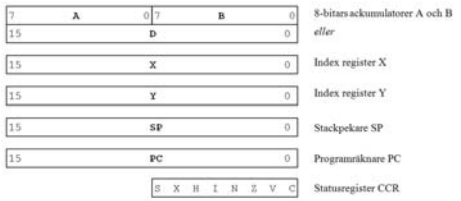
Maskinorienterad Programmering 2010/11

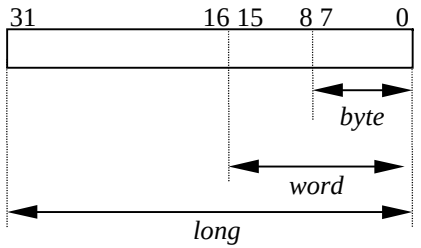
Maskinnära programmering – C och assemblerspråk

Ur innehållet:

- 32-bitars aritmetik med 16-bitars processor
- IEEE754 – flyttal

CPU12, ordlängder och datatyper






```
char     c;    /* 8-bitars datatyp, storlek byte */
short    s;    /* 16-bitars datatyp, storlek word */
long     l;    /* 32-bitars datatyp, storlek long */
int      i;    /* storlek implementationsberoende */
```

Lämpliga arbetsregister för **`short`** och **`char`** är **`D`** *respektive* **`B`**

32 bitars datatyper ryms ej i något CPU12-register.

Aritmetiska operatorer (+,-,×,/,%)

Addition
Subtraktion
Multiplikation
Heltalsdivision
Restdivision

I den bästa av världar kan alla operatorer användas på godtyckliga sifferkombinationer.

Dagens programspråk lever dessvärre inte i den bästa världen...

Operator OCH datatyp avgör val av maskininstruktion, eller sekvens av instruktioner...

Addition ...

Pseudo språk:

```
char  ca,cb,cc;
        ...
ca = cb + cc;
```

8 bitar

```
...
LDAB    cb      ; operand 1
ADDB    cc      ; adderas
STAB    ca      ; skriv i minnet
```

Pseudo språk:

```
short sa,sb,sc;
        ...
sa = sb + sc;
```

16 bitar

```
LDD     sb      ; operand 1
ADDD    sc      ; adderas
STD     sa      ; skriv i minnet
```

32 bitar

Pseudo språk:

```
long la,lb,lc;
        ...
la = lb + lc;
```

```
LDD     lb+2    ; minst signifikanta "word" av b
ADDD    lc+2    ; adderas till minst signifikanta "word" av c
STD     la+2    ; tilldela, minst signifikanta "word"
LDD     lb      ; mest signifikanta "word" av b
ADCB    lc+1    ; adderas till låg byte av mest signifikanta
                ; "word" av c
ADCA    lc      ; adderas till hög byte av mest signifikanta
                ; "word" av c
STD     la      ; tilldela, mest signifikanta "word"
```

Subtraktion ...

Pseudo språk:

```
char  ca,cb,cc;
        ...
ca = cb - cc;
```

8 bitar

```
        ...
        LDAB    cb      ; operand 1
        SUBB    cc      ; subtraheras
        STAB    ca      ; skriv i minnet
```

Pseudo språk:

```
short sa,sb,sc;
        ...
sa = sb - sc;
```

16 bitar

```
        LDD     sb      ; operand 1
        SUBD    sc      ; subtraheras
        STD     sa      ; skriv i minnet
```

32 bitar

Pseudo språk:

```
long la,lb,lc;
        ...
la = lb - lc;
```

```
        LDD     lb+2    ; minst signifikanta "word" av b
        SUBD    lc+2    ; subtraheras från minst signifikanta "word" av c
        STD     la+2    ; tilldela, minst signifikanta "word"
        LDD     lb      ; mest signifikanta "word" av b
        SBCB    lc+1    ; subtrahera låg byte av mest signifikanta
                        ; "word" av c
        SBCA    lc      ; subtrahera hög byte av mest signifikanta
                        ; "word" av c
        STD     la      ; tilldela, mest signifikanta "word"
```


