Maskinorienterad Programmering 2012/2013

Kodningskonventioner och programbibliotek

Ur innehållet:

- Kodningskonventioner
  - maskinnära programmering i C och assemblerspråk
- Programbibliotek
  - 32-bitars operationer med 16 bitars processor

Kodningskonventioner (XCC12)

- Parametrar överförs till en funktion via stacken.
- Då parametrarna placeras på stacken bearbetas parameterlistan från höger till vänster.
- Utrymme för lokala variabler allokeras på stacken. Variablerna behandlas i den ordning de påträffas i koden.
- Prolog kallas den kod som reserverar utrymme för lokala variabler.
- Epilog kallas den kod som återställer (återlämnar) utrymme för lokala variabler.
- Den del av stacken som används för parametrar *och* lokala variabler kallas *aktiveringspost*.

Beroende på datatyp används för returparameter HC12's register enligt följande tabell:

Storlek	Benämning	C-typ	Register
8 bitar	byte	char	B
16 bitar	word	short int och pekartyp	D
32 bitar	long	long int	Y/D

Övningsexempel:

Skriv en egen version av standardfunktionen `strncpy` med användande av *pekare* (indexering är inte tillåten…)

1. Använd XCC12, kompilera till assemblerkod.
2. Utgå från resultatet och förbättra assemblerkoden så långt du kan.

Specifikation av **`strncpy`**:

Synopsis

```
#include <string.h>
void strncpy(char * s1, const char * s2, int n);
```

Description

The `strncpy` function copies the string pointed to by `s2` (including the terminating null character) into the array pointed to by `s1`. At most `n` characters (including null) are copied. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The `strncpy` function returns nothing.

Lösning (rättfram implementering):

```
void  strncpy(char *s1, char *s2, int n)
{
      while (*s2 != 0) {
            *s1++ = *s2++;
            if (--n <= 0)
                  break;
      }
      *s1 = 0;
}
```



```
;         void strncpy (char *s1, const char *s2, int n)
_strncpy:
;     4 | {
;     5 |           while (*s2 != 0) {
_2:
          TST       [4,SP]
          BEQ       _3
;     6 |                       *s1++ = *s2++;
          LDX       2,SP
          LDY       4,SP
          LDAB      1,Y+
          STAB      1,X+
          STX       2,SP
          STY       4,SP
;     7 |                       if (--n == 0)
          LDY       6,SP
          DEY
          STY       6,SP
          TFR       Y,D
          CPD       #0
          BEQ       _3
;     8 |                                   break;
_4:
          BRA       _2
_3:
;     9 |           }
;    10 |           *s1 = 0;
          CLR       [2,SP]
;    11 | }
          RTS
```

Bestäm kandidater för registerallokering:
Parametrar och lokala variabler:

`4,SP`: (s2) typen 'pekare' 3 st referenser
`2,SP`: (s1) typen 'pekare' 3 st referenser
`6,SP`: (n) typen 'int' 2 st referenser

Register X och Y är självskrivna pekare

Register D kan hålla en 'int'
Men då kan inte B användas för datakopieringen…

Lösning:
Allokera X till s1
Allokera Y till s2
Allokera D till n
Använd MOV för datakopiering…

```
;         void strncpy (char *s1, const char *s2, int n)
_strncpy:
;     4 | {
          LDD       6,SP
          LDY       4,SP
          LDX       2,SP
;     5 |           while (*s2 != 0) {
_2:
          TST       ,Y
          BEQ       _3
;     6 |                     *s1++ = *s2++;
          MOVB      1,Y+,1,X+
;     7 |                     if (--n == 0)
          SUBD      #1
          BGT       _2
;     8 |                               break;
_3:
;     9 |           }
;    10 |           *s1 = 0;
          CLR       ,X
;    11 | }
          RTS
```

Med lämplig registerallokering kan koden förbättras betydligt…

```
n -> D
s2 -> Y
s1 -> X
```


Exempel (ES 2.26):

Följande C-deklarationer har gjorts på ”toppnivå” (global synlighet):

```
char    a,b,c;
char    min( char a, char b );
```

a) Visa hur variabeldeklarationerna översätts till assemblerdirektiv för HCS12.

b) Visa hur följande sats översätts till assemblerkod för HCS12:

```
c = min( a , b );
```

Vi löser på tavlan...

