Kompendium:

”Maskinnära programmering för HC12”

Roger Johansson, 2013

1 Maskinnära programmering för HC12

Kapitlets syfte är att illustrera såväl möjligheter som begränsningar med maskinnära programmering. Större delen behandlar maskinnära programmering i allmänhet såväl som högnivåprogrammering. Dedicerade avsnitt tillämpar därefter kodningskonventioner och enkel översättarteknik för kodgenerering.

Assemblerspråket

Att programmera en dator i maskinspråk är en krävande och tidsödande uppgift, i stället använder vi assemblerspråk. Processen att översätta ett assemblerprogram till ett program i maskinspråk kallas att ”assemblera” (från engelskans *assembly*, ’sätta samman’). Utifrån ett program skrivet i assemblerspråk sätts maskinspråket samman av *assemblatorn*.

Vi inleder detta kapitel med att beskriva den generella uppbyggnaden av ett assemblerspråk. Vi beskriver dess relation till maskinspråk och behandlar översättningsprocessen mellan assemblerspråk och maskinspråk. I denna inledning kommer vi att behandla flera viktiga begrepp som är vanliga vid programutveckling.

Assemblerspråket är specifikt för en viss centralenhet (mikroprocessor) och vi exemplifierar här med Freescales familj av MC68HCS12-kretsar *CPU12* (*Central Processing Unit 12*).

Maskininstruktionen

En centralenhet tolkar och utför *maskininstruktioner*. En maskininstruktion är ett binärt ord, dvs en följd av nollor och ettor organiserade i en fix längd (*maskinord*). Maskinordets längd varierar mellan olika typer av centralenheter men är det samma för en given centralenhet. Man talar exempelvis om 8-, 16-, 32- eller 64-bitars mikroprocessorer. Maskininstruktioner kan ha olika längd, dvs. bestå av olika antal maskinord, beroende på instruktionens komplexitet. Man talar om olika *instruktionsformat* och menar då hur en instruktion är uppdelad i *operationskod* och *operander*. Med *operationskod* (OP-kod) menas den styrinformation som processorn läser för att bestämma *vad* den skall utföra. OP-koden anger också *hur många operander* som finns och *var* dessa finns.

En *assemblerinstruktion* har en mycket enkel uppbyggnad, en komplett maskininstruktion anges med en *mnemonic* och eventuella operander. Speciella tecken används för att skilja mellan olika adresseringssätt (*addressing modes*). En assemblerinstruktion motsvarar alltså en maskininstruktion med ett direkt 1:1 förhållande, för varje korrekt assemblerinstruktion finns alltså endast en maskininstruktion.

Exempel 1.1

CPU12-instruktionen *ReTurn from Subroutine* har mnemonic RTS.

I assemblerprogrammet skrivs den:

```
RTS
```

vid assembleringsprocessen översätts den till maskininstruktionen:

00111001 (binär form) vilket också kan skrivas $39 (hexadecimal form)

Assemblerspråket är *radorienterat*, dvs en rad i ett assemblerprogram kan innehålla högst en assemblerinstruktion. För instruktioner med operander anges operanden efter instruktionens mnemonic. För att skilja mnemonic från operand används ”blanksteg” dvs. mellanslag (”SPACE”) eller tabulator (”TAB”).

Assemblerinstruktionens adresseringssätt anges ofta med någon form av specialtecken. Freescale använder exempelvis ofta tecknet ’#’ för att ange omedelbar adressering (*immediate adressing*). Adresseringssättet anger att instruktionens operand utgör data som då följer omedelbart efter operationskoden.

Exempel 1.2

CPU12-instruktionen *LoaD Ackumulator A* har mnemonic LDAA.
I assemblerprogrammet skrivs den

```
LDAA  operand
```

operanden kan anta flera former beroende på vilket adresseringssätt som avses, exempelvis innebär:

```
LDAA  #45
```

att värdet 45 placeras i ackumulator A, medan formen

```
LDAA  45
```

anger att värdet som finns *på adress* 45 i datorns minne placeras i ackumulator A.

Vi återkommer strax till en uttömmande behandling av assembler/maskininstruktioner men ska först behandla ett annat vanligt element, *assemblerdirektiv*.

Assemblerdirektiv

Assemblerspråket innehåller också en rad olika *direktiv* till assemblatorn. Assemblerdirektiv används för att instruera assemblatorn att göra något. Assemblerdirektiv är vanligtvis, precis som mnemonics, specifika för den centralenhet som används, man kan dock ofta se en viss enhetlighet mellan assemblatorer för någon processorfamilj, eller processorfamiljer från samma tillverkare.

Exempel 1.3 Reservera minnesutrymme

Assemblerdirektiv kan användas för att reservera minnesutrymme. Assemblers för CPU12 accepterar exempelvis följande direktiv:

```
RMB   antal        Reserve Memory Bytes
```

där *antal* kan anges med godtycklig talbas och anger det antal *bytes* (8-bitars ord) man vill reservera. På samma sätt återfinns ofta direktivet:

```
RMW   antal        Reserve Memory Words
```

Detta har samma funktion men reserverar i stället words (16-bitars ord). Följande direktiv exempelvis, är därför funktionellt likvärdiga:

```
RMB   4      och
RMW   2
```

Exempel 1.4 Reservera minnesutrymme och initiera minnesinnehåll

Assemblerdirektiv kan också användas för att låta assemblern reservera minnesinnehåll *och* initiera minnesinnehåll med data. Assemblers för CPU12 accepterar exempelvis följande direktiv:

```
FCB   data         Form Constant Byte(s)
```

där *data* exempelvis kan anges med godtycklig talbas och då anger det värde man vill initiera. Om värdet inte ryms inom 8 bitar kommer assemblern att generera varnings- eller felutskrift.
På samma sätt återfinns ofta direktivet:

```
FCW   data         Form Constant Word(s)
```

Detta har samma funktion men initierar i stället words (16-bitars ord). Följande direktiv exempelvis, är därför funktionellt likvärdiga:

```
FCB   $10,$20      och
FCW   $1020
```

Exempel 1.5

Assemblerdirektivet ORG (origin) används för att ange an absolut startadress i datorns minne. På detta sätt kan vi styra placeringen av kod respektive data till fixa adresser.

```
ORG   $2000
```

.. första instruktion (eller data) här placeras på adress $2000

```
ORG   $2800
RMB   10    reservera minnesare 10 bytes, med start på adress $2800
ORG   $3000
FCB   1,2,3,4,5   initiera minnesarea med start på adress $3000
```

Observera att assemblerdirektivet ORG i sig inte ger upphov till att vare sig kod eller data genereras för måldatorn. Direktivet påverkar endast placeringen av efterföljande kod/data i minnet.

Symbolhantering

Symboler används bland annat för att markera positioner i ett assemblerprogram. Varje symbolnamn måste väljas *unikt* dvs, får bara definieras *en* gång i programmet. Symbolnamnets längd måste vara begränsat. Exempelvis är symbolnamnets längd begränsat till 256 tecken i QA/RA-assemblatorerna. Symbolens *första* tecken måste vara en bokstav (a-z eller A-Z) *eller* en "understrykning". Observera att de svenska tecknen å, ä och ö visserligen tillåts av QA/RA assemblatorerna men de får vanligtvis *inte* förekomma i symbolnamn.

