1 Aritmetik

32 bitars multiplikation/division med 16-bitars processor.

IEEE-754, flyttalsrepresentation

1.1 Multiplikation

Multiplikation av binära tal kan utföras på samma sätt som vid decimala tal. Allmänt skriver vi $P=X\times Y$ där:

- P är produkten av multiplikationen
- X är multiplikand
- Y är multiplikator

Det finns åtskilliga algoritmer för binär multiplikation. I huvudsak skiljs dom åt genom sin snabbhet. Vi kommer inte här att ge en fullständig framställning utan nöjer oss med att visa att varje multiplikation kan åstadkommas med de elementära operationerna *skift* och *addition*.

Reglerna för multiplikation i binära talsystemet är mycket enkla:

```
0 × 0 = 0
0 × 1 = 0
1 × 0 = 0
1 × 1 = 1
```

Vi tillämpar dessa regler och samma metod då vi ställer upp, och multiplicerar, som med decimala tal, på samma sätt som vi lärde i grundskolan. (I bland kallas detta "papper och penna-metoden").

Exempel 1.1 Multiplikation (P=X×Y), X=5, Y=6, "papper och penna-metoden", X>0, Y>0.

```
          Decimalt      Binärt

 X          5             0101       multiplikand
×Y        × 6           *0110       multiplikator
=P        = 30           0000
                        0101
                       0101
                    + 0000
                    =  0011110 = 2^4+2^3+2^2+2^1=30
```

Om vi speciellt betraktar uppställningen av binärmultiplikation i exemplet inser vi att samma resultat kan fås enbart genom att använda operationerna *addition* och *högerskift*. Detta är en direkt konsekvens av att vi arbetar i det binära talsystemet. I "papper-och-penna" metoden ser vi att vi antingen adderar noll, eller adderar multiplikanden, iterativt. För varje siffra vi avverkar flyttar vi oss en decimalpunkt till vänster, vilket också kan beskrivas som ett högerskift av multiplikanden. Vi provar därför nu en metod som endast använder de enkla operationerna addition och högerskift. Metoden bildar produkten genom en iterativ procedur som successivt genererar partialprodukter (PP). Då hela ordlängden är bearbetad har vi alltså den sista partialprodukten vilken också är den slutliga produkten. Algoritmen är:

Multiplikation P(produkt) = X(multiplikand) × Y (multiplikator)
Partialprodukt 0, PP(0) = 0
med början på multiplikatorns LSB (y_0)
för varje bit i hos multiplikatorn
Om y_i=1 addera multiplikand till nästa partialprodukt
annars addera 0 till nästa partialprodukt
skifta resultatet ett steg till höger
tills alla bitar inspekterats

Vi illustrerar algoritmen med följande exempel.

Exempel 1.2 Multiplikation (P=X×Y) tal utan tecken, X=6, Y=5, 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
```

Den beskrivna algoritmen fungerar definitivt för positiva tal. Det finns också all anledning att tro att algoritmen fungerar även om X är ett negativt tal (på tvåkomplementform) under förutsättning att högerskiftet görs med bibehållet tecken (aritmetiskt skift). Utan bevis, illustrerar vi det med följande exempel.

Exempel 1.3 Multiplikation (P=X×Y) tal med tecken, X=-6, Y=5,med addition/aritmetiskt skift, X<0, Y>0.

```
X=-6=         =1010 (multiplikand)
Y=5=y3y2y1y0  =0101 (multiplikator)

PP(0)           0000        y0=1⇒ADD X
              + 1010
                1010        skifta aritmetiskt
PP(1)           11010       y1=0⇒ADD 0
              + 0000
                11010       skifta aritmetiskt
PP(2)           111010      y2=1⇒ADD X
              + 1010
                1100010     skifta aritmetiskt
PP(3)           1100010     y3=0⇒ADD 0
              + 0000
                1100010     skifta aritmetiskt

P=PP(4)=      11100010 = -(00011110)2 = -(30)10
```

Vid additionen av PP(2) till X ser vi att vi får spill från den mest signifikanta positionen. Med de givna förutsättningarna (X<0) sker detta endast då vi har en partialprodukt som är mindre än 0. Spillet kan alltså ses som en kopia av teckenbiten. Eftersom vi skiftar aritmetiskt återställs denna på korrekt sätt.

För att slutligen generalisera och tillåta såväl Y<0 som X<0 måste vi göra en modifikation av algoritmen. Bakgrunden är enkel, vid tvåkomplementrepresentation utgör den mest signifikanta biten ingen egentlig värdesiffra utan är enbart teckenrepresentation. Då vi bildar den sista partialprodukten, dvs. slutprodukten, markerar talet *Y*'s teckenbit att -*X* ska adderas (i stället för *X*).

Vi illustrerar med ytterligare ett exempel, metoden kallas också *Robertson's metod*:

Exempel 1.4 Multiplikation (P=X×Y) tal med tecken, X=-6, Y=-5, 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
```