Multiplikation (P=X×Y) tal utan tecken, med addition/skift, X>0, Y>0.

```
X=6=            =0110  (multiplikand)
Y=5=y3y2y1y0    =0101  (multiplikator)

PP(0)           0000           y0=1⇒ADD X
              + 0110
                0110           skifta
PP(1)           0011 0         y1=0⇒ADD 0
              + 0000
                0011 0         skifta
PP(2)           0001 10        y2=1⇒ADD X
              + 0110
                0111 10        skifta
PP(3)           0011 110       y3=0⇒ADD 0
              + 0000
                0011 110       skifta

P=PP(4)=        00011110 = 2^4+2^3+2^2+2^1 = 30
```


Multiplikation (P=X×Y) tal med tecken, med addition/aritmetiskt skift, X<0, Y<0.

```
X=-6=           =1010 (multiplikand)-X = 0110
Y=-5=y3y2y1y0   =1011 (multiplikator)
```

Observera hur vi här använder en extra teckenbit, (5-bitars tal i operationen)

```
PP(0)           00000          y0=1⇒ADD X
              + 11010
                11010          skifta aritmetiskt
PP(1)           111010         y1=1⇒ADD X
              + 11010
                101110         skifta aritmetiskt
PP(2)           1101110        y2=0⇒ADD 0
              + 00000
                1101110        skifta aritmetiskt
PP(3)           11101110       y3=1⇒ADD -X
              + 00110
                00011110       skifta aritmetiskt

P=PP(4)=        00011110 = 30
```


Slutsatser:

- Multiplikation utförs enkelt med grundläggande operationer addition och skift.
- Vi kan **inte** använda exakt samma algoritm på tal med respektive utan tecken.
- Resultatet av en ”unsigned” multiplikation av 2 st. *n*-bitars tal kräver $2 \times n$ bitars register.
- Resultatet av en ”signed” multiplikation av 2 st n-bitars tal kräver $(2 \times n) - 1$ bitars register.

Implementering ”papper och penna-metod”

```
long   mul32 ( long a, long b)
{      /* Multiplikation med "skift/add" */
       long   result, mask;
       int    i;
       mask = 1;
       result = 0;
       for( i = 0; i<32; i++ )
       {
              if ( mask & a )
                     result = result + b ;
              b = b << 1;
              mask   = mask << 1;
       }
       return result;
}
```

Fungerande men långsam metod...

HCS12, 8-bitars multiplikation...

Pseudo språk:

```
unsigned char  ca,cb,cc;
    ...
ca = cb * cc;
```

Assemblerspråk:

```
LDAB   cb     ; operand 1
LDAA   cc     ; adderas
MUL           ; multiplicera
STAB   ca     ; skriv i minnet
```

Endast ”unsigned”...

HCS12, 16-bitars multiplikation (unsigned)...

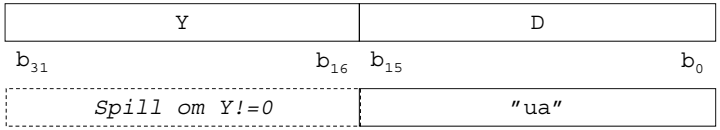
Pseudo språk:

```
unsigned short ua,ub,uc;
    ...
ua = ub * uc;
```

Assemblerspråk:

```
LDD    ub     ; operand 1
LDY    uc     ; operand 2
EMUL          ; multiplicera
STD    ua     ; skriv i minnet
```

Resultat (2n bitar)



HCS12, 16-bitars multiplikation (signed) ...