Införandet av symboler underlättar programmeringsarbetet. Symboler kan exempelvis motsvara godtyckliga adresser i måldatorns minne. De är då i själva verket en slags ”markering” av någon position vars absoluta adress egentligen är betydelselös för programmets funktion. En sådan symbol kallas *relokerbar*, dvs ”möjlig att flytta”.

Exempel 1.6 Symboliska namn för variabler

```
            ORG   $2800
start       RMB   1
stopp       RMB   1
```

Symbolen ’start’ hamnar här på adress $2800, medan symbolen ’stopp’ får absoluta adressen $2801. Genom att, i programmet, referera dessa adresser via symbolerna behöver vi inte bekymra oss om de aktuella adresserna:

```
            LDAA  start
            LDAA  stop
```

I vissa fall är det inte möjligt att använda relokerbara symboler. Exempelvis finns, i varje datorsystem, så kallade *portar*, med en fast adress i minnesarean. Vi kan fortfarande använda symboliska namn på portar men vi måste använda ett annat assemblerdirektiv.

Exempel 1.7

Assemblerdirektivet EQU (equate) används för att ersätta en symbol med ett numeriskt värde. Direktivet används på följande sätt:

```
symbolnamn  EQU   värde
```

Då ”symbolnamn” används i uttryck, assemblerdirektiv eller som operand i instruktioner kommer assemblern att ersätta symbolen med”värde”.
Det är inte meningsfyllt att prata om storlek i samband med EQU-direktivet. Följande sekvens visar exempelvis hur samma symolnamn är meningsfyllt i två fall men inte i det tredje fallet:

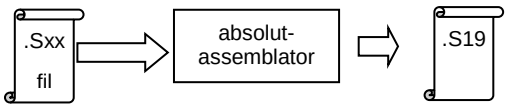
```
hugebyte    EQU   257          ryms ej med 8 bitar, dock med 16...
            FCW   hugebyte     Ok!
            FCB   hugebyte     FEL...
```

Relokerbarhet

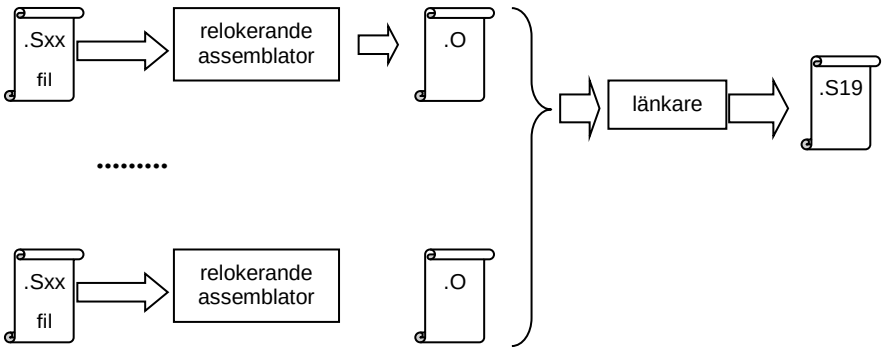
Absoluta symboler används för att representera data och adresser som inte får ändras. Exempelvis måste en symbol för adressen till en IO-port vara absolut. Vi har redan sett assemblerdirektivet `EQU` som kan användas för att definiera en sådan symbol. Med *relokerbarhet* menar vi egenskapen att en symbols värde kan komma att ändras utan att det påverkar ett program. Vi har sett exempel på direktiv även för sådana symboler (`RMB, FCB, FCW`) etc. Eftersom symboler används, rent allmänt, för att representera någon speciell position i programmet, och därmed också någon (oväsentligt *vilken*) adress i datorn minne, är det naturligt att symbolerna också är relokerbara.

En *absolut assemblator* kan användas för att översätta *en* källtext till ett maskinprogram. Alla symboler som refereras förutsätts då vara kända vid assembleringen. En *relokerande assemblator* översätter inte källtexten direkt till ett maskinprogram. I stället skapas så kallad *objektkod*, där

symbolers adresser och alla referenser till symboler sparas tillsammans med den kod och data som så småningom ska komma att bli det färdiga maskinprogrammet. Det slutgiltiga maskinprogrammet skapas här av en *länkare*, vars uppgift är att kombinera flera filer med objektkod till ett slutgiltigt program med maskinkod. Detta innebär att symboler som definierats i en fil med källtext kan refereras från en annan källtextfil.



En absolutassemblators arbetssätt



Arbetssätt hos en relokerande assemblator med länkare

Assemblerprogrammering med CPU12

Vi inleder dessa avsnitt med att beskriva hur man skriver ett komplett assemblerprogram för CPU12. Vi använder några av de direktiv vi introducerat för att styra assemblatorn.

Nästa avsnitt behandlar *programmerarens bild*, CPU12's *instruktionsgrupper* med en snabb översikt av instruktionsuppsättningen. Avsnittet avslutas med en utförlig beskrivning av processorns olika *adresseringssätt*.

Därefter följer ett längre avsnitt som beskriver ett strukturerat sätt att programmera CPU12 i assembler. Här behandlas bland annat aritmetiska operationer, jämförelser och test, och hur du styr programflödet. Efter en första genomläsning kan du utnyttja detta avsnitt som ett uppslagsverk för att hitta svaren på hur du löser olika problem då du konstruerar dina assemblerprogram.

Kapitlet avslutas med ett avsnitt som behandlar maskinnära programmering i C, dvs hur du blandar kod skriven i programspråket 'C' med dina assemblerprogram. Speciellt beskriver vi konventioner som används av XCC12 ("Cross C Compiler 12").

Ett assemblerprogram byggs upp av *kod*, *data* och *assemblerdirektiv*. Koden utgörs av *instruktionssekvenser* som kan utföra operationer på data. Data kan utgöras av *konstanter* eller *variabler*. Assemblerdirektiv kan användas bland annat för att reservera minnesutrymme för data, ange *var* kod respektive data ska placeras m.m.

Det finns strikta regler för hur assemblerprogrammet ska se ut. Programmet läses av assemblatorn, rad för rad, och översätts till *maskinkod* dvs, mönster av ettor och nollor. Maskinkoden kan tolkas och utföras av processorn.

En rad, i assemblerprogrammet delas in i maximalt 4 fält. Första fältet kan enbart användas för att ange en "etikett". Man väljer då ett *symboliskt namn* och kan därefter använda detta namn för att

ange (referera) denna position i programmet. Anledningen till att man använder sådana symboliska namn är, som tidigare sagts, att man då slipper skriva *absoluta* minnesadresser i programmet för de positioner vars absoluta adresser inte har någon egentlig betydelse.