Produkten av två godtyckliga tvåkomplementstal med *i* och *j* bitar kan maximalt vara (*i*+*j*-1). Detta inses av att *i* bitar ska skiftas *j* gånger (totalt *i*+*j* bitar), men samtidigt krävs endast en teckenbit för produkten (dvs *i*+*j*-1). Vanligtvis gäller att ordlängden hos multiplikator och multiplikand är densamma, *i*=*j*=*n*, då krävs maximalt 2*n*-1 bitar för att representera produkten av två *n*-bitars tal.

Vi har nu sett algoritmer som hanterar multiplikation av såväl tal på binär form som tal på tvåkomplementsform. Vi har sett hur multiplikationen kan utföras med hjälp av de enkla operationerna addition och skift. Vi har också sett att multiplikation av binära tal kräver sin algoritm (med logiskt skift), medan multiplikation av tvåkomplementstal kräver en (mycket liten) modifikation av algoritmen, användning av aritmetiskt skift. En dator kan självfallet inte avgöra om bitsträngarna ska tolkas som tal med, eller utan tecken. Processorer som tillhandahåller instruktioner för multiplikation har därför alltid minst två varianter av denna: en för multiplikation av tal utan tecken (*unsigned multiplication*) och en annan instruktion för multiplikation av tal med tecken (*signed multiplication*).

1.1.1 32-bitars multiplikation som biblioteksfunktion

För processorer som inte tillhandahåller maskininstruktioner för multiplikation kan dessa implementeras som "biblioteksfunktion". Följande implementering bygger direkt på "papper-och-penna"-metoden. Den fungerar för både tal med tecken och tal utan tecken. Observera att bara de 32 minst signifikanta bitarna av resultatet behålls och att eventuellt "spill" inte detekteras.

```
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;
}
```

För processorer som har instruktioner för multiplikation (8 och/eller 16 bitar) kan man oftast åstadkomma bättre implementeringar genom att använda dessa.

Exempel 1.5 Visa hur 32-bitars multiplikation av tal utan tecken kan utföras med hjälp av 16-bitars multiplikation, addition och skiftoperationer.

Antag a, b 32-bitars tal, skriv:

$a = ah \times 2^{16} + al$, där ah är de 16 mest signifikanta bitarna och al de 16 minst signifikanta bitarna.

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

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:

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

dvs. eftersom ah, bh, al och bl är 16 bitars tal kan 32 bitars multiplikation genomföras med 16-bitars multiplikation, addition och skiftoperationer.

Första termen försvinner eftersom produkten (ah × bh) ska skiftas 32 steg till vänster och detta inte ryms inom resultatets 32 bitar, vi kan därför skriva:

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

Resultatet från Exempel 1.5 kan enkelt implementeras enligt följande:

```
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;
}
```

Vi kan slutligen implementera multiplikation av 32-bitars tal med tecken genom att använda ovanstående funktion tillsammans med teckenöverläggning, enligt följande:

```
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;
}
```

1.2 Division

Precis som för multiplikation, kan binär division utföras enligt "papper och penna-metod". I detta avsnitt ska vi illustrera hur detta går till. Metoden kan endast användas på positiva heltal. Vi behandlar inte metoder för division av tvåkomplementtal i det generella fallet även om sådana metoder finns.

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: $R = X - QY$

Av Exempel 1.6 nedan ser vi hur partialresterna bildas genom att succesivt subtrahera (största möjliga) kvotsiffra multiplicerad med Y som fortfarande ger en *positiv* partialrest. Kvotsiffran fås genom *prövning*.

Exempel 1.6: Decimal division, återställning av resten, 3967/15

```
X = 3967
Y = 15
                R  = X  - Q×Y
     0264,4
15  |3967,0   3697=3697-0×15                                  Utgångsläge
    -0
     3967,0   3967=3967-(0×10^3)×15                           steg 1
    -30
      967,0    967=3697-(0×10^3+2*10^2)×15                   steg 2
    - 90
       67,0     67=3697-(0×10^3+2×10^2+6×10^1)×15            steg 3
    -  60,0
        7,0      7=3697-(0×10^3+2×10^2+6×10^1+4×10^0)×15     steg 4
    -   6,0
        1,0      1=3697-(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
```

Låt oss nu studera ett exempel på binär division med användning av samma metod. Som vi ska se blir detta faktiskt enklare eftersom endast två kvotsiffror (0 och 1) kan förekomma. Prövningen av Q*Y för positiv partialrest kräver alltså inte någon egentlig multiplikation.