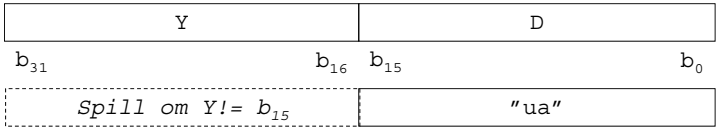
Pseudo språk:

```
short ua,ub,uc;
    ...
ua = ub * uc;
```

Assemblerspråk:

```
LDD    ub     ; operand 1
LDY    uc     ; operand 2
EMULS         ; multiplicera
STD    ua     ; skriv i minnet
```

Resultat (2n bitar)



HCS12, 32-bitars multiplikation, med EMUL

Antag *a*, *b* 32-bitars tal, skriv:

$a = ah \times 2^{16} + al$,

$b = bh \times 2^{16} + bl$,

Det gäller då att:

$a \times b = (ah \times 2^{16} + al) \times (bh \times 2^{16} + bl) =$

$2^{32} (ah \times bh) + 2^{16} (ah \times bl + al \times bh) + (al \times bl)$

Detta är samma sak som:

$(ah \times bh) << 32 + (ah \times bl + al \times bh) << 16 + (al \times bl)$

vi kan därför skriva:

$a \times b = (ah \times bl + al \times bh) << 16 + (al \times bl)$

Endast ”unsigned”...

Implementering...

```
unsigned long mulu32 ( unsigned long a, unsigned long b)
{
        unsigned long  result;
        unsigned short ah,al,bh,bl;
        ah =  (unsigned short )( a >> 16 );
        al =   (unsigned short ) a ;
        bh =  (unsigned short )( b >> 16 );
        bl =   (unsigned short ) b ;
        result = (((unsigned long)( ah*bl + al*bh ))<< 16 ) + ( al*bl );
        return result;
}
long muls32 (long a, long b)
{
        long r;
        r = mulu32 ( ((a < 0) ? -a : a),((b < 0) ? -b : b) );
        if ( (a < 0) ^ (b < 0))
                return -r;
        else
                return r;
}
```


Division

En division kan skrivas som:

$$\frac{X}{Y} = Q + \frac{R}{Y}$$

Där:

- X är dividend
- Y är divisor
- Q är kvot, resultatet av heltalsdivisionen X/Y.
- R är resten, resultatet av modulusdivisionen X mod Y.

Av sambandet framgår att resten kan uttryckas: R = X - Q×Y

Exempel: Decimal division, återställning av resten, 3967/15

```
X = 3967
Y = 15
                    R  = X - Q×Y
     0264,4
15  |3967,0      3967=3967-0×15                          Utgångsläge
    -0
     3967,0      3967=3967-(0×10^3)×15                        steg 1
    -30
      967,0       967=3967-(0×10^3+2*10^2)×15                 steg 2
    - 90
       67,0        67=3967-(0×10^3+2×10^2+6×10^1)×15          steg 3
    -  60,0
        7,0         7=3967-(0×10^3+2×10^2+6×10^1+4×10^0)×15    steg 4
    -   6,0
        1,0         1=3967-(0×10^3+2×10^2+6×10^1+4×10^0+4×10^-1)×15  steg 5

DVS.    3967/15 = 264,4 + 10/15 × 10^-1
```

En divisionsalgoritm

Algoritm: *Division med återställning*

$R = X - Q \times Y$

$Q = q_0 q_1 q_2 \cdots q_{n-1}$

n=antal kvotbitar att beräkna

$R_0 = X$

$R_1 = R_0 - q_0 \times Y$ $\quad q_0 = 1$ om $R_1 \geq 0$, $q_0 = 0$ annars

för $i = 2..n$

$R_i = 2 \times R_{i-1} - q_i \times Y$ $\quad q_i = 1$ om $R_i \geq 0$, $q_i = 0$ annars

Anm: För binära tal reduceras operationen $q_i \times Y$ till ADD Y.

Implementering: 32-bitars ”unsigned” heltalsdivision

```
unsigned long divu32 (unsigned long a, unsigned long b)
{
        unsigned long rest = 0L;
        unsigned char count = 31;
        unsigned char c;

        do{
                if( a & 0x80000000 )
                        c = 1;
                else
                        c = 0;
                a = a << 1;
                rest = rest << 1;
                if(c)
                        rest = rest | 1L;

                if(rest >= b){
                        rest = rest - b;
                        a = a | 1L;
                }
        } while(count--);
        return a;
}
```


Implementering: 32-bitars ”signed” heltalsdivision

```
long divs32 (long a, long b)
{
      long r;

      r = divu32((a < 0 ? -a : a),(b < 0 ? -b : b));

      if ( (a < 0) ^ (b < 0))
            return -r;
      else
            return r;
}
```


Implementering: 32-bitars ”unsigned” restdivision

```
unsigned long  modu32 (unsigned long a ,unsigned long b)
{
      unsigned long c = a/b;  /* heltalsdivision */
      return ( a - b * c );
}
```

Resultatet vid ”signed” restdivision är implementationsberoende....