Nästa fält i assemblerprogrammets rad kan vara en instruktion eller ett direktiv. Det är viktigt att förstå skillnaden mellan en *assemblerinstruktion* och ett *assemblerdirektiv*. Direktivet instruerar assemblatorn att *göra* någonting vid assembleringstillfället medan instruktionen *översätts till maskinkod* för att så småningom utföras av processorn vid exekveringen av programmet.

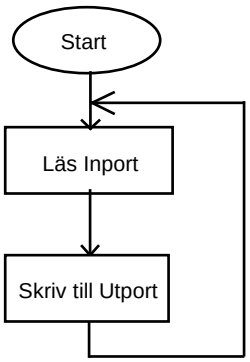
Assemblatorns tredje fält ska ange eventuella operander för assemblerinstruktioner. Detta fält används även tillsammans med assemblerdirektiv men vi kallar då detta *argument* till direktivet.

Assemblerraden kan avslutas med en godtycklig kommentarstext, dvs. någon beskrivning av vad som utförs så att programmet blir lättare att läsa och förstå.

Fälten skiljs åt med blanktecken, dvs ”tabulatur” eller ”mellanslag”. Detta innebär att man inte kan använda blanksteg i symbolnamn, eller exempelvis sätta in blanksteg mellan operanderna (*även* om detta skulle se prydligare ut).

Första assemblerprogrammet

Vårt första assemblerprogram visar exempel på hur vi kan läsa data (8 bitar) från en inport placerad på adress $600 i minnet, vi skriver därefter samma data till en utport på adress $400 i minnet, detta upprepas i en “oändlig slinga”.



Flödesplan för första assemblerprogrammet

Assemblerkod, det första assemblerprogrammet:

```
; Programmet läser från en inport och kopierar till en utport
InPort        EQU             $600
OutPort       EQU             $400
              ORG             $1000
Start:
              LDAB            InPort      ; Läs från inporten…
              STAB            OutPort     ; Skriv till utporten
              BRA             Start       ; Börja om...
```

Symbolfält, blankt eller kommentar	Instruktion (mnemonic) eller assembler-direktiv	Operand(er) till instruktion eller argument till direktiv	Eventuell kommentarstext

Fälten separeras med blanktecken, dvs ”tabulatur” eller ”mellanslag”.

Assemblerradens fjärde fält kan användas för kommentarer till enskilda instruktioner eller direktiv. För att ytterligare öka läsbarheten kan vi i bland tvingas skriva betydligt längre kommentarer i programmet. Vi kan då, genom att ange ett *semicolon* (;) eller en *stjärna* (*) i radens första position, använda återstoden av denna rad för kommentarer.

CPU12 - programmerarens bild

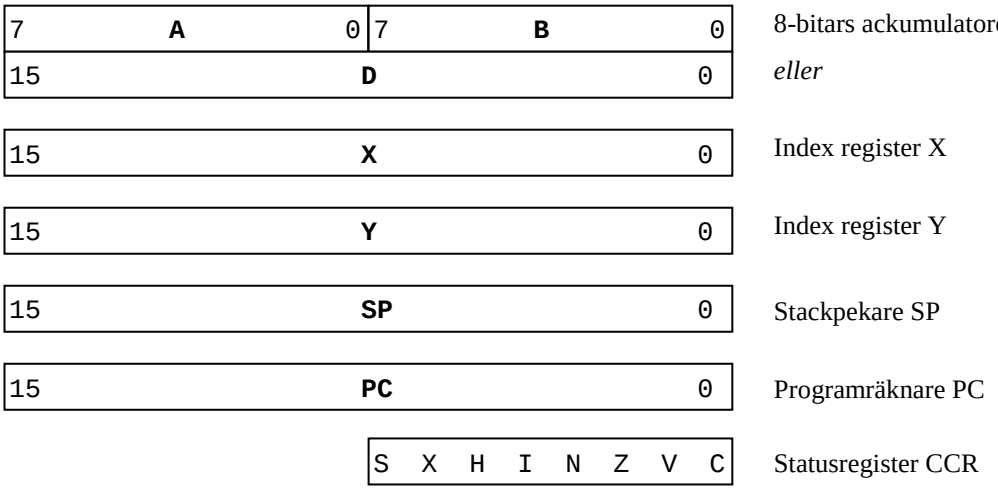
I “programmerarens bild” av en processor ingår processorns arbetssätt, dess registeruppsättning, vilka funktioner dessa register har, processorns adresseringssätt, dvs olika möjligheter att ange var data finns och processorns instruktionsuppsättning. Det ska vi ägna oss åt i detta avsnitt.

Registermodell

CPU12 har olika typer av register, en av dessa typer kallas ”ackumulator” (*accumulator*) avsett för aritmetik och logiska operationer. Som namnet antyder ackumuleras resultat i registret, dvs vid binära operationer innehåller från början registret en operand, den andra operanden anges via något adresseringssätt, resultatet av operationen ersätter det tidigare innehållet i ackumulatorn.

Utöver ackumulatorregistren finns:

- två 16-bitars ”index-register”, X och Y, huvudsakligen avsedda för adressberäkningar.
- en 16-bitars stackpekare, SP, systemets hårdvarustack.
- en 16-bitars programräknare, PC, för adressering av den instruktion som ska exekveras
- ett 8-bitars statusregister, CCR, med bl.a. aritmetikflaggor och avbrottsflaggor.



Ackumulerande register

Det finns två ackumulatorregister, A och B som kan användas oberoende av varandra. Ordlängden hos dessa är 8 bitar. Vissa instruktioner (LDD, STD, ADDD etc) använder dock båda ackumulatorerna samtidigt. Ackumulatorregistren sätts då samman av instruktionerna och bildar i stället ett 16-bitars ord där ackumulator A innehåller de 8 mest signifikanta bitarna och ackumulator B de 8 minst signifikanta bitarna. De flesta operationer kan utföras på såväl A som B. Det finns dock några få undantag; ABA (Add B to A), SBA (Subtract B from A) och CBA (Compare B to A). I dessa fall är operandernas ordning viktig. Den decimaljusterande instruktionen DAA (Decimal Adjust A) som används i samband med BCD-aritmetik finns bara för ackumulator A.

Indexregister

Nästa typ av register kallas ”indexregister”. Syftet med indexregister är att tillfälligt kunna beräkna adressen till någon operand, därefter, med hjälp av ackumulatorn utföra operationer med operanden på den beräknade adressen. En annan benämning på denna typ av register är helt enkelt ”adressregister”. Flera adresseringssätt tillåter användning av indexregister där alltså innehållet i registret tolkas som adress till instruktionens operand.

Stackpekare

Stackpekaren är ett register med speciella funktioner. Dess innehåll sägs vara en ”pekare” till systemets ”stack”. Stacken, oftast en liten del av primärminnet, används för att (automatiskt) spara registerinnehåll (PC och CCR) vid subrutinanrop som exempelvis JSR (Jump to SubRoutine) och CALL (Call subroutine in paged memory) sparas automatiskt registerinnehåll som PC och CCR på stacken. Instruktioner som RTS (ReTurn from Subroutine) och RTC (ReTurn from Call) återställer automatiskt registerinnehåll så att exekveringen fortsätter omedelbart efter subrutinanropet.Vid avbrott, sparas automatiskt samtliga processorns registers innehåll till stacken.