Exempel (ES 2.32):

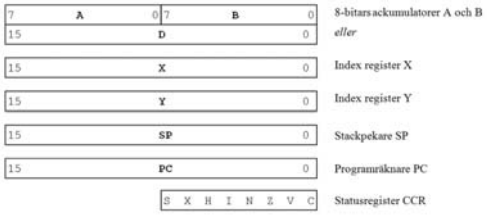
Inledningen (parameterlistan och lokala variabler) för en funktion ser ut på följande sätt:

```
void  function( long c, char b, unsigned int a )
{
      char  d;
      long  e;
.....
```

a) Visa hur utrymme för lokala variabler reserveras i funktionen (*prolog*).
b) Visa funktionens *aktiveringspost*, ange speciellt offseter för parametrar och lokala variabler.
c) Visa hur tilldelningen `d = b;` utförs i assemblerspråk
d) Visa hur tilldelningen `e = a;` utförs i assemblerspråk

Vi löser på tavlan...

Programbibliotek, HCS12 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.
Operationer implementeras i programbibliotek.

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
```

Pseudo språk:

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

32 bitar

```
        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  (multiplikator)
Y=5=y3y2y1y0    =0101  (multiplikand)

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

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)...

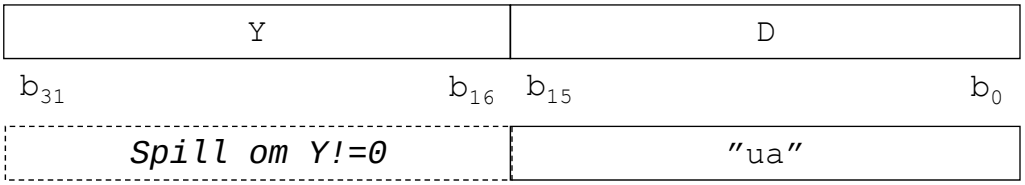
Pseudo språk:

```
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) ...

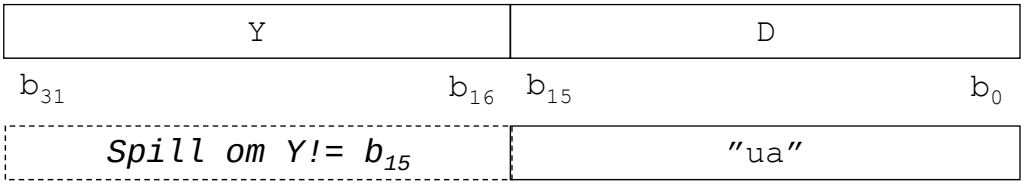
Pseudo språk:

```
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:

$a = ah \times 2^{16} + al$, ah = MSW(a) och al = LSW(a)

$b = bh \times 2^{16} + bl$, bh = MSW(b) och bl = LSW(b)

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:

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

Eftersom vi bara kräver 32 bitars resultat kan vi 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*

$R = X - Q \times Y$

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

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.

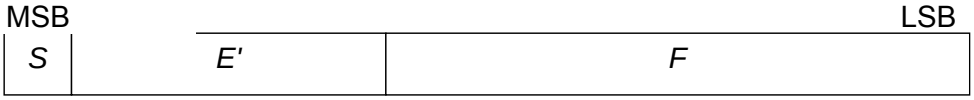
IEEE-754 flyttal standard specificerar:

- 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

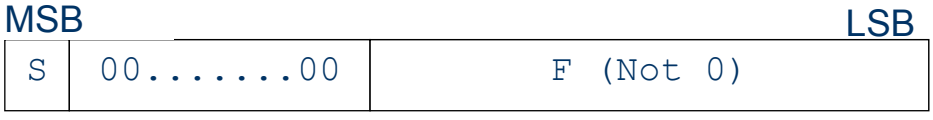
”Första ordningens singulariteter”

± 0

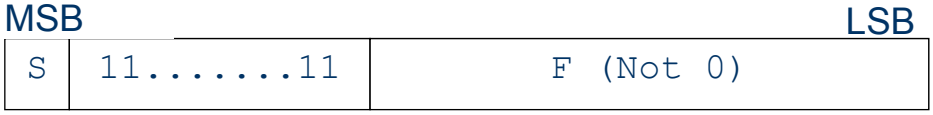
± ∞

”Andra ordningens singulariteter”

Denormaliserat



”Not a Number”



Typomvandlingar

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")

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.

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.

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

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

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*.