Normaliserat flyttalsformat

Ett flyttal uttrycks allmänt som:

$$(-1)^S \ M \times 2^E$$

där:

S (*sign*) är teckenbiten för flyttalet

S=0 anger ett positivt flyttal ty $(-1)^0 = 1$

S=1 anger ett negativt flyttal ty $(-1)^1 = -1$

M utgör talets mantissa

E utgör talets exponent.

Exponenten väljs från någon representation med inbyggt tecken.

Det är värt att notera att för ett *normaliserat* flyttal på binär form gäller att:

$$(M)_2 = 1.xxxxx$$

dvs. mantissans första siffra är alltid är 1.

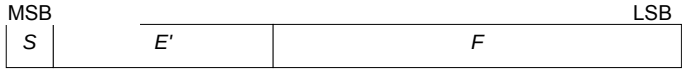
IEEE-754 flyttal standard specificerar:

- flyttalsformat
- noggrannhet i resultat från aritmetiska operationer
- omvandling mellan heltal och flyttal
- omvandling till/från andra flyttalsformat
- avrundning
- undantagshantering vid operationer på flyttal, exempelvis division med 0 och resultat som ej kan representeras av flyttalsformatet.

Standarden definierar fyra olika flyttalsformat:

- *Single format*, totalt 32 bitar
- *Double format*, totalt 64 bitar
- *Single extended format*, antalet bitar är implementationsberoende
- *Double extended format*, totalt 80 bitar

IEEE-754 flyttalsformat:



där:

• F (*fractional part*) kallas också *signifikand*, är den normaliserade mantissan × 2, dvs. den första (implicita) ettan i mantissan utelämnas i representationen och mantissan skiftas ett steg till vänster. På så sätt uppnår vi ytterligare noggrannhet eftersom vi får ytterligare en siffra i det lagrade talet.

• E' (*karakteristika*) är exponenten uttryckt på *excess(n)* format, *n* beror på vilket av de fyra formaten som avses.

• S (*sign*) är teckenbit för F.

Följande tabell anger hur de olika formaten disponeras enligt standarden:

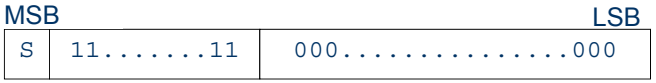
Format	S	E	F
Single	1 bit	8 bitar excess(127)	23 bitar
Double	1 bit	11 bitar excess(1023)	52 bitar
Extended	1 bit	15 bitar excess(2047)	64 bitar

Speciella kodningar

”Första ordningens singulariteter”

± 0

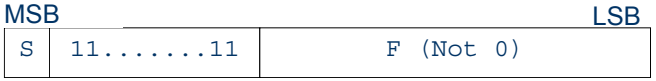
± ∞



”Andra ordningens singulariteter”

Denormaliserat

”Not a Number”



Typomvandlingar

```
unsigned long ui;
float f;

f = ui;    /* Heltal till flyttal */
ui = f;    /* Flyttal till heltal */
```

Vissa processorer har en speciell enhet för hantering av flyttal (floating-point coprocessor). Sådana enheter har maskininstruktioner för såväl typomvandlingar som aritmetiska flyttalsoperationer.