Stacken kan av programmeraren också användas för lagring av mellanresultat (temporära data).

Programräknare

Programräknaren är det speciella register som används för att adressera nästa instruktion som skall utföras. Vid instruktionsexekvering uppdateras programräknaren av hårdvaran. Det finns också adresseringssätt som omfattar programräknarens värde, exempelvis vid villkorliga programflödesändringar.

Statusregister

Statusregistret innehåller såväl statusbitar, sätts av hårdvaran vid aritmetiska/logiska operationer, som styrbitar för att bestämma processorns beteende under olika speciella omständigheter:

S: *stop disable*, den speciella instruktionen `STOP`, används för att avbryta processorns exekvering och försätta den i ett tillstånd med minimal strömförbrukning. I vissa applikationer är detta inte det lämpligaste och därför finns denna bit som måste nollställas för att `STOP`-instruktionen ska utföras. Om S-biten däremot är 1 behandlas `STOP` som *no operation*. S-biten sätts till 1 av hårdvara vid RESET.

I: *mask interrupt*, denna bit används för att ”maskera”, dvs utestänga en avbrottsbegäran via processorns IRQ ingång. Vid ett avbrott sätts biten av hårdvaran efter det att processorns registerinnehåll sparats till stacken. Biten sätts för att en avbrottsrutin ska kunna utföras utan att bli ytterligare avbruten (nästlade avbrott). I en avbrottsrutin återställs normalt I-biten automatiskt då rutinens sista instruktion, RTI, utförs eftersom processorns registerinnehåll, bl.a. CCR återställs från stacken och hårdvaran satt I-biten efter att ha sparat registerinnehåll vid avbrottsbegäran.

X: *enable non-maskable interrupts*, HC12 har två olika avbrottsmekanismer, den generella IRQ som kan maskeras genom att sätta I-biten till 1 och XIRQ, en avbrottsingång som *inte* kan maskeras. För att undvika att XIRQ oavsiktligt aktiveras exempelvis vid spänningstillslag finns X-biten. Denna bit sätts till 1 av hårdvara vid RESET. Då biten är 1 accepteras inte XIRQ. Ett användarprogram kan efter vederbörlig initiering nollställa biten varefter XIRQ kommer att accepteras. Då biten nollställts kan den inte ettställas igen av programvara och XIRQ fungerar därför därefter som icke-maskerbart avbrott.

Statusbitarna sätts, nollställs eller lämnas oförändrade beroende på instruktion. Bitarna har följande betydelse:

H: halfcarry, används vid BCD-aritmetik där fyra bitar representerar ett tal 0-9. Det största tal som kan representeras i exempelvis ackumulator A blir då 99. H sätts vid addition av två tal, tolkade på BCD-form där addition av de minst signifikanta fyra bitarna genererar ”carry”. H-biten påverkas endast av instruktionerna `ABA` (add ackumulator B to A), `ADD` (add without carry) och `ADC` (add with carry). Efter någon av dessa instruktioner kan sedan `DAA` (decimal adjust A) användas för att justera innehållet i ackumulator A till korrekt BCD-format.

N: *negative*, är den mest signifikanta biten i ett resultat. Den vanliga tolkningen är därför N som teckenbit (tvåkomplementform) dvs N-biten sätts till 1 om resultatet är mindre än noll, biten sätts till noll annars. Observera att de flesta instruktioner, inte enbart aritmetiska, påverkar N-flaggan.

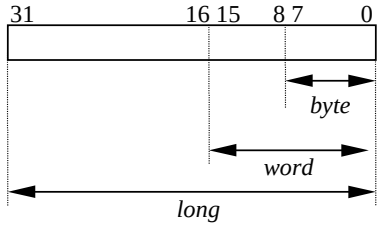
Z: *zero*, biten sätts till 1 då ett resultat blir noll, biten sätts till noll annars. Observera att de flesta instruktioner, inte enbart aritmetiska, påverkar Z-flaggan.

V: *overflow*, biten sätts till 1 vid tvåkomplementspill, biten sätts till noll annars. Observera att de flesta instruktioner, inte enbart aritmetiska, påverkar V-flaggan.

C: *carry*, sätts/nollställs som resultat av addition och som *borrow* vid subtraktion. C-biten används också för att indikera fel vid multiplikation och division. Det finns också skiftinstruktioner där C-biten ingår i skiften.

Datatyper

Begreppet *datatyper* introduceras ofta med användningen av högnivåspråk. Exempel på högnivåspråk är **C**, **Java**, **ADA** etc. Grundläggande för alla datatyper är den representation som kan ges typen i den underliggande hårdvaran. Den i särklass vanligaste datatypen är *heltal*. Maskinvaran sätter gränser för hur små eller hur stora heltal som *enkelt* kan representeras, detta avspeglas direkt utav det antal bitar som används vid exempelvis aritmetiska operationer. För CPU12 gäller att full 16-bitars aritmetik kan utföras som enkla operationer (instruktioner).



Datatyper med storlekar byte, word och long dvs 8,16 eller 32 bitar

I programspråket 'C' finns enkla och sammansatta datatyper. Låt oss titta närmre på dessa och se hur vi lämpligast representerar dom i ett CPU12-system med avseende på den registeruppsättning vi tidigare studerat. Observera att i 'C' är representationen för datatyperna `char`, `short` och `long` definierade av språket, medan storleken hos pekar-typer och heltalstypen `int` är implementationsberoende, dvs dessa typer kan ha olika storlek beroende på den underliggande hårdvaran.

Enkla datatyper med CPU12

```
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 */
```

Representationen för datatypen `int` bör väljas för bästa prestanda. Eftersom CPU12 inte direkt hanterar 32-bitars tal är det lämpligaste valet 16 bitar, dvs storlek *word*. Detta innebär då att datatyperna `short` och `int` är ekvivalenta.

För pekar-typerna baseras valet av representation på de tillgängliga adressregistren (X och Y) och det lämpligaste valet blir därför 16 bitar.

```
char        *cptr;      /* pekar (16 bitar) på 8-bitars datatyp */
short       *sptr;      /* pekar (16 bitar) på 16-bitars datatyp */
int         *iptr;      /* pekar (16 bitar) på 32-bitars datatyp */
```

För pekar-typerna har vi dessutom hos CPU12 en liten komplikation eftersom arkitekturen också definierar ett expanderat ("paged") minnesutrymme. Detta kan adresseras med hjälp av extra register (xPAGE) och ger då en maximal minnesrymd om 8 Mbyte. Användning av det expanderade minnesutrymmet resulterar dock i större kodstorlek varför det är lämpligt att hålla i sär de båda pekar-typerna. I många CPU12-kompilatorer har man därför infört attributet `far` för en pekar-typ som kan användas för att adressera även det expanderade minnet. Denna pekartyp kommer då att få storleken 23 (16+7) bitar. Användning av långa pekartyper är i högsta grad beroende på den använda C-kompilatorn och vi koncentrerar oss här på den grundläggande pekar-typen (16 bitar).