Exempel 1.7: Binär division X/Y, med återställning, X=13, Y=5

```
X = 1101
Y = 0101
                     R  = X  - Q×Y
       0010
0101  |1101       1101=1101- 0×0101                          Utgångsläge
 -0000
      1101       1101=1101-(0×2^3)×0101                     steg 1
  -0000
      1101       1101=1101-(0×2^3+0×2^2)×0101               steg 2
   -0101
       0011        11=1101-(0×2^3+0×2^2+1×2^1)×0101         steg 3
    -0000
       0011        11=1101-(0×2^3+0×2^2+1×2^1+0×2^0)*0101   steg 4

DVS.  1101/0101 = 0010 + 0011/0101 (2 + 3/5)
```

Den visade metoden kan formuleras i en algoritm där vi endast använder operationerna vänsterskift och addition. Efter subtraktion av *Y* från aktuell partialrest sätts kvotsiffran till 1 om skillnaden blir positiv, om skillnaden blir negativ sätts kvotsiffran till 0 och partialresten *återställs* genom addition av *Y*, därefter fortsätter man med nästa steg. Om partialresten blir 0 har divisionen "gått jämnt ut" och operationen är klar. Ofta är detta inte fallet och man kan då fortsätta om man är villig att acceptera decimalsiffror i svaret (jämför med föregående exempel).

Algoritm: ***Division med återställning***

$R = X - Q*Y$

$Q=q_0q_1q_2...q_{n-1}$

n=antal kvotbitar att beräkna

$R_0 = X$

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

för i = 2..n

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

Anmärkning: Antalet kvotbitar vi önskar bestämmer antalet partialrester som ska beräknas och därmed också det antal vänsterskift som måste utföras. Eftersom partialrest *n* inte ska skiftas får vi *n-1* vänsterskift.

Exempel 1.8: Utför binär division av 13/5 med återställningsmetod.

Svaret ska anges med 4 kvotbitar och 4 restbitar. Varje steg i algoritmen ska redovisas.

Lösning:

X=13 = 1101
Y=5 = 0101
-Y=-5= 1011
n=4 (3 skift ska utföras)

Utgångsuppställningen:

$R_0 = X$ 0001**101** ← 3 bitar ska skiftas
+(−Y) 1011

steg 1:

R_0−Y **1**100101 <0⇒**q_0=0**⇒Återställ
+(Y) 0101
R_1 0001101
2* R_1 0011010

steg 2:

+(−Y) 1011
2*R_1−Y **1**110010 <0⇒**q_1=0**⇒Återställ
+(Y) 0101
R_2 0011010
2* R_2 0110100

steg 3:

+(−Y) 1011
2*R_2−Y **0**001100 ≥0⇒**q_2=1**
R_3 0001100
2* R_3 0011000

steg 4:

+(−Y) 1011
2*R_3−Y 1110000 <0⇒**q_3=0**⇒Återställ
+(Y) 0101
R_4 0011000

n=4 och algoritmen terminerar här

Q= $q_0q_1q_2q_3$ =%0010=2
R= =%0011=3
DVS: 13/5 = 2 + 3/5

1.1.2 32-bitars division som biblioteksfunktion

För processorer som saknar instruktioner för 32-bitars division visas här en implementering av ”divu32”, dvs. heltalsdivision av två 32-bitars tal utan tecken. Observera hur vi ”återanvänder” lagringsplatsen för parametern `a`, genom att succesivt skifta in eventuella kvotbitar till `a`.

```
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;
}
```

Genom att införa teckenöverläggning kan vi också hantera division av 32-bitars tal med tecken:

```
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;
}
```

Avslutningsvis visar vi en enkel implementering av modulusdivision, där vi helt enkelt använder definitionen av operationen:

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

1.3 Flyttal

Vi har tidigare uteslutande behandlat kodning av heltal. I många sammanhang är detta inte tillräckligt exempelvis då vi vill uttrycka mycket små tal så som avstånd mellan atomer eller vi vill uttrycka mycket stora tal så som avstånd mellan galaxer i universum.

En bitsträng kan förses med en ”tänkt” binärpunkt och tolkningen av talvärdet göras därefter. Vi kallar detta allmänt för *fixtal* och menar då en fast binärpunkt någonstans i bitsträngen.