Typomvandlingar

Då vi inte har hårdvarustöd för flyttal, dvs. speciella maskininstruktioner, måste vi tillhandahålla ***programbibliotek*** *för alla operationer på flyttal...*

```
unsigned long ui;
float f;

f = ui;        /* Heltal till flyttal */
ui = f;        /* Flyttal till heltal */
```

float → signed long int	(ftol, "float to long")
unsigned long int → float	(ultof, "unsigned long to float")

Typomvandlingar

Då vi inte har hårdvarustöd för flyttal, dvs. speciella maskininstruktioner, måste vi tillhandahålla ***programbibliotek*** *för alla operationer på flyttal...*

```
unsigned long ui;
float f;

f = ui;        /* Heltal till flyttal */
ui = f;        /* Flyttal till heltal */
```

float → signed long int	(ftol, "float to long")
unsigned long int → float	(ultof, "unsigned long to float")

Exempel: Skriv talet $(2{,}52)_{10}\ 10^4$ som ett IEEE754-single format flyttal

Omvandling till binär, normaliserad form ger:

$$(2{,}52)_{10}\ 10^4 = (1.100\ 0100\ 1110\ 000)_2 \times 2^{14}$$

Mantissan ska ha totalt 24 bitar (varav 23 lagras)

$$M = (1.100\ 0100\ 1110\ 0000\ 0000\ 0000)_2$$

Signifikanden F dvs. den del av mantissan som ska lagras fås om vi stryker den mest signifikanta ettan, vi har då:

$$F = (100\ 0100\ 1110\ 0000\ 0000\ 0000)_2$$

Exponenten uttrycks av karakteristikan E', excess(127) kod, dvs. $E' = E+127$, där E betecknar exponenten i talet vi utgår från (2^{14}) dvs. E=14 varför

$$E' = 14 + 127 = 141 = (1000\ 1101)_2$$

eftersom talet är positivt får vi $S = 0$. Vi får slutligen:

$$(2{,}52)_{10}\ 10^4 = (\mathbf{0100\ 0110\ 1100\ 0100\ 1110\ 0000\ 0000\ 0000})_{SFP}$$

Algoritm: Omvandling av 32 bitars tal *x*, utan tecken, till flyttal *sfp*, ***sfp ← x***

1. *sfp*(*S*)=0; *x'* ← *x*;
2: Bestäm *signifikand*, bitar x_{32} .. x_{24}, ryms ej i mantissan, dividera därför successivt *x'* med 2, justera exponenten:

```
while( x' > (2^24-1) )
begin
        sfp(E') ← sfp(E') + 1;
        x'  ← ( x'  >> 1 );
end
```

4: Normalisera mantissa, för en normaliserad mantissa ska bit 23 vara 1, om inte så är fallet vänsterskiftar vi *x'* tills detta är sant. Observera att algoritmen här förutsätter att *x'* är skild från 0.

```
while( x' < 2^23 )
begin
        x'  ← ( x'  << 1 );
end
```

Mantissan är nu normaliserad, eftersom den inledande ettan (bit x_{23}') inte lagras) nollställer vi den

```
x23' ← 0
sfp(M) ← x'
```

5: Sätt samman flyttalet:

```
sfp ←  ( sfp(S) << 31 ) | ( sfp(E') << 23 ) | sfp(M)
```


Algoritm: Omvandling av flyttal *sfp* till 32 bitars tal *x*, med tecken, ***x ← sfp***

1. Extrahera tecken
   sign = *sfp*(*0*);
2. Extrahera exponenten (Karakteristika med subtraherad förskjutning)
   exp = *sfp*(*E'*) - 127 - 23;
   - Om denna exponent är större än 8 är talet för stort för att kunna representeras
     if (*exp* > 8) *x* ← LONG_MAX;
   - Om denna exponent är mindre än -25 är talet för litet för att kunna representeras
     if (*exp* < -24) *x* ← LONG_MIN;
3. Extrahera mantissa (signifikand med inledande etta)
   mant = *sfp*(*M*) | b_{23}
4. Skifta mantissan tills exponenten blir 0
   if (*exp* > 0) (*mant* << exp);
   if (*exp* < 0) (*mant* >> exp);
5. Returnera heltal med rätt tecken
   if(sign) *x* ← -(*mant*);
   else *x* ← *mant*;

Implementering

Då alla tre IEEE-formaten finns tillgängliga motsvaras dessa vanligtvis av datatyper enligt :

`float`	*Single precision* (32 bitar)
`double`	*Double precision* (64 bitar)
`long double`	*Extended precision* (80 bitar)

Observera dock att detta är implementationsberoende, dvs. det kan skilja mellan olika kompilatorer.