I **C** kallas en sammansatt datatyp `struct`. Denna kan sättas samman av såväl enkla datatyper som andra sammansatta datatyper. En pekare till en `struct` har samma restriktioner som övriga pekartyper och representeras därför lämpligen med 16 bitar. Vid kompilering av deklarationen kan offseten till de ingående variablerna bestämmas eftersom storleken av alla datatyper som ingår i `structen` måste vara kända.

Exempel 1.8 En på sammansatt datatyp

```
struct composite  {
       long int     a;
       short int    b;
       short int    c;
};
```

Ett adressregister, exempelvis **X** kan användas för att peka på den första komponenten (variabel **a**). Adresserna till de övriga blir då:

Variabel **b**: **X**+4

Variabel **c**: **X**+6

Flyttal

CPU12 har inga instruktioner för hantering av flyttal. Alla flyttalsoperationer måste därför implementeras i form av programbibliotek.

Instruktionsformer

I CPU12:s instruktionsuppsättning finns instruktioner med *ingen, en* eller *två* operander. Instruktioner med två operander (*binära operationer*) finns i två syntaktiska varianter. I den första varianten anges båda operander explicit i operandfältet. I en andra variant anges en av operanderna i stället som en del av instruktionens mnemonic.
Formen för en instruktion med två explicita operander är

mnemonic källoperand,destinationsoperand

där *mnemonic* anger vilken instruktion det är frågan om. Denna svarar precis mot en *maskininstruktion*. Operanderna anger *var* data skall läsas/och eller skrivas, av instruktionen.

Formen för en instruktion där en av operanderna implicit är något register:

mnemonicRegister operand

Exempel 1.9

Instruktion med två explicita operander

```
       movb  #1,variable
  mnemonic     källoperand  destinationsoperand
```

Instruktion med implicit källoperand register A

```
       staa  variable
mnemonic    källoperand      destinationsoperand
```

Instruktion med implicit destinationsoperand register A

```
       ldaa  variable
                      källoperand
mnemonic    destinationsoperand
```

Instruktioner med *en* operand kallas *unära operationer*. Former för instruktioner med *en* operand är:

mnemonic *operand*
mnemonic *Register*

Exempel 1.10

Instruktionen COM (*complement*) inverterar varje bit hos operanden

```
COM   variable    ; invertera variabel i minnet
COMA              ; invertera innehållet i register A
```

För de flesta en-operandsinstruktioner gäller att operanden läses, operationen utförs och operanden uppdateras med det nya resultatet. Det finns dock undantag exempelvis testinstruktioner som endast påverkar flaggorna i CCR. Slutligen finns instruktioner som *saknar* operand, exempelvis `NOP` (*no operation*), `RTS` (*return from subroutine*), `STOP` (*stop execution*).

Adresseringssätt

CPU12 tillhandahåller 15 olika adresseringssätt (*eng: adressing mode*). Med adresseringssätt menas det sätt på vilket *effektiva adressen* (EA), bestäms. Med effektiva adressen menas adressen till data som ska användas av instruktionen. Tabell 1.1 sammanfattar de tillgängliga adresseringssätten:

Tabell 1.1 Adresseringssätt

Adresseringssätt	Operand	Beskrivning
Inherent	Saknas	Information om EA finns implicit i instruktionen

Immediate	`#opr8`
	eller
	`#opr16i`
		Data följer omedelbart efter instruktionen. Storlek (8 eller 16 bitar) bestäms implicit av instruktionen.

Direct	`opr8`	EA i adressintervallet $0000-$00FF där Opr8 anger de 8 minst signifikanta bitarna.
Extended	`opr16`	EA i adressintervallet $0000-$FFFF

Relative	`rel8`
	eller
	`rel16`
		8 eller 16 bitars offset, tolkad som tal med tecken, adderas till programräknarens värde och bildar EA.

Indexed
5 bits offset
	`oprx5,x`
	`oprx5,y`
	`oprx5,sp`
	`oprx5,pc`
		5-bitars konstant, tolkad som tal med tecken, adderas till värdet I register (X,Y,SP eller PC). Resultatet är EA.

Indexed
pre-decrement
	`oprx3,-x`
	`oprx3,-y`
	`oprx3,-sp`
		En konstant, 1-8, subtraheras från värdet i register (X,Y eller SP). Registret uppdateras med detta värde. Resultatet är EA.

Indexed
pre-increment
	`oprx3,+x`
	`oprx3,+y`
	`oprx3,+sp`
		En konstant, 1-8, adderas till värdet i register (X,Y eller SP). Registret uppdateras med detta värde. Resultatet är EA.

Indexed
post-decrement
	`oprx3,x-`
	`oprx3,y-`
	`oprx3,sp-`
		Värdet i register (X,Y eller SP) är EA. Efter operation subtraheras en konstant, 1-8, slutligen uppdateras registret med detta värde.

Indexed
post-increment
	`oprx3,x+`
	`oprx3,y+`
	`oprx3,sp+`
		Värdet i register (X,Y eller SP) är EA. Efter operation adderas en konstant, 1-8, slutligen uppdateras registret med detta värde.

Indexed
ackumulator offset
	`A|B|D,x`
	`A|B|D,y`
	`A|B|D,sp`
	`A|B|D,pc`
		Värdet i register (X,Y,SP eller PC) adderas till värdet i ackumulator (A eller B eller D) och bildar EA.

Indexed
9 bits offset
	`oprx9,x`
	`oprx9,y`
	`oprx9,sp`
	`oprx9,pc`
		9-bitars konstant, tolkad som tal med tecken, adderas till värdet i register (X,Y,SP eller PC). Resultatet är EA.

Indexed
16 bits offset
	`oprx16,x`
	`oprx16,y`
	`oprx16,sp`
	`oprx16,pc`
		16-bitars konstant adderas till värdet I register (X,Y,SP eller PC).
		Resultatet är EA.

Indexed indirect
16 bits offset
	`[oprx16,x]`
	`[oprx16,y]`
	`[oprx16,sp]`
	`[oprx16,pc]`
		16-bitars konstant adderas till värdet i register (X,Y,SP eller PC).
		Resultatet är *adressen* till EA.

Indexed indirect
D ackumulator offset
	`[D,x]`
	`[D,y]`
	`[D,sp]`
	`[D,pc]`
		Värdet i register (X,Y,SP eller PC) adderas till värdet i ackumulator D och bildar *adressen* till EA.

Var och en av CPU12:s adresseringssätt kommer nu att beskrivas i detalj med exempel på hur effektiva adressen bestäms.

Inherent adressering

Detta adresseringssätt innebär att instruktionen inte kräver någon extra operandinformation. Eventuella operander är fullständigt bestämda av instruktionen.

