03_02_Number Bases and Bit Manipulations

“large” units

kilo	10^3	1,000
mega	10^6	1,000,000
giga	10^9	1,000,000,000
tera	10^{12}	1,000,000,000,000
peta	10^{15}	1,000,000,000,000,000
exa	10^{18}	1,000,000,000,000,000,000
zetta	10^{21}	1,000,000,000,000,000,000,000
yotta	10^{24}	1,000,000,000,000,000,000,000,000

units < 1

milli	10^{-3}
micro	10^{-6}
nano	10^{-9}
pico	10^{-12}
femto	10^{-15}
atto	10^{-18}
zepto	10^{-21}
yocto	10^{-24}

some names for large numbers

kilo	10^3	thousand
mega	10^6	million
giga	10^9	billion
tera	10^{12}	trillion
peta	10^{15}	quadrillion
exa	10^{18}	quintillion
zetta	10^{21}	sextillion
yotta	10^{24}	septillion

kilo: 1,000 or 1,024?

powers of 10			powers of 2	
kilo	10^3	1,000	2^{10}	1,024
mega	10^6	1,000,000	2^{20}	1,048,576
giga	10^9	1,000,000,000	2^{30}	1,073,741,824
tera	10^{12}	1,000,000,000,000	2^{40}	1,099,511,627,776
peta	10^{15}	1,000,000,000,000,000	2^{50}	1,125,899,906,842,624
exa	10^{18}	1,000,000,000,000,000,000	2^{60}	1,152,921,504,606,846,976
zetta	10^{21}	1,000,000,000,000,000,000,000	2^{70}	1,180,591,620,717,411,303,424
yotta	10^{24}	1,000,000,000,000,000,000,000,000	2^{80}	1,208,925,819,614,629,174,706,176

usually use:

- powers of 2 for storage
- powers of 10 for just about everything else

proposed prefixes for powers of 2

powers of 10			powers of 2		
kilo	10^3		kibi	2^{10}	1,024
mega	10^6		mebi	2^{20}	1,048,576
giga	10^9		gibi	2^{30}	1,073,741,824
tera	10^{12}		tebi	2^{40}	1,099,511,627,776
peta	10^{15}		pebi	2^{50}	1,125,899,906,842,624
exa	10^{18}		exbi	2^{60}	1,152,921,504,606,846,976
zetta	10^{21}		zebi	2^{70}	1,180,591,620,717,411,303,424
yotta	10^{24}		yobi	2^{80}	1,208,925,819,614,629,174,706,176

- haven’t exactly taken the world by storm

Trick for approximating large numbers

- What is 2^{22}?
  - 2^{20} is about a million
  - 2^{2} is 4
  - 2^{22} is about 4 million
- What is 2^{36}?
  - 2^{30} is about a billion
  - 2^{6} is 64
  - 2^{36} is about 64 billion

kilo	10^3	$\approx 2^{10}$
mega	10^6	$\approx 2^{20}$
giga	10^9	$\approx 2^{30}$
tera	10^{12}	$\approx 2^{40}$
peta	10^{15}	$\approx 2^{50}$
exa	10^{18}	$\approx 2^{60}$
zetta	10^{21}	$\approx 2^{70}$
yotta	10^{24}	$\approx 2^{80}$

powers of 2.
memorize.

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1,024

some powers of 16

16^0	1
16^1	16
16^2	256
16^3	$4{,}096$
16^4	$65{,}536$

- Do have to memorize
- Notice how these are also powers of 2

number system: position is important.