Implementering: heltal till flyttal

```
union float_long
{
    float f;
    long  l;
};
float ultof ( unsigned long a ) /* unsigned long till float */
{
          union float_long fl;
          int exp = 23 + 127;
          if ( a==0 )         {/* Specialfall, måste testas först */
                    return 0.0;
          }
          /* Normalisera */
          while (a & 0xFF000000) {
                    a = a >> 1;
                    exp = exp + 1;
          }
          while ( a < 0x00800000 ){
                    a  = a << 1;
                    exp = exp - 1;
          }
          a = a & ~0x00800000 ; /* nollställ implicit inledande etta */
          /* tilldelning till union medlem typ "long" */
          fl.l = (unsigned long) exp<<23 | a;
          return (fl.f);      /* returnera som flyttal, teckenbit är 0 */
}
```

Implementering: flyttal till heltal

```
long ftol (float a1)
{
        union     float_long fl;
        int               exp;
        char              sign=0;
        long              l;

        fl.f = a1;
        if (!fl.l) return (0);
        if( fl.l & 0x80000000 ) sign++;

        exp = (((unsigned long)( fl.l ) >> 23) & (unsigned int) 0x00FF) - 127 - 23;
        l = ((( fl.l ) & (unsigned long)0x007FFFFF) | 0x00800000 );

        if (exp > 8)    return LONG_MAX; /* största möjliga 'long int' */
        if (exp < -25)  return LONG_MIN; /* minsta möjliga 'long int' */

        /* exponenten ska vara noll... */
        if( exp > 0 ){ /* skifta mantissan till vänster */
                l = l << exp;
        }

        if (exp < 0 ) { /* skifta mantissan till höger */
                l = l >> -exp;
        }
        if( sign ) return -l;
        return l;
}
```


Addition/subtraktion av flyttal

1. Bestäm talens mantissor ur F (dvs. lägg till en etta framför den mest signifikanta biten i respektive F).
2. Bestäm talens exponenter på tvåkomplementsform.
3. Beräkna exponentskillnaden
4. Skifta mantissan för talet med *minst* exponent *höger* det antal gånger som exponentskillnaden anger (minns att exponenten anger binärpunkten i flyttalet).
5. Utför addition (subtraktion) av mantissorna efter tecken-överläggning.
6. Normalisera resultatet genom att skifta resultatmantissan samtidigt som resultatexponenten korrigeras.

Exempel: Additionen av 2,52 10^4 + 2,52 10^3, IEEE-single format flyttal

Omvandla talen till binärformat:

$A=(2,52)_{10}\ 10^4=(01000110100010011100000000000000)_{SFP}$

$B=(2,52)_{10}\ 10^3=(01000101000111011000000000000000)_{SFP}$

1. och 2. Dela upp talen i tecken, exponent och mantissa:

$A = (-1)^{S_A} \times M_A \times E_A$ och $B = (-1)^{S_B} \times M_B \times E_B$

$S_A = 0$

$M_A = (1.F)_A = (1.10001001110000000000000)$

$E_A = E'_A - 127 = (00001110)$ (dvs 141-127 = 14)

$S_B = 0$

$M_B = (1.F)_B = (1.00111011000000000000000)$

$E_B = E'_B - 127 = (00001011)$ (dvs 138-127 = 11)

3. Exponentskillnaden (14-11) är 3.
4. Skifta talet med minst exponent (M_B) tre steg höger

$M_B\ ' = 0.00100111011000000000000000$

observera att vi tillåter större upplösning hos mantissan under utförande av operationen.