Exempel 1.11 Inherent adressering

```
NOP  ;    (no operation)
RTS  ;    (return from subroutine)
INX  ;    (increment X)
```

Omedelbar (immediate) adressering

Operanden är med ”omedelbar adressering” placerad direkt efter operationskoden och instruktionen kräver därför ingen speciell adressberäkning. Symbolen ’#’ används för att ange omedelbar adressering. Det är ett vanligt fel att oavsiktligt utelämna denna. Exempelvis betyder:

```
LDAA #$10
```

att hexadecimala talet $10 placeras i ackumulator A medan instruktionen:

```
LDAA $10
```

anger att det 8-bitars tal som finns på adress $10 placeras i ackumulator A.

Operanden kan vara 8 eller 16 bitar vilket bestäms vid assembleringen beroende på vilken instruktion som används:

Exempel 1.12 Omedelbar adressering

```
LDAA #$10      ; 8 bitars operand
LDAB #10       ; 8 bitars operand
LDD  #$1234    ; 16 bitars operand
```

Direkt (direct) adressering (HCS12)

Detta adresseringssätt kallas i bland också ”page zero adressing” eftersom den används för operander i adressintervallet $0000-$00FF. Eftersom endast de 8 minst signifikanta bitarna av adressen då behöver anges som operand sparar detta utrymme och exekveras snabbare. Observera att syntaxen kan vara densamma som vid ”utökad adressering” (se nedan). Exempelvis kan instruktionen:

```
LDAA $10
```

använda direkt adressering eller utökad adressering. För att fullständigt ange att direkt adressering ska utnyttjas, använd symbolen ’<’.

Exempel 1.13 ”Direct page”-adressering

```
LDAA <$10 ; tvingande direkt adressering
LDAA $10  ; tvetydigt, assemblator avgör adresseringssätt.
```

Direkt (direct) adressering (HCS12X)

HCS12X har utökats med ett 8-bitars register **DP** (*Direct Page Register*) som används vid direktadressering. Innehållet i **DP**-registret bildar de 8 mest signifikanta bitarna och tillsammans med operanden kan då 64 kByte av primärminnet adresseras. **DP**-registret finns i de speciella minnesavbildningsblock (Memory Map Modeule, MMC) som lagts till i HCS12X. Vid RESET är innehållet i DP 0 och funktionen kompatibel med HCS12.

Utökad (Extended) adressering

Med detta adresseringssätt kan 64-kByte av minnet adresseras. Adressen kodas alltså med 16 bitar.

Exempel 1.14 Utökad adressering

Följande instruktioner kopierar båda innehållet på adress 10 till register A

```
LDAA >$10      ; tvingande utökad adressering
LDAA $10
```

Följande instruktion kopierar en byte från adress F030

```
LDAA $F020
```

Följande instruktion kopierar 2 bytes från adresserna F030 och F031

```
LDD  $F030
```

Instruktionsgrupper

Det finns en rad omständigheter som påverkar såväl valet av instruktioner ur en instruktionsuppsättning som de operander som kan användas till instruktionerna. I detta avsnitt redogörs för de *grupper* av instruktioner som man kan identifiera i CPU12:s instruktionsuppsättning. Vi ger exempel på hur instruktionerna kan användas För en fullständig beskrivning av varje instruktion (varianter, operander, flaggsättning mm.) måste du dock studera *instruktionslistan*.

Instruktioner för att kopiera data i minnet

Instruktionerna MOVB (*move byte*) och MOVW (*move word*) används för att kopiera data direkt i minnet utan att något register behöver användas för mellanlagring.

Tabell 1.2: MOVE-instruktioner

Mnemonic	Funktion	Operation
MOVB	Move byte (8 bitar)	$(M_1) \rightarrow M_2$
MOVW	Move word (8 bitar)	$(M:M+1)_1 \rightarrow M:M+1_2$

Notera att det hos HCS12 finns restriktioner för adresseringssätt hos såväl källoperand som destinationsoperand. För HCS12X tillåts betydligt fler kombinationer.

Det finns också en rad olika instruktioner som använder register för att kopiera, flytta data etc (LOAD-STORE). LOAD-instruktioner måste också användas om en efterföljande instruktion förutsätter operanden i något register.

LOAD-instruktioner används för att kopiera data från minnet till något register. Det är viktigt att skilja på typerna LD (*load*) och LEA (*load effective address*). Den sistnämnda typen har en begränsad uppsättning adresseringssätt och används uteslutande för manipulation av pekare, dvs *minnesadresser*.

Tabell 1.3: LOAD-instruktioner

Mnemonic	Funktion	Operation
LDAA	Load A	(M)→A
LDAB	Load B	(M)→B
LDD	Load D	$(M:M+1)_1 \rightarrow A:B$
LDS	Load SP	$(M:M+1)_1 \rightarrow SP_H:SP_L$
LDX	Load index register X	$(M:M+1)_1 \rightarrow X_H:X_L$
LDY	Load index register Y	$(M:M+1)_1 \rightarrow Y_H:Y_L$
LEAS	Load effective address into SP	Effective address→SP
LEAX	Load effective address into X	Effective address→X
LEAY	Load effective address into Y	Effective address→Y

STORE-instruktioner används för att överföra data från ett register till någon plats i minnet.

Tabell 1.4: STORE-instruktioner

Mnemonic	Funktion	Operation
STAA	Store A	(A)→M
STAB	Store B	(B)→M
STD	Store D	(A)→M, (B)→M+1
STS	Store SP	$SP_H:SP_L \rightarrow M:M+1$
STX	Store X	$X_H:X_L \rightarrow M:M+1$
STY	Store Y	$Y_H:Y_L \rightarrow M:M+1$

Exempel 1.15

Antag deklaration:

```
      char  variable;
```

och tilldelningssatsen

```
      variable = 1;
```

Denna kan kodas på flera olika sätt, exempelvis:

```
1)    MOVB  #1,variable
2)    LDAA  #1
      STAA  variable
3)    LDAB  #1
      STAB  variable
```

Valet mellan MOV respektive LOAD/STORE instruktioner beror huvudsakligen på resten av programmet. Då det gäller kodstorlek och exekveringshastighet är lösningarna likvärdiga. Fördelen med att använda MOV är då värdet (1) inte ska användas i en direkt efterföljande operation, vi behöver inte upplåta något register för tilldelningen. Om å andra sidan, nästa sats i programmet exempelvis är:

```
      variable = variable + another_variable;
```

hade det varit fördelaktigare att välja att välja något av alternativen 2) eller 3) eftersom vi då redan har den första operanden i ett register, vilket krävs för att utföra en addition (mer om detta nedan).

Instruktioner för att kopiera/flytta data mellan register

Data kan överföras direkt mellan register utan att minnet behöver användas. Det finns två varianter, TFR (transfer) för att kopiera data från ett register till ett annat och EXG (exhange) för att byta innehåll mellan två register.