Exempel 1.9 Fixtal

Bitsträngen 11011100 kan förses med en ”tänkt” binärpunkt enligt 1101.1100, tolkningen av talet blir då:

$$2^3+2^2+2^0+2^{-1}+2^{-2} = 8+4+1+0,5+0,25 = 13,75$$

Fixtalsaritmetik kan vara användbart i många sammanhang men det löser dock knappast det grundläggande problemet dvs. att, med samma representation, kunna ange såväl mycket små som mycket stora tal. För att komma till rätta med detta kan vi införa en *flytande* binärpunkt, dvs. information om var, i talet, binärpunkten är placerad finns inkodat i talet.

Exempel 1.10 Mantissa och exponent

Talet 132 kan skrivas om som en produkt av två tal enligt:

$$132\times10^0 = 13,2\times10^1 = 1,32\times10^2 = 0,32\times10^3 \text{ osv..}$$

vi säger att vi delat upp talet i *mantissa* och *exponent*. Under förutsättning att exponentens och mantissans bas är den samma så gäller att exponenten anger decimalkommats placering i talet.

1.1.3 Normaliserad form

Om det gäller att mantissan är i intervallet $1 \leq M < \beta$, där M är mantissan och β är talbasen, sägs den vara *normaliserad*. Av detta följer alltså att den normaliserade mantissans talvärde alltid är mindre än talbasen och samtidigt, 1 som minst.

Som Exempel 1.10 antyder finns det ett oändligt antal former att ange ett tal på formen *mantissa*/*exponent*, det finns dock bara *en* normaliserad form.

Exempel 1.11 Uttryck talet 132 på normaliserad form

Den normaliserade formen av talet 132 är

$$1,32\times10^2$$

ty endast denna form uppfyller:

$$1 \leq 1,32 < 10$$

Ovanstående resonemang kan naturligtvis tillämpas oberoende av talsystem.

Exempel 1.12

Uttryck talen
a) $(1101.011)_2$
b) $(1E.0A)_{16}$

på normaliserad form.

***Lösning*:**

Skriv talen som mantissa och exponent:
a) $(1101.011)_2 = (1101.011)_2 \times 2^0$
b) $(1E.0A)_{16} = (1E.0A)_{16} \times 16^0$

”Flytta” binärpunkten så att mantissan uppfyller villkoret för normaliserad form. För varje steg vi flyttar binärpunkten till vänster adderar vi 1 till exponenten.
a) $(1101.011)_2 = (1.101011)_2 \times 2^3$
b) $(1E.0A)_{16} = (1.E0A)_{16} \times 16^1$

Exempel 1.13: Omvandling av normaliserad form med olika radix

Skriv talet $(2{,}52)_{10}\ 10^4$ på normaliserad form $(M)_2 \times 2^E$

Lösning:

Omvandla först hela talet till binär form på känt sätt:

$2{,}52\ 10^4 = 25200 = (1100\ 0100\ 1110\ 000)_2 \times 2^0$

normalisera talet:

$= (1.100\ 0100\ 1110\ 000)_2 \times 2^{14}$

1.1.4 Representation av flyttal

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. Detta innebär att vi, då vi lagrar flyttal, kan utelämna denna siffra och på så vis åstadkomma ett kompaktare format. Detta utnyttjas speciellt i standardiserade flyttalsformat.

1.4 IEEE754 - flyttalsstandard

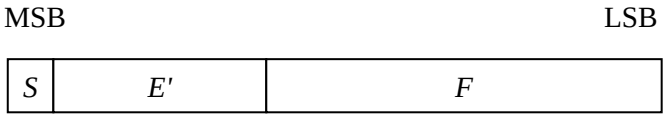
IEEE-flyttalsstandard specificerar

- 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

Ett IEEE-flyttalsformat delas upp i tre fält enligt följande:



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

Exempel 1.14 Omvandling av heltal till flyttal

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

Lösning:

Vi har tidigare kommit fram till resultatet:

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

Mantissan ska ha totalt 24 bitar, vi ”fyller på” med nollor på slutet...

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

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

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

Exponenten i IEEE-formen 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 sammanställer nu resultatet i 32-bitars form (*SFP*) och får slutligen:

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

För praktiska ändamål är det lämpligt att formulera algoritmer för omvandlingar mellan vanliga datatyper.