Think grade school. Decimal number 5,342.

$$
\begin{array}{rr}
 & 5{,}000 \\
 & 300 \\
 & 40 \\
+ & 2 \\
\hline
 & 5{,}342
\end{array}
\qquad
\begin{array}{ll}
 & 5 \text{ thousands} \\
 & 3 \text{ hundreds} \\
 & 4 \text{ tens} \\
+ & 2 \text{ ones} \\
\hline
 & 5{,}342
\end{array}
$$

$$
\begin{array}{ll}
 & 5 * 1{,}000 \\
 & 3 * 100 \\
 & 4 * 10 \\
+ & 2 * 1 \\
\hline
 & 5{,}342
\end{array}
\qquad
\begin{array}{ll}
 & 5 * 10^3 \\
 & 3 * 10^2 \\
 & 4 * 10^1 \\
+ & 2 * 10^0 \\
\hline
 & 5{,}342
\end{array}
$$

binary numbers

- positions are powers of 2, not 10.
- binary number 0b1101:

$$\begin{array}{rr} & 1 * 2^3 \\ & 1 * 2^2 \\ & 0 * 2^1 \\ + & 1 * 2^0 \\ \hline & \end{array} \qquad \begin{array}{rr} & 1 * 8 \\ & 1 * 4 \\ & 0 * 2 \\ + & 1 * 1 \\ \hline & 13 \end{array}$$

HEX

- Bit strings get long
- More compact representation
- HEX
  - 4 bits represented by 1 HEX digit

Hex.
Memorize.

dec	hex	bin
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

bits and groups of bits

- bit. ***b***inary dig***it***. can either be a 0 or 1

- 8 bits are a byte
- 4 bits are nibble
- 2 bits are a half-nibble

converting from binary to hex

- Break bit string into groups of 4 (starting from the right)
- Convert each 4-bit group to HEX
- Example:

$$10110100000101$$

$$10\ 1101\ 0000\ 0101$$

$$\begin{array}{cccc} 2 & D & 0 & 5 \\ 0010 & 1101 & 0000 & 0101 \end{array}$$

All three of our bases

dec	bin	hex
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f
16	10000	10
17	10001	11
18	10010	12
19	10011	13

dec	bin	hex
20	10100	14
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1a
27	11011	1b
28	11100	1c
29	11101	1d
30	11110	1e
31	11111	1f
32	100000	20
33	100001	21
34	100010	22
35	100011	23
36	100100	24
37	100101	25
38	100110	26
39	100111	27

dec	bin	hex
40	101000	28
41	101001	29
42	101010	2a
43	101011	2b
44	101100	2c
45	101101	2d
46	101110	2e
47	101111	2f
48	110000	30
49	110001	31
50	110010	32
51	110011	33
52	110100	34
53	110101	35
54	110110	36
55	110111	37
56	111000	38
57	111001	39
58	111010	3a
59	111011	3b

There’s octal too

dec	bin	oct	hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13

dec	bin	oct	hex
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17
24	11000	30	18
25	11001	31	19
26	11010	32	1a
27	11011	33	1b
28	11100	34	1c
29	11101	35	1d
30	11110	36	1e
31	11111	37	1f
32	100000	40	20
33	100001	41	21
34	100010	42	22
35	100011	43	23
36	100100	44	24
37	100101	45	25
38	100110	46	26
39	100111	47	27

dec	bin	oct	hex
40	101000	50	28
41	101001	51	29
42	101010	52	2a
43	101011	53	2b
44	101100	54	2c
45	101101	55	2d
46	101110	56	2e
47	101111	57	2f
48	110000	60	30
49	110001	61	31
50	110010	62	32
51	110011	63	33
52	110100	64	34
53	110101	65	35
54	110110	66	36
55	110111	67	37
56	111000	70	38
57	111001	71	39
58	111010	72	3a
59	111011	73	3b

adding decimal numbers

$$\begin{array}{rrrrr} & 1 & 2 & 3 & 4 \\ + & & 9 & 3 & 7 \\ \hline \end{array}$$

adding decimal numbers

$$
\begin{array}{ccccc}
 & 1 & & 1 & \\
 & 1 & 2 & 3 & 4 \\
+ & & 9 & 3 & 7 \\
\hline
 & 2 & 1 & 7 & 1
\end{array}
$$

binary addition tables

$$\begin{array}{rr} & 0 \\ + & 0 \\ \hline & 0 \end{array} \qquad \begin{array}{rr} & 0 \\ + & 1 \\ \hline & 1 \end{array} \qquad \begin{array}{rr} & 1 \\ + & 0 \\ \hline & 1 \end{array}$$