Tabell 1.5: TFR-instruktioner

Mnemonic	Funktion	Operation
TAB	Transfer A to B anm: Ekv. Med `TFR A,B`	(A)→B
TAP	Transfer A to CCR anm: Ekv. Med `TFR A,CCR`	(A)→CCR
TBA	Transfer B to A	(B)→A
TFR	Transfer register to register	(A,B,CCR,D,X,Y eller SP) → (A,B,CCR,D,X,Y eller SP)
TPA	Transfer CCR to A anm: Ekv. Med `TFR CCR,A`	(CCR)→A
TSX	Transfer SP to X anm: Ekv. Med `TFR SP,X`	(SP)→X
TSY	Transfer SP to Y anm: Ekv. Med `TFR SP,Y`	(SP)→Y
TXS	Transfer X to SP anm: Ekv. Med `TFR X,SP`	(X)→SP
TYS	Transfer Y to SP anm: Ekv. Med `TFR Y,SP`	(Y)→SP

Observera att det finns speciella mnemonics för vissa registerkombinationer. Detta är en direkt följd av att HC12 utformades kompatibel med föregångaren HC11, som inte hade den generella TFR-instruktionen.

Anmärkning: Registren kan kombineras på åtskilliga sätt och om exempelvis storleken på de ingående registren är olika tillämpas trunkering eller teckenutvidgning. Jämför med instruktionen SEX (*sign extend*). HCS12X arkitekturen tillför här en rad nya registerkombinationer med mening. Se instruktionslistan för en detaljerad beskrivning av detta.

TFR instruktioner kommer till användning speciellt för att tillfälligt spara delresultat vid evaluering av uttryck men vissa programkonstruktioner kan också tvinga fram speciell registeranvändning. Vi kommer att se exempel på detta längre fram.

Tabell 1.6: EXG-instruktioner

Mnemonic	Funktion	Operation
EXG	Exchange register to register	(A,B,CCR,D,X,Y eller SP) ↔ (A,B,CCR,D,X,Y eller SP)

XGDX	Exchange D with X
	anm: Ekv. Med
	`EXG D,X`
	`EXG X,D`
		(D) ↔ (X)

XGDY	Exchange D with Y
	anm: Ekv. Med
	`EXG D,Y`
	`EXG Y,D`
		(D) ↔ (Y)

Slutligen finns instruktioner för att teckenutvidga innehållet i något 8 bitars register till ett resultat som placeras i ett 16 bitars register:

Mnemonic	Funktion	Operation
SEX	Teckenutvidga 8 bitars operand	(A,B,CCR) → (D,X,Y eller SP)

Exempel 1.16

Antag deklarationer:

```
char  c;
short s;
```

och tilldelningssatsen

```
s = c;
```

Korekt kodning av tilldelningssatsen blir:

```
LDAB  c     ; variabel c (8 bitar) till ackumulator B
SEX   B,D   ; teckenutvidga, resultat nu I A:B
STD   s     ; skriv tillbaks resultatet (16 bitar)
```

Aritmetisk operation addition

Addition av tal och en ”ackumulering” av resultatet är en grundläggande funktion hos alla datorer. Ordlängden hos de tillgängliga registren är dimensionerande för hur stora tal som kan adderas av en enstaka maskininstruktion. Hos HCS12 har vi exempelvis ADDA, ADDB för att addera 8-bitars tal och ADDD för att addera 16-bitars tal. För att addera tal med större ord måste vi upprepa additionen, men då också ta hänsyn till eventuell minnessiffra från tidigare operation. Därför finns det alltid ytterligare en variant exempelvis ADCB (*add with carry B*). Hos HCS12 finns, av historiska skäl, ytterligare additionsinstruktioner, exempelvis ABA, men sådana har ofta mindre betydelse eftersom dom bara kan användas under speciella omständigheter

Tabell 1.7: Instruktioner för addition, HCS12

Mnemonic	Funktion	Operation
ABA	Addera B till A	(A)+(B) → A
ABX	Addera B till X anm: Ekv. med LEAX B,X	(X)+(B) → X
ABY	Addera B till Y anm: Ekv. med LEAY B,Y	(Y)+(B) → Y
ADCA	Addition med carry till A	(A)+(M)+C → A
ADCB	Addition med carry till B	(B)+(M)+C → B
ADDA	Addition till A	(A)+(M) → A
ADDB	Addition till B	(B)+(M) → B
ADDD	Addition till D (A:B)	(D)+(M:M+1) →D

Låt oss nu ge några exempel på hur dessa instruktioner används.

Exempel 1.17 Addition av 8 bitars tal

Antag deklarationer:

```
char  ca,cb,cc;
```

och tilldelningssatsen

```
ca = cb + cc;
```

En kodning av tilldelningssatsen skulle kunna vara:

```
LDAB  cb    ; operand 1
ADDB  cc    ; adderas till operand 2
STAB  ca    ; resultatet skrivs i minnet
```

Exempel 1.18 Addition av 16 bitars tal

Antag deklarationer:

```
short sa,sb,sc;
```

och tilldelningssatsen

```
sa = sb + sc;
```

En kodning av tilldelningssatsen skulle kunna vara:

```
LDD   sb    ; operand 1
ADDD  sc    ; adderas till operand 2
STD   sa    ; resultatet skrivs i minnet
```

Exempel 1.19 Addition av 32-bitars tal (HCS12)

Antag deklarationer:

```
long  la,lb,lc;
```

och tilldelningssatsen

```
la = lb + lc;
```

En kodning av tilldelningssatsen skulle kunna vara:

```
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"
```

Att notera: Varken STD eller LDD påverkar C-flaggan i CCR. Om en minnessiffra (carry) genereras vid additionen av de 16 minst signifikanta bitarna finns den alltså kvar inför nästa additionsoperation. När vi adderar de 16 mest signifikanta bitarna använder vi sekvensen ADCB ... ADCA, eftersom register B utgör de minst signifikanta 8 bitarna i register D. Om en instruktion som ADED (*add with carry D*) hade funnits, hade vi använt den i stället. Eventuell carry från första additionen följer alltså med här. Slutligen adderar vi de 8 mest signifikanta bitarna av operanderna (ADCA) och på samma sätt har en eventuell carry propagerats hit. Slutligen skrivs mest signifikanta "word" tillbaks till minnet.

Den utvidgade arkitekturen HCS12X tillför följande instruktioner för addition

Tabell 1.8: Instruktioner för addition, HCS12X

Mnemonic	Funktion	Operation
ADDX	Addition till X	(X)+(M) → X
ADDY	Addition till Y	(Y)+(M) → Y
ADED	Addition till D med carry	(D)+(M:M+1)+C →D
ADEX	Addition till X med carry	(X)+(M:M+1)+C →X
ADEY	Addition till Y med carry	(Y)+(M:M+1)+C →Y

Exempel 1.20 Addition av 32 bitars tal (HCS12X)

Vi hade i stället kunnat koda föregående exempel så här:

```
LDY   lb+2  ; minst signifikanta "word" av b
LDD   lb    ; mest signifikanta "word" av b
ADDY  lc+2  ; adderas till minst signifikanta "word" av c
ADED  lc    ; adderas till mest signifikanta "word" av c, med carry
STY   la+2  ; tilldela, minst signifikanta "word"
STD   la    ; tilldela, mest signifikanta "word"
```

Aritmetisk operation subtraktion

Instruktionerna för subtraktion följer mönstret av additionsinstruktioner.