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

Anm: Algoritmen termineras med sfp ←

1. *sfp*(*S*)=0; $x' \leftarrow x$;
2: Bestäm signifikand
   - Bitar x_{32} .. x_{24}, får inte plats i mantissan, dividera därför successivt x' med 2 och justera samtidigt 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 (dess position upptas av exponenten) 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

Anm: Algoritmen termineras då *x* ←

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) | b23
```

4. Skifta mantissan tills exponenten blir 0

```
if ( exp > 0 )
      x  ← (mant  << exp );
if ( exp < 0 )
      x  ← (mant  >> exp );
```

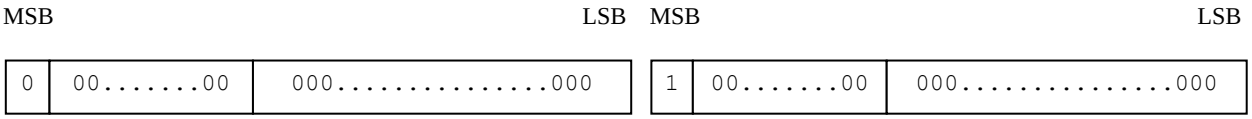
5. Returnera heltal med rätt tecken

```
if( sign )
      x  ← -(mant);
else
      x  ← mant;
```

1.1.5 Speciella kodningar av IEEE flyttalsformat

Flyttalet noll

Det finns inget sätt att ange talet noll på normaliserad form. Standarden säger i stället att såväl exponent som mantissa här ska vara noll. Observera att detta ger frihet att koda såväl plus som minus noll beroende på hur flyttalets teckenbit sätts. Flyttalet noll kan följaktligen kodas på något av följande sätt:



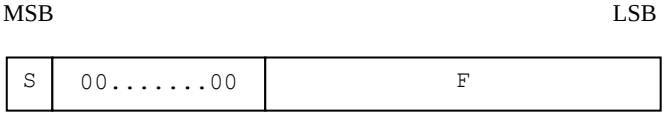
Avrundningsmetoder

Operationen kan utföras med större precision än formatet tillåter. Som konsekvens av detta kan vi i bland tvingas avrunda resultatet. Standarden tillåter fyra olika avrundningsmetoder:

- *Round to nearest*, resultatet avrundas till det representerbara värdet som ligger närmst det verkliga värdet. Om det verkliga värdet ligger mitt emellan två representerbara värden avrundas resultatet till ett jämnt tal.
- *Round to zero*, resultatet trunkeras till rätt precision, dvs. avrundningsbitarna ignoreras.
- *Round towards minus infinity*, resultatet avrundas nedåt till närmsta representerbara värde.
- *Round towards plus infinity*, resultatet avrundas uppåt till närmsta representerbara värde.

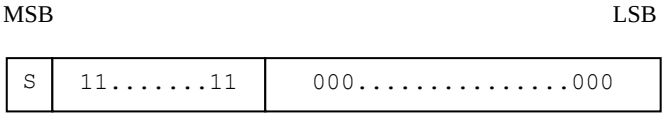
Denormaliserade tal

Denormaliserade tal uppträder som resultat av operationer där exponenten vid normalisering av resultatet inte kan minskas längre, dvs. man uppnått den (till beloppet) största negativa exponent som kan representeras. Detta innebär att mantissan inte kan normaliseras. I stället för att avrunda ett sådant resultat tillåter här standarden ett icke-normaliserat resultat. Det denormaliserade talet indikeras genom att exponenten sätts till noll men mantissan är skild från noll. Ett denormaliserat tal kodas följaktligen:



Oändligheter

Då resultatet av en flyttalsoperation överskrider det representerbara talområdet, dvs. exponenten anger ett tal som inte kan normaliseras indikeras detta som *infinity*. *Infinity* kan, precis som talet noll, ha tecken och det indikeras genom att samtliga exponentens bitar sätts till ett och mantissan sätts till noll.



Icke-tolkningsbara resultat

IEEE-standarden inbegriper också en klass av resultat kallade "NaN", (Not A Number). Detta utgör resultatet av en operation som inte har någon matematisk tolkning, exempelvis "oändlighet dividerat med oändlighet" eller "division med noll". Samtliga exponentens bitar sätts här till ett och mantissan är skild från noll. För en operation där minst en av operanderna är *NaN* produceras alltid resultatet *NaN*.

1.1.6 Typomvandling som biblioteksrutin

I programspråket 'C' förekommer flyttalstyper med olika precision, på samma sätt som man skiljer på `short` respektive `long int` bland heltalstyperna.

Exempel 1.15 Olika flyttalstyper i 'C'

Då alla tre IEEE-formaten finns tillgängliga motsvaras dessa vanligtvis (dock inte alltid) av datatyper enligt följande:

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

Det kan dock skilja mellan olika kompilatorer. För exempelvis XCC12 gäller att alla flyttalstyper implementeras som IEEE *single precision*.

Vissa omvandlingar till och från IEEE flyttalsformat är vanliga och förtjänar speciell uppmärksamhet. Vi behandlar därför nu hur omvandlingar kan ske med hjälp av programrutiner, kodade i programspråket C.

Exempel 1.16 Typomvandling

Betrakta följande C-konstruktion:

```
unsigned long ui;
float f;

f = (float) ui;    /* Heltal till flyttal */
```

Detta är ett exempel på en *typomvandling*, dvs. heltalet i variabeln `ui`, som är ett 32-bitars tal utan tecken måste typkonverteras till motsvarande representation för flyttalet i variabeln `f`, IEEE *single precision*. Vi vet sedan tidigare att bitmönstren, för ett och samma talvärde, skiljer sig åt beroende på tolkningen, dvs. *datatypen*. Här måste alltså någon typ av omvandling av bitsträngen för `ui` göras.

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

Exempel 1.17 Maskininstruktion för typomvandling

Power-PC arkitekturen definierar maskininstruktioner för typomvandlingar mellan heltal och flyttal, exempelvis:

```
FCTIW fr0,fr1      ; "floating convert to integer word "
FCFID fr0,fr1      ; "floating convert from integer doubleword "
```

Instruktionerna utför alltså typomvandlingar som en maskininstruktion.

Mindre, billigare processorer saknar vanligtvis speciell enhet för hantering av flyttal. Vi måste då tillhandahålla funktioner för omvandlingarna. Låt oss betrakta typerna (`unsigned`) `long` och `float`, vi identifierar då följande typomvandlingar:

float	→ **unsigned long int**	(ftoul, "float to unsigned long")
float	→ **signed long int**	(ftosl, "float to signed long")
unsigned long int	→ **float**	(ultof, "unsigned long to float")
signed long int	→ **float**	(sltof, "signed long to float")

Reglerna för typkonverteringar i 'C' gör att konverteringarna från **float** till **unsigned** respektive **signed long** i själva verket är samma sak. Ett flyttal som är mindre än noll kan inte representeras korrekt med en **unsigned** typ, så i stället konverteras det till en **signed** typ. Slutsatsen blir att vi klarar oss med en konverteringsrutin från **float**, vi kallar den `ftol` ("float to long")

För konverteringar *till* **float** gäller att det räcker med en omvandlingsrutin för tal utan tecken, *unsigned long to float*. Vi kan då implementera omvandlingen av tal med tecken, *signed long to float*, med hjälp av teckenöverläggning. Av resonemanget framgår att vi i själva verket behöver implementera två konverteringsrutiner:

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

Vi har tidigare gett algoritmer för dessa omvandlingar och fortsätter här med dess implementering i 'C'.

Vi börjar med typkonverteringen *unsigned long to float*. För att ha friheten att tolka ett bitmönster på olika sätt använder vi "union"-konstruktionen i C. Följande deklaration:

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

ger oss en typ, `float_long`, som upptar 32 bitar i minnet. En variabel deklarerad med denna typ kan nu användas för att representera en och samma bitsträng som två olika typer, samtidigt. Detta använder vi i följande konverteringsrutin `ultof`. Observera specialfallet om talet noll ska omvandlas.

```
float ultof ( unsigned long a ) /* unsigned long to 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 */
}
```

Signed long to float implementerar vi nu enkelt på följande sätt:

```
float sltof ( signed long a ) /* signed long to float */
{
      if ( a < 0)
            return  -( ultof ( -a ) );
      else
            return ultof ( a );
}
```

Slutligen visar vi en omvandlingsrutin för *float to signed long*:

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

	fl.f = a1;

	if (!fl.l)
		return (0);
	sign = 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;

}
```

1.1.7 Aritmetiska operationer på flyttal

Aritmetiska operationer på flyttal blir avsevärt mer komplicerade än operationer på heltal. Operationerna måste utföras i flera steg.

Flyttalsaddition/subtraktion

Vid addition eller subtraktion av flyttal utförs ett flertal operationer där mantissa och exponent måste behandlas var för sig, dessutom tillkommer teckenöverläggning eftersom mantissan ju är given på tecken-beloppsform. Vid operation på mantissorna måste vi först se till att båda talen har samma exponent. Addition/subtraktion av IEEE-flyttal utförs i följande steg:

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 1.18 Addition av flyttal

Visa additionen av $2{,}52\ 10^4 + 2{,}52\ 10^3$ av flyttal givna enligt IEEE-single format form:

Lösning:

Omvandla talen till binärformat:

$A=(2,52)_{10}\ 10^4=(0\underline{10001101}10001001110000000000000)_{SFP}$

$B=(2,52)_{10}\ 10^3=(0\underline{10001010}00111011000000000000000)_{SFP}$

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

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

$S_A = 0$
$M_A = (1.F)_A = (1.00010011100000000000000)$
$E_A = E'_A - 127 = (00001110)$ (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.