Exempel: fortsättning

5. Utför additionen, båda talen positiva:

$$\begin{array}{ll} 1.10001001110000000000000 & M_A \\ +0.00100111011000000000000000 & M_B' \\ \hline =1.10110001001000000000000000 & \end{array}$$

6. Normalisera resultatet, i detta fall är resultatet redan i normaliserad form:

$S_R = 0$

$M_R = 1.10110001001000000000000000$

$E_R = 14$

Vilket nu ger oss representationen:

$F_R = (10110001001000000000000000)$

$E_R' = 127+14 = (10001101)_2$

Vi kan slutligen sätta samman resultatet:

$R = (01000110110110001001000000000000)_{SFP}$

Multiplikation/division av flyttal

En flyttalsmultiplikation/division utförs betydligt enklare än addition och subtraktion.

Vid flyttalsmultiplikation multipliceras mantissorna medan exponenterna adderas.

Vid flyttalsdivision divideras mantissorna medan dividendens exponent subtraheras från divisorns exponent, detta kan kortare skrivas som:

$A \times B = 2^{(E_A + E_B)} * M_A \times M_B$ respektive
$A / B = 2^{(E_A - E_B)} * M_A / M_B$

Implementering: multiplikation 1(3)

```
float mulf (float f1, float f2)
{
  union float_long fl1, fl2;
  unsigned long result;
  int exp;
  unsigned long sign;

  fl1.f = f1;
  fl2.f = f2;

  if (!fl1.l || !fl2.l)
      return ( 0.0 );
```

```
union float_long
{
    float f;
    long  l;
};
```


Implementering: multiplikation 2(3)

```
/* Bestäm tecken hos resultatet */
sign = (0x80000000 & fl1.l) ^ (0x80000000 & fl2.l);
/* Addition av exponenter, endast 'en förskjutning' i resultatet */
exp = (((unsigned long)(fl1.l >> 23)) & 0xFF) - 126;
exp = exp + (((unsigned long)( fl2.l >> 23)) & 0xFF);
/* Extrahera mantissor, maska in implicit bit */
fl1.l = (((fl1.l) & (unsigned long)0x007FFFFF) | 0x800000);
fl2.l = (((fl2.l) & (unsigned long)0x007FFFFF) | 0x800000);
/* Multiplicera mantissor */
result = fl1.l * fl2.l;
```


Implementering: multiplikation 3(3)

```
/* skifta 32 bitars resultat till 24 bitar och avrunda  (uppåt) */
  if (result & (unsigned long)0x80000000) {
    result += 0x80;
    result >>= 8;
    result &= ~0x800000; /* nollställ implicit inledande etta */
  }
  else { /* Vi "sparar" ytterligare en bits precision här… */
    result += 0x40;
    result >>= 7;
    exp--;
  }
/* packa ihop flyttalet ... */
  fl1.l = (unsigned long) exp<<23 | result | sign ;
  return (fl1.f);
}
```

Testoperationer på flyttal

En test av ett IEEE-flyttal kan ge följande resultat:

- normaliserat
- denormaliserat
- plus noll
- minus noll
- negativt
- plus oändligheten
- minus oändligheten
- plus *Not A Number*
- minus *Not A Number*

Detta kan jämföras med de testresultat vi kan få då ett vanligt heltal testas (*Zero* eller *Negative*).

En enhet för flyttalsaritmetik har därför ytterligare en uppsättning flaggbitar som är avsedda att återspegla de speciella resultat som fås vid en flyttalstest.

Jämförelseoperationer på flyttal

En flyttalsjämförelse ska, enligt standarden, kunna testa vilkoren:

- Equal To
- Greater Than
- Less Than
- Unordered

Tack vare kodningen blir dessa jämförelseoperationer enkla att implementera.

- *Equal To*, indikerar att operanderna är identiska
- *Greater Than/Less Than*, indikerar att operand A är större/mindre än operand B, detta inbegriper en teckenöverläggning och om talen har samma tecken kan resterande del av operanderna jämföras på samma sätt som vid heltalsjämförelse. Detta är en av fördelarna med att koda exponenten på *excess*-form i stället för tvåkomplementsform.
- *Unordered* innebär att minst ett av talen är *Not A Number*.