$$\begin{array}{rrr} & & 1 \\ + & & 1 \\ \hline & 1 & 0 \end{array} \qquad \begin{array}{rrr} & & 1 \\ & & 1 \\ + & & 1 \\ \hline & 1 & 1 \end{array}$$

adding bit strings

$$\begin{array}{rccccc} & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

adding bit strings

$$\begin{array}{rccccc} & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 & 1 \\ \hline & & & & & \end{array}$$

$$\begin{array}{rccccc} & & 1 & 1 & & \\ & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 & 1 \\ \hline & 1 & 1 & 1 & 0 & 1 \end{array}$$

Adding Hex

$$
\begin{array}{cccccc}
 & 6 & B & 9 & 7 & 7_{16} \\
+ & & C & A & 9 & 2_{16} \\
\hline
\end{array}
$$

Adding Hex

$$
\begin{array}{cccccc}
 & 1 & 1 & 1 & & \\
 & 6 & B & 9 & 7 & 7_{16} \\
+ & & C & A & 9 & 2_{16} \\
\hline
 & 7 & 8 & 4 & 0 & 9_{16}
\end{array}
$$

Another Example.

$$
\begin{array}{rrrrrr}
 & 3 & \mathrm{B} & \mathrm{B} & \mathrm{B} & 3_{16} \\
+ & & \mathrm{E} & 8 & 1 & 4_{16} \\
\hline
\end{array}
$$

Another Example. Solution.

$$
\begin{array}{cccccc}
 & 1 & 1 & & & \\
 & 3 & B & B & B & 3_{16} \\
+ & & E & 8 & 1 & 4_{16} \\
\hline
 & 4 & A & 3 & C & 7_{16}
\end{array}
$$

What happens?

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int prod = 200*300*400*500;
  printf("prod = %d\n", prod);

  prod = (200)*(300*400*500);
  printf("prod = %d\n", prod);

  prod = (200*300*400)*500;
  printf("prod = %d\n", prod);

  return 0;
}
```

How about in Java?

```
public class Overflow {
    public static void main(String args[]) {
        int prod = 200*300*400*500;
        System.out.println("prod = " + prod);

        prod = (200)*(300*400*500);
        System.out.println("prod = " + prod);