5. Utför additionen, båda talen positiva:

```
 1.00010011100000000000000    M_A
+0.00100111011000000000000000 M_B´
=1.01100010010000000000000000
```

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

$S_R = 0$
$M_R = 1.01100010010000000000000000$
$E_R = 14$

Vilket nu ger oss representationen:

$F_R = (01100010010000000000000000)$
$E_R' = 127+14 = (10001101)_2$

Vi kan slutligen sätta samman resultatet:

$R = (01000110110110001001000000000000)_{SFP}$

Addition och subtraktion som biblioteksrutiner

Vi ska nu visa hur addition/subtraktion kan implementeras av programrutiner som enbart konstruerats av operationer på heltal. Vi visar först implementeringen av addition, därefter hur vi med hjälp av teckenöverläggning enkelt kan implementera även operationen ”subtraktion”. I denna implementering utför vi addition av mantissorna med största tillgängliga precision. Vi har 32 bitar tillgängliga, en bit representerar tecken och ytterligare en bit krävs för att detektera spill från additionen, dvs. 30 bitar. Detta ger oss bättre noggrannhet men ger också extra normaliseringar, dvs. något mer komplex kod.

```
float addf (float f1, float f2)
{
      long mant1, mant2;
      union float_long fl1, fl2;
      int exp1, exp2;
      long sign = 0;

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

      /* Kontroll att ingen av parametrarna är 0 */
      if (!fl1.l)
            return (fl2.f);
      if (!fl2.l)
            return (fl1.f);

      /* Extraktion av exponenter */
      exp1 = (((unsigned long)(fl1.l) >> 23) & (unsigned int) 0x00FF);
      exp2 = (((unsigned long)(fl2.l) >> 23) & (unsigned int) 0x00FF);

      /* Kontroll att addition är meningsfylld */
      if (exp1 > exp2 + 25)
            return (fl1.f);
      if (exp2 > exp1 + 25)
            return (fl2.f);

      /* Extraktion av mantissa,
            vi använder 24+6 dvs 30 bitar och avrundar efter addition */
      mant1 = (((fl1.l) & (unsigned long) 0x007FFFFF ) | 0x800000 )<<6;
      mant2 = (((fl2.l) & (unsigned long) 0x007FFFFF ) | 0x800000 )<<6;

      /* Teckenöverläggning */
      if ( fl1.l & 0x80000000 )
            mant1 = -mant1;
      if ( fl2.l & 0x80000000 )
            mant2 = -mant2;

      if (exp1 > exp2)  /* Skifta mantissa med lägst exponent */
      {
            mant2 = (mant2 >> (exp1 - exp2) );
      }
      else
      {
            mant1 = (mant1 >> (exp2 - exp1) );
            exp1 = exp2;      /* Vi använder 'exp1' nedan */
      }
      mant1 += mant2;   /* Addera mantissor */

      /* Vi har nu icke-normaliserat resultat i 'exp1/mant1' */
      if (  mant1 == 0 )
            return (0.0);

      if (mant1 < 0)
      {     /* Justera mantissa, resultatets teckenbit sätts till 1 */
            mant1 = -mant1;
```

```
            sign = 0x80000000;
      }

      /* Normalisera upp till 30 bitars mantissa om det behövs */
      while (!(mant1 & (unsigned long) 0xE0000000))
      {
            mant1 <<= 1;
            exp1--;
      }

      /* Normalisera ned till 30 bitars mantissa om det behövs */
      if (mant1 & (unsigned long) 0x40000000)
      {
            mant1 >>= 1 ;
            exp1++;
      }

      /* Avrundning, lsb av mantissa ”round to nearest even” */
      mant1 += (mant1 & (unsigned long)0x40) ?
                  (unsigned long) 0x20 : (unsigned long) 0x1F;

      /* Normalisera ned till 30 bitars mantissa om det behövs */
      if (mant1 & (unsigned long) 0x40000000)
      {
            mant1 >>= 1;
            exp1++;
      }

      /* ”Kasta” nu de 6 extra bitarna vi använt vid addition av mantissor */
      mant1 = mant1 >> 6;

      /* nollställ implicit inledande etta */
      mant1 = mant1 & ~0x00800000;

      /* packa ihop flyttalet ... */
      fl1.l = (unsigned long) exp1<<23 | mant1 | sign ;
      return (fl1.f);
}
```

Implementeringen av ”subf” blir nu enkel:

```
float subf (float f1, float f2)
{
      union float_long fl1, fl2;
      fl1.f = f1;
      fl2.f = f2;
      /* toggla teckenbiten hos operand 2 och utför addition */
      fl2.l ^= 0x80000000;
      return ( addf( fl1.f , fl2.f ) );
}
```

