 0U \Rightarrow 4{,}294{,}967{,}295U < 0U$

sign-extension & Zero-extension

```
short sx = -12345;                  FALSE X
unsigned short usx = sx;            /* 53191 */
int x = sx;                         /* -12345 */
unsigned ux = usx;                  /* 53191 */
```

sx = -12345 : CF C7

usx = 53191 : CF C7

x = -12345 : FF FF CF C7 → sign extension.

ux = 53191 : 00 00 CF C7 → zero extension.

sign extension

$w = 3$ to $w = 4$

$[101]$ $[1101]$

$-4+1 = -3$ $-8+4+1 = -3$.

$[1111] = [111] = -1$

---

short to unsigned int ?

```
short sx = -12345;      /* CF C7 */
unsigned uy = sx;
```

CF C7 —①→ FF FF CF C7 —②→ (i.e.) interpreted as 4294954951 ✓

① change size

② change from signed to unsigned.

if 1. signed → unsigned
   2. change size
} ⇒ 53,191. ✗

---

* Truncating Numbers

```
int x = 53191;
short sx = (short) x;   /* -12345 */
int y = sx;             /* -12345 */
```

(∵ sign extension).

Advice on Signed vs. Unsigned

```
size_t strlen(const char *s);
```

↓ unsigned int

```
int strlonger(char *s, char *t)
{ return strlen(s) - strlen(t) > 0; }
```

When
- $s > t$, OK
- $s = t$, OK.
- $s < t$, NOT OK. ∵ eg. $1u - 2u > 0u$ ($-1u$)? 0xFFFFFFFF > 0x00000000

Connection

```
return strlen(s) > strlen(t);
```

⇓ TRUE X

(X) Don't mix unsigned int and int! (size_t)

* Security vulnerability in getpeername.

INTEGER ARITHMETIC

1. Unsigned Addition

$w = 4$: 0000 to 1111

(ie) 0 to 15

$\therefore$ max sum = 15 + 15

= 30

needs 5 bits

1 0 1 1 1 0

32 16 8 4 2 1

Word size inflation

w bits + w bits → needs $(w+1)$ bits

$(w+1)$ + $(w+1)$ → needs $(w+2)$ bits.

How to do unsigned arithmetic?

Use modular arithmetic.

eg.
$$\begin{array}{rll} & [1001] & 9 \\ + & [1100] & 12 \\ \hline (1)\,0101 & & 21 \end{array}$$

→ needs 5 bits

21 % 16 = 5

$16 = 2^4$

→ discard the last bit ≡ subtracting 2^w from the sum.

$$x +^{u}_{w} y = \begin{cases} x+y, & x+y < 2^w \\ x+y-2^w, & 2^w \le x+y < 2^{w+1} \end{cases}$$

($+^{u}_{w}$: unsigned addition; w bits)

Overflow of arithmetic operation – Full integer result cannot fit with the word size. limits of the data type

* Two's complement addition

Range of w-bit int values: $-2^{w-1} \le x, y \le 2^{w-1} - 1$

Range of sum: $-2^{w} \le x+y \le 2^{w} - 2$

$\Rightarrow$ require $(w+1)$ bits to store the result.

$\because 2^{w-1} + 2^{w-1}$

$= 2\left(2^{w-1}\right)$

$= 2^1 \cdot 2^{w-1} = 2^{w+1}$

$= \boxed{2^w}$

$x +_w^t y = \begin{cases} x+y-2^w, & 2^{w-1} \le x+y \quad \text{- positive overflow} \\ x+y, & -2^{w-1} \le x+y < 2^{w-1} \quad \text{- Normal} \\ x+y+2^w, & x+y < -2^{(w-1)} \quad \text{- Negative overflow.} \end{cases}$

eg.

x	y	$x+y$	$x +_4^t y$
-8	-5	-13	3
[1000]	[1011]	[10011]	[0011]

$-13 + 16 = 3$

```
1000
1011
-----
10011
```

-8	5	-3	-3
[1000]	[0101]	~~[11101]~~ [11101]	[1101]
2	5	7	7
[0010]	[0101]	[00111]	[0111]

```
1000
0101
-----
1101
```

$10 - 16 = -6$

5	5	10	-6
[0101]	[0101]	[01010]	[1010]

```
0101
0101
-----
1010
```

Two's - Complement Negation.

x's additive inverse is $-x$

$\because \quad x + (-x) = 0$

For all x in $-2^{w-1} < x < 2^{w-1}$, additive inv. is $-x$.

But for $\boxed{x = -2^{w-1}}$, add. inv is -2^{w-1}.

$\because -2^{w-1} + -2^{w-1} = -2^{w}$ negative overflow so add 2^w

$= -2^w + 2^w = 0.$

Unsigned Multiplication

$0 \leq x, y \leq 2^w - 1$

$0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

needs 2w bits to represent.

Solution

Truncation of higher order bits. eg. if $w = 2$

$2^4 - 2^3 + 1 = 16 - 8 + 1 = 9$

need 4 bi

$$\boxed{x *_w^u y = (x \cdot y) \bmod 2^w}$$

eg. $w = 2$ $\quad 3 \times 3 = 9$

$[1\,1] \quad [1\,1] \quad [1\,0\,0\,1]$

$= 9 \,\%\, 2^2 = 9 \,\%\, 4 = 1$

$[0\,0\,1]$

Two's - Complement Multiplication

$w=3$	x	y	$x \cdot y$	$x \cdot y$ truncated
unsigned	5 [101]	3 [011]	15 [001111]	7 [111]
2's comp.	-3 [101]	3 [011]	-9 [110111]	-1 [111]

* Multiplying by constants

$x * 14$

$14 = 2^3 + 2^2 + 2^1$

$(x) = (x \ll 3) + (x \ll 2) + (x \ll 1)$?

∴ Integer multiplication takes 10 or more CPU cycles.

eg. $1 * 14 = 14$

[0001] [1110] = [1110]

[0001] $\ll 3$ = [1000]

[0001] $\ll 2$ = [0100]

[0001] $\ll 1$ = [0010]

+

[1110]

Also: $14 = 2^4 - 2^1$

so, $x * 14 = (x \ll 4) - (x \ll 1)$

[0001] $\ll 4$ = 10000

[0001] $\ll 1$ = 0010 −

1110

if $x = 2$ [0010]

$2 * 14 = 28 = 28 \% 16 = 12$

[0010] [1110] [11100] [1100]

$= [0010] \ll 3 + [0010] \ll 2 + [0010] \ll 1$

$= 10000 + 1000 + 0100 = (1)1100 = [1100]$

12.

Dividing by Powers of Two

Integer division - 30 or more clock cycles.

unsigned division by $2^K \equiv x \gg K$ (logical right shift)

(ie) $x / 2^K$

1^{11} for +ve signed number.

(X) Negative signed numbers (arithmetic right shift).

$-7/2 = -3$

But $[1001]/2^1 = [1001] \gg 1$ ($\because K=1$)

$= [1100] = (-4)$ Wrong!

$\therefore$ Biasing

ie. if $x < 0$, then $\boxed{x = x + (1 \ll K) - 1}$

eg $x = -7 < 0$, $\therefore x = [1001] + ([0001] \ll 1) - [000$

-7

$= [1001] + [0010] - [0001]$

$= [1001] + [0001] =$

$= [1010]$

Now $x/2^1 = [1010] \gg 1 = [1101] = (-3)$ Wow