COMPUTERWORLD

Bits, bit operationer, integers og floating point

Denne artikel beskriver hvordan data gemmes som bits og hvordan man kan manipulere med bits.

Den forudsætter en vis programmerings forståelse og erfaring. Kode eksemplerne er i C/C++/Java/C# syntax.

Skrevet den **16. Feb 2010** af **arne_v** I kategorien **Programmering / Generelt** | ★★★☆☆

Historie:
V1.0 - 12/11/2004 - original
V1.1 - 16/02/2010 - smårettelser

integer

Integers er heltal som normalt opbevares i 1, 2 eller 4 bytes i binært format.

I C/C++/Java/C# familien af sprog kan man angive integer konstanter i decimal (10 tals systemet), hexadecimal (16 tals systemet) eller oktal (8 tals systemet).

I hexadecimal (16 tals systemet) bruger man A-F for 10-15.

31 er en decimal konstant (3*10+1=31)
0x1F er en hexdecimal konstant (med samme værdi da 1*16+15=31)
037 er en oktal konstant (med samme værdi da 3*8+7=31)

Når man regner med bits bruger man ofte hexadecimal fordi det er nemt at omregne til binært format. Et hexadecimalt ciffer svarer til 4 binære cifre.

0x1F = 00011111 binært

Negative tal repræsenteres normalt i noget man kalder twos complement. Man negerer (vender) alle bits og ligger 1 til.

så -0x07 = -00000111 binært = 11111000 + 1 binært = 11111001 binært = 0xF9.

Derfor vil negative tal altid have højeste bit sat. Og -1 vil altid have alle bit sat.

Derudover kan tal repræsenteres i tekst form f.eks. "31".

Bemærk at binære former og tekst former er helt forskelligt repræsenteret.

I C på en little endian (se forklaring senere) maskine med 4 byte int's vil 12345 i binær form være bytes 0x39 0x30 0x00 0x00 mens det i

tekst form vil være 0x31 0x32 0x33 0x34 0x35 0x00.

shift

Man kan shifte integers.

Det består i at man rykker bitsene enten mod højre eller
mod venstre.

0x07 >> 1 = 00000111 >> 1 binært = 00000011 binært = 0x03
0x07 << 1 = 00000111 << 1 binært = 00001110 binært = 0x0E

Den opmærksomme læser har nok genneskuet at >> 1 svarer til / 2
og << 2 svarer til * 2.

Og tilsvarende vil >> n svare til / pow(2,n) og << n svare
til * pow(2,n).

Vær opmærksom på at mens man ved venstre shift sætter 0'er ind:

0x07 << 1 = 00000111 << 1 binært = 00001110 binært = 0x0E

Så sætter man samme bit som højeste bit ind ved højre shift:

0x07 >> 1 = 00000111 >> 1 binært = 00000011 binært = 0x03
0xF9 >> 1 = 11111001 >> 1 binært = 11111100 binært = 0xFC

Vær meget forsigtg med højre shift på negative tal !

basale bit operationer

De basale bit operationer er AND, OR, XOR og NOT.
De udføres på enkelte bits af gangen.

AND:

```
v1   v2   v1 AND v2
1    1       1
1    0       0
0    1       0
0    0       0
```

I C/C++/Java/C# familien af sprog bruges & som operator for
bitvis AND i modsætning til && som er logisk AND.

Eksempel:
  0x07 & 0x0E giver 0x06

OR:

```
v1   v2   v1 OR v2
1    1       1
1    0       1
```

```
0     1        1
0     0        0
```

I C/C++/Java/C# familien af sprog bruges | som operator for
bitvis OR i modsætning til || som er logisk OR.

Eksempel:
 0x07 | 0x0E giver 0x0F

XOR:

```
v1    v2    v1 XOR v2
1     1        0
1     0        1
0     1        1
0     0        0
```

I C/C++/Java/C# familien af sprog bruges ^ som operator for XOR.

Eksempel:
 0x07 ^ 0x0E giver 0x09

NOT:

```
v     NOT v
1       0
0       1
```

I C/C++/Java/C# familien af sprog bruges ~ som operator for NEGATE.

Eksempel:
 ~0x07 giver 0xF8

logiske bit operationer

v = integer hvor bit bliver set/clear/test
nb = bit nummer startende med 0

Bit set:

```
int bitset(int v, int nb) {
    return (v | (1 << nb));
}
```

Bit clear:

```
int bitclr(int v, int nb) {
```

```
   return (v & ~(1 << nb));
}
```

Bit test:

```
boolean bittst(int v, int nb) {
    return ((v & (1 << nb)) != 0);
}
```

[Java bruger boolean, C++/C# bruger bool, C bruger int]

Komplet Java eksempel:

```
public class Bits {
    int bitset(int v, int nb) {
        return (v | (1 << nb));
    }
    int bitclr(int v, int nb) {
        return (v & ~(1 << nb));
    }
    boolean bittst(int v, int nb) {
        return ((v & (1 << nb)) != 0);
    }
    void test() {
        int v = 0;
        v = bitset(v, 1);
        v = bitset(v, 2);
        v = bitset(v, 3);
        v = bitclr(v, 2);
        System.out.println(v);
        System.out.println(bittst(v, 1));
        System.out.println(bittst(v, 2));
    }
    public static void main(String[] args) {
        (new Bits()).test();
    }
}
```

Output:

```
10
true
false
```

little/big endian

Integers > 1 bytes kan skrives til disk/net på 2 forskellige måder.