Flyttalsmultiplikation och Division

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:

$A*B = 2^{(E_A+E_B)} * F_A * F_B$ respektive

$A/B = 2^{(E_A-E_B)} * F_A/F_B$

Multiplikation och division som biblioteksrutiner

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

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

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

      /* 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)) & (unsigned int) 0x00FF) - 126;
      exp = exp + (((unsigned long)( fl2.l >> 23)) & (unsigned int) 0x00FF);
      /* 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;

      /* skifta 32 bitars resultat till 24 bitar och avrunda */
      if (result & (unsigned long)0x80000000)
      {
            result += 0x80;
            result >>= 8;
      }
      else
      {
            result += 0x40;
            result >>= 7;
            exp--;
      }

      result &= ~0x800000; /* nollställ implicit inledande etta */

      /* packa ihop flyttalet ... */
      fl1.l = (unsigned long) exp<<23 | result | sign ;
      return (fl1.f);
}
```

```
float divf (float f1, float f2)
{
      union float_long fl1, fl2;
      long result;
      unsigned long mask;
      long mant1, mant2;
      int exp ;
      unsigned long sign;

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

      if (!fl2.l) {     /* division med 0 ? */
            return (0xFFFFFFFF ); /* returnera 'NaN' */
      }
      if (!fl1.l)       /* dividend 0 ? */
            return (0.0);

      /* Bestäm tecken hos resultatet */
      sign = (0x80000000 & fl1.l) ^ (0x80000000 & fl2.l);

      /* Subtraktion av exponenter, endast 'en förskjutning' i resultatet */
      exp = (((unsigned long)(fl1.l >> 23)) & (unsigned int) 0x00FF);
      exp = exp - (((unsigned long)( fl2.l >> 23)) & (unsigned int) 0x00FF);
      exp = exp + 126;

      /* Extrahera mantissor, maska in implicit bit */
      fl1.l = (((fl1.l) & (unsigned long)0x007FFFFF) | 0x800000);
      fl2.l = (((fl2.l) & (unsigned long)0x007FFFFF) | 0x800000);

      /* dividend > divisor ger 25 signifikanta resultatsiffror */
      if (mant1 < mant2)
      {
            mant1 = mant1 << 1;
            exp--;
      }

      /* Division av mantissor med upprepad subtraktion */
      mask = 0x1000000;
      result = 0;
      while (mask)
      {
            if (mant1 >= mant2)
            {
                  result = result | mask;
                  mant1  = mant1 - mant2;
            }
            mant1 = mant1 << 1;
            mask  = mask >> 1;
      }

      /* avrunda uppåt */
      result = result + 1;

      /* normalisera (högerskift, ty divisor > dividend */
      exp++;
      result = result >> 1; /* Nu 24 sign. bitar i mantissa */

      result &= ~0x800000; /* nollställ implicit inledande etta */

      /* packa ihop flyttalet ... */
      fl1.l = (unsigned long) exp<<23 | result | sign ;
      return (fl1.f);
}
```

Flyttalstest och jämförelser

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

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".

Sammanfattning IEEE-flyttalsstandard

Låt oss som avslutning bestämma talområden och upplösning hos de olika IEEE-formaten

Single format, (32 bitar) har 23 bitars signifikand, 8 bitars exponent och 1 teckenbit. Det till beloppet minsta normaliserade tal, skilt från noll, som då kan representeras är: E'=1 och F = 0 vilket ger

E = -126
M = 1.00.....0

dvs: $1.0 \times 2^{-126} \cong 1{,}2 \times 10^{-38}$

Det minsta *denormaliserade* tal som kan representeras får vi då E'=0 och endast den minst signifikanta biten i F är 1enligt:

E = -126 och
M= 00000000000000000000001

dvs: $2^{-126} \times 2^{-23} \cong 1{,}4 \times 10^{-45}$

Det största normaliserade tal som kan representeras får vi då E'=254 och F = 11111111111111111111:

E = 127
M = 1. 11111111111111111111

vilket ger: $1.11111111111111111111 \times 2^{127} \cong 2 \times 2^{127} \cong 3{,}4 \times 10^{38}$

Upplösningen ges av värdet hos den minst signifikanta biten i F ty detta värde anger den minsta skillnad mellan två flyttal vi kan representera. Vikten hos denna bit ger oss alltså upplösningen enligt:

$2^{-23} \cong 1{,}19\ 10^{-7} = 0{,}000000119$

Mot bakgrund av att all aritmetik utförs med högre precision kan man utgå från att den första åtskiljande siffran alltid avrundas till ett korrekt värde. Vi ser då att vi kan utgå från att vi har minst 7 decimala siffror's noggrannhet.