        prod = (200*300*400)*500;
        System.out.println("prod = " + prod);
    }
}
```

Python?

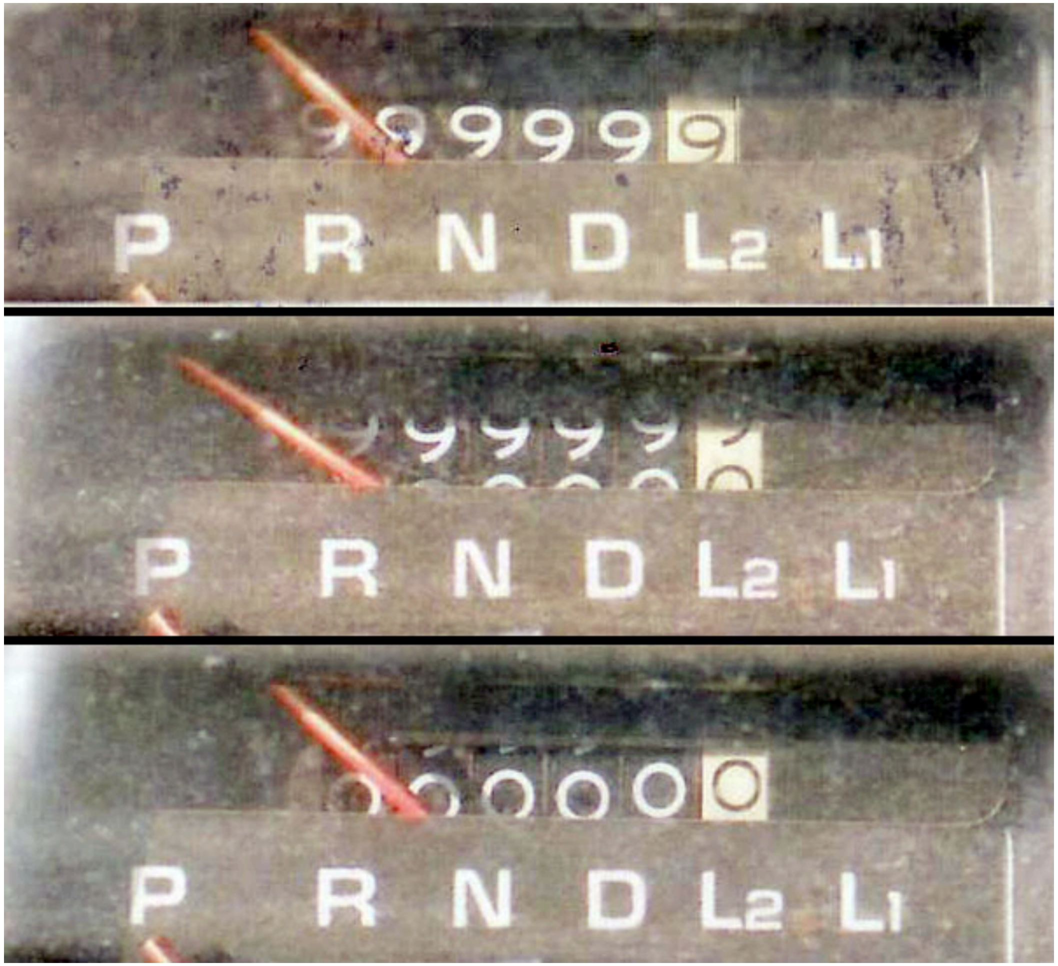
```
#!/usr/bin/env python

prod = 200*300*400*500
print "prod =", prod

prod = (200)*(300*400*500)
print "prod =", prod

prod = (200*300*400)*500;
print "prod =", prod
```

what happens here?



word sizes

- think *width* of the odometer
- word size – fundamental system parameter
- *nominal* size of an int, pointer
- sizes:
  – most machines today: 4 bytes
  – high-end machines: 8 bytes
  – so how much RAM can we address on each?
- be careful, Intel program documentation:
  – word = 16 bits

aside: Intel programmer terms

byte	1 byte
word	2 bytes
doubleword	4 bytes
quadword	8 bytes
double quadword	16 bytes

sizes

```
printf("sizeof(char)=%lu\n", sizeof(char));
printf("sizeof(short)=%lu\n", sizeof(short));
printf("sizeof(int)=%lu\n", sizeof(int));
printf("sizeof(long)=%lu\n", sizeof(long));
printf("sizeof(void*)=%lu\n", sizeof(void*));
```

results

when I run on my laptop:

```
sizeof(char)=1
sizeof(short)=2
sizeof(int)=4
sizeof(long)=4
sizeof(void*)=4
```

```
sizeof(char)=1
sizeof(short)=2
sizeof(int)=4
sizeof(long)=8
sizeof(void*)=8
```

when I run on Temple CIS dept Linux box:

NOT (Bitwise Negation)

A	$\sim A$
0	1
1	0

NOT (Bitwise Negation)

A	$\tilde{}A$
0	1
1	0

Example:

A	11010010
$\tilde{}A$	00101101

AND

A	B	$A\&B$
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	$A\&B$
0	0	0
0	1	0
1	0	0
1	1	1

Example:

$$\begin{array}{rr} A & 11010010 \\ B & 01111010 \\ \hline A\&B & 01010010 \end{array}$$

OR

A	B	$A \vert B$
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	B	$A \vert B$
0	0	0
0	1	1
1	0	1
1	1	1

Example:

$$\begin{array}{rl} A & 11110000 \\ B & 00001111 \\ \hline A|B & 11111111 \end{array}$$

XOR

A	B	$A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

XOR

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

Example:

$$\begin{array}{rr} A & 11111100 \\ B & 00111111 \\ \hline A\wedge B & 11000011 \end{array}$$

More on XOR

a XOR a = ?

a XOR b XOR b = ?

More on XOR

a XOR a = 0

a XOR b XOR b = a

interesting trick