De kaldes henholdsvis little endian og big endian (big endian kaldes også net order).

Little endian betyder at den mindst signifikante byte kommer først.

Big endian betyder at den mest signifikante byte kommer først.

En 4 byte integer med værdien 7 decimal = 0x00000007 hex vil blive til bytes som følger:

little endian - 0x07 0x00 0x00 0x00

big endian - 0x00 0x00 0x00x 0x07

Det er åbenlyst at det er vigtigt ved udveksling af binære data, at man er enig om little endian versus big endian.

Little endian CPU'er:
- Intel x86 (inkl. AMD kompatible)
- Intel Itanium*
- DEC VAX
- DEC Alpha*

Big endian CPU'er:
- Motorola 680x0
- SUN SPARC*
- HP PA
- IBM 360/370/390
- IBM PowerPC*

(de med * markerede CPU'er kan faktisk operere i begge modes)

floating point

Idag bruger alle CPU'er den såkaldte IEEE standard for floating point.

type	range	præcision	bits	sign bits	exponent bits	exponent bias	fraction bits
float	-3.4E38..3.4E38	ca. 7 cifre	32	1	8	127	23
double	-1.8E308..1.8E308	ca. 16 cifre	64	1	11	1023	52

værdi = pow(-1,sign) * pow(2,exponent-exponent bias) * 1.fraction

eller

værdi = +/- pow(2,e) * (1 + b0/2 + b1/4 + b3/8 + ...)

Derudover findes der specielle bit værdier for uendelig, ikke et tal (NaN) etc..

Man skal være klar over at floating point har nogle specielle egenskaber:

1) fordi der kun er et endeligt antal bit mønstre i 32 eller 64 bit og der er uendeligt mange reelle tal i ranget, så kan man ikke repræsentere alle reelle tal eksakt i floating point

2) ligesom man med decimal har problemer med 1/3 der ikke kan repræsenteres eksakt i et endeligt antal cifre, så har man i floating point problemer med 1/10 der ikke kan repræsenteres eksakt, og derfor kan man ikke regne med at "pæne" decimale tal kan repræsenteres eksakt i floating point

3) fordi der sker en afrunding fra matematisk resultat til nærmeste floating point værdi ved beregning så kan der opstå mindre (og i nogle ekstreme tilfælde endda større regne unøjagtigheder)

Brug derfor kun floating point til data hvor du kan leve med lidt unøjagtighed ude på decimalerne. Det giver f.eks. ingen mening at diskutere om du har 30.5 km eller 30.500001 km fra hvor du bor til på arbejde (forskelen er 1 mm).

Men brug aldrig floating point til penge. Bogholdere og revisorer har sære ideer om at regnskaber skal stemme til sidste øre.

Kommentar af simonvalter d. 13. Nov 2004 | 1

Du skal nok have lidt baggrundsviden om dette for at kunne følge det men om ikke andet så kan det give lidt inspiration til at lære mere. Ok artikel.

Hvis man vil i dybden med ting som nand/nor gates osv skal man nok have fat i en bog om maskinarkitektur.

Kommentar af newage (nedlagt brugerprofil) d. 13. Nov 2004 | 2

Overordnet en god, men lidt overfladisk artikel - men jeg har nok bare erhvervet mig en arbejdsskade :)

Du starter fint ud med integers og beskriver et par forskellige radixes (binær, octal, decimal, hexa)
Du kunne eventuelt ha' vist hvordan man konverterer fra det ene radix til et andet.

Du beskriver også fint negative tal repræsenteret binært. Men jeg savner noget mere kød - og du kommer slet ikke ind på 1's komplement, selvom 2's komplement - som du skriver - er den der bruges i nutidens maskiner.

Du beskriver bitshifts kort, men godt. Jeg savner lidt detaljer om forskellen mellem aritmetisk og logisk shift.

Du taler lidt om bitoperationer, hvor du nævner AND, OR, XOR og NEGATE (NOT) som de mest basale. Dette er jeg ikke helt enig i.
Jeg vil mene at de (5) basale gates er NOT, NAND, NOR, AND, OR - og man kunne så komme ind på comparators, som bruger en 6. gate XOR.
Desuden bruger man oftest NAND/NOR gates istedet for AND/OR, da NAND/NOR kun kræver 2 transistorer hvor AND/OR kræver 3.
Desuden kan man lave AND/OR operationer kun vha. NAND/NOR gates.

Du skriver lidt om big/little endian - kort men godt.

Du beskriver overfladisk floating point - og nævner IEEE 754 standarden. Jeg kunne godt tænke mig lidt

mere teori.

Ting jeg mangler ved artiklen:

- Integer overflows
- konvertering af 16-bit tal til 32-bit vice versa (kan der opstå der problemer?)

Tilslut vil jeg sige, at artiklen er god til den normale bruger (Slår mig selv i hovedet).

Kommentar af andr3as d. 13. Nov 2004 | 3