```
void swap1(int *a, int *b) {
  int tmp = *a;
  *a=*b;
  *b=tmp;
}

void swap2(int *a, int *b) {
  *a^=*b;  /* a = a XOR b */
  *b^=*a;
  *a^=*b;
}
```

some bit operators in C

```
printf("a=%d, NOT a=%d\n", a, ~a);
printf("a=%d, b=%d, a AND b = %d\n", a, b, a&b);
printf("a=%d, b=%d, a OR b = %d\n", a, b, a|b);
printf("a=%d, b=%d, a XOR b = %d\n", a, b, a^b);
```

```c
#include <stdio.h>

#define SMART 0x0001
#define HANDSOME 0x0010
#define FUNNY 0x0100
#define FUN_TO_BE_AROUND 0x1000

typedef int attr;

int isSmart(attr);
int isHandsome(attr);
int isFunny(attr);
int isFunToBeAround(attr);

int main(void)
{
  attr you = SMART | HANDSOME | FUNNY;

  printf("your attributes:");

  if (isSmart(you))
    printf(" smart,");
  else
    printf(" not smart,");

  if (isHandsome(you))
    printf(" handsome,");
  else
    printf(" not handsome,");

  if (isFunny(you))
    printf(" funny,");
  else
    printf(" not funny,");

  if (isFunToBeAround(you))
    printf(" fun to be around\n");
  else
    printf(" not fun to be around\n");

  return 0;
}

int isSmart(attr a)
{
  return a & SMART;
}

int isHandsome(attr a)
{
  return a & HANDSOME;
}

int isFunny(attr a)
{
  return a & FUNNY;
}

int isFunToBeAround(attr a)
{
  return a & FUN_TO_BE_AROUND;
}
```

reminder about logical operators

- 0x0 is false
- anything else is true

What is the difference between ! and ~?

The ! symbol represents **boolean or logical negation**.

-For any value of x other than zero, !x evaluates to zero or false, and when x is zero, !x evaluates to one, or true.

The ~ symbol represents **bitwise negation**.

-Each bit in the value is toggled, so for a 16-bit x == 0xA5A5, ~x would evaluate to 0x5A5A.

don’t confuse logical and bit ops!

```
unsigned char x=0x31;

printf("~x=0x%x\n", ~x);
printf("~~x=0x%x\n", ~~x);

printf("!x=0x%x\n", !x);
printf("!!x=0x%x\n", !!x);
```

don't confuse logical and bit ops!

```
unsigned char x=0x31;

printf("~x=0x%x\n", ~x);
printf("~~x=0x%x\n", ~~x);

printf("!x=0x%x\n", !x);
printf("!!x=0x%x\n", !!x);
```

```
~x=0xffffffce
~~x=0x31
!x=0x0
!!x=0x1
```

don’t confuse logical and bit ops!

```
~x=0xffffffce
~~x=0x31
!x=0x0
!!x=0x1
```

$$0\text{x}31 = 00000000000000000000000000110001_2$$

$$\tilde{}0\text{x}31 = 11111111111111111111111111001110_2$$

What's this number?

657

What's this number?

657

six hundred fifty seven

not seven hundred fifty six?

What’s this number?

657

- Most significant digit first
  - six hundred fifty seven
- Least significant digit first
  - seven hundred fifty six

Byte ordering

- Big endian
  - most significant byte first
  - Examples: SPARC, old PowerPC Macs, Internet (aka "network byte order")
- Little endian
  - least significant byte first
  - Examples: x86, DEC Alpha

Byte ordering

- machine with 4 byte ints
- int i=0x01234567

big endian

address	**value**
1000	01
1001	23
1002	45
1003	67

little endian

address	**value**
1000	67
1001	45
1002	23
1003	01

Reminder:

Don’t confuse byte ordering with bit ordering