Tabell 1.9: Instruktioner för subtraktion

Mnemonic	Funktion	Operation
SBA	Subtrahera B från A	(A)-(B) → A
SBCA	Subtrahera med borrow från A	(A)-(M)-C → A
SBCB	Subtrahera med borrow från B	(B)-(M)-C → B
SUBA	Subtrahera från A	(A)-(M) → A
SUBB	Subtrahera från B	(B)-(M) → B
SUBD	Subtrahera från D (A:B)	(D)-(M:M+1) →D

Exempel 1.21 Subtraktion av 8 bitars tal

Antag deklarationer:

```
char  ca,cb,cc;
```

och tilldelningssatsen

```
ca = cb - cc;
```

En kodning av tilldelningssatsen skulle kunna vara:

```
LDAB  cb    ; operand 1
SUBB  cc    ; operand 2 subtraheras från innehållet i register B
STAB  ca    ; resultatet skrivs i minnet
```

Exempel 1.22 Subtraktion av 16 bitars tal

Antag deklarationer:

```
short sa,sb,sc;
```

och tilldelningssatsen

```
sa = sb - sc;
```

En kodning av tilldelningssatsen skulle kunna vara:

```
LDD   sb    ; operand 1
ADDD  sc    ; operand 2 subtraheras från innehållet i register D
STD   sa    ; resultatet skrivs i minnet
```

Exempel 1.23: Subtraktion av 32-bitars tal (HCS12)

Antag deklarationer:

```
long  la,lb,lc;
```

och tilldelningssatsen

```
la = lb - lc;
```

En kodning av tilldelningssatsen skulle kunna vara:

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

Precis som för addition gäller att varken STD eller LDD påverkar C-flaggan i CCR. Om en minnessiffra (*borrow*) genereras vid subtraktionen av de 16 minst signifikanta bitarna finns den alltså kvar inför nästa subtraktionsoperation. När vi subtraherar de 16 mest signifikanta bitarna använder vi sekvensen SBCB ... SBCA, eftersom register B utgör de minst signifikanta 8 bitarna i register D. Om en instruktion som SBED (*subtract with borrow D*) hade funnits, hade vi använt den i stället. Eventuell *borrow* från första subtraktionen följer alltså med här. Slutligen subtraherar vi de 8 mest signifikanta bitarna av operanderna (SBCA) och på samma sätt har en eventuell *borrow* propagerats hit. Slutligen skrivs mest signifikanta "word" tillbaks till minnet.

Den utvidgade arkitekturen HCS12X tillför följande instruktioner för subtraktion

Tabell 1.10: Instruktioner för subtraktion, HCS12X

Mnemonic	Funktion	Operation
SUBX	Subtraktion från X	(X)-(M) → X
SUBY	Subtraktion från Y	(Y)-(M) → Y
SBED	Subtraktion från D med borrow	(D)-(M:M+1)-C →D
SBEX	Subtraktion från X med borrow	(X)-(M:M+1)-C →X
SBEY	Subtraktion från Y med borrow	(Y)-(M:M+1)-C →Y

Notera att C-flaggan representerar *borrow* vid subtraktion.

Exempel 1.24: Subtraktion av 32 bitars tal (HCS12X)

Vi hade i stället kunnat koda föregående exempel så här:

```
LDY   b+2   ; minst signifikanta "word" av b
LDD   b     ; mest signifikanta "word" av b
SUBY  c+2   ; subtraheras från minst signifikanta "word" av c
SBED  c     ; subtraheras från mest signifikanta "word" av c, med borrow
STY   a+2   ; tilldela, minst signifikanta "word"
STD   a     ; tilldela, mest signifikanta "word"
```

BCD aritmetik

För BCD aritmetik används i första hand additionsintruktionerna ABA, ADCA och ADDA. Med en instruktion (DAA) kan därefter resultatet decimaljusteras korrekt baserat på H-flaggan. Notera dock att även instruktionerna ADCB och ADDB påverkar H-flaggan dock kan inte resultatet i B decimaljusteras direkt (det finns ingen DAB-instruktion...).

Mnemonic	Funktion	Operation
DAA	Decimaljustera A	$(A)_{10}$

Vid BCD-aritmetik kan ackumulator A användas för att representera två BCD-siffror. En additionsoperation ger självfallet inte rätt resultat på BCD-form. För detta krävs att delresultatet från operationen på de minst signifikanta fyra bitarna på något sätt propageras till de fyra mest signifikanta bitarna, dvs från den minst signifikanta BCD-siffran till den mest signifikanta BCD-siffran. C-flaggan kan uppenbarligen inte användas för detta, men för just denna situation finns en speciell statusflagga hos HCS12, "half-carry" eller H. H-flaggan sätts till 1 om reultatet av additionen av de minst signifikanta fyra bitarna är större än 9, annars sätts H-flaggan till 0.

DAA-instruktionen testar H-flaggan, om denna är 1, adderas konstanten 6 till de minst signifikanta fyra bitarna, H-flaggan propagerar därefter och innebär då att 1 adderas till de mest signifikanta fyra bitarna. Om additionen av de mest signifikanta bitarna nu också ger ett resultat större än 9, kommer C-flaggan att sättas till 1, annars nollställas.

Addition av större BCD-tal kan alltså utföras genom att ADDA (och därefter ADCA) används med efterföljande decimaljustering.

Datatypen BCD finns sedan länge inte i något av de större programspråken. Användbarheten av denna instruktion måste därför anses vara mycket begränsad.

Addition och subtraktion med 1

Addition (inkrementera) respektive subtraktion (dekrementera) med talet 1 är så vanliga operationer att de av prestandaskäl implementerats som egna instruktioner.

Tabell 1.11: Addition med 1

Mnemonic	Funktion	Operation
`INC`	Incrementera i minnet	(M)+$01 → M
`INCA`	Inkrementera A	(A)+ $01 → A
`INCB`	Inkrementera B	(B)+ $01 → B
`INS`	Inkrementera SP anm: Ekv. med `LEAS 1,SP`	(SP)+ $0001 → SP
`INX`	Inkrementera X anm: Ekv. med `LEAX 1,X`	(X)+ $0001 → X
`INY`	Inkrementera Y anm: Ekv. med `LEAY 1,Y`	(Y)+ $0001 → Y

Tabell 1.12: Subtraktion med 1

Mnemonic	Funktion	Operation
`DEC`	Dekrementera i minnet	(M)-$01 → M
`DECA`	Dekrementera A	(A)- $01 → A
`DECB`	Dekrementera B	(B)- $01 → B
`DES`	Dekrementera SP anm: Ekv. med `LEAS -1,SP`	(SP)- $0001 → SP
`DEX`	Dekrementera X anm: Ekv. med `LEAX -1,X`	(X)- $0001 → X
`DEY`	Dekrementera Y anm: Ekv. med `LEAY -1,Y`	(Y)- $0001 → Y

INC och DEC instruktionerna kan alltså användas för att översätta satser som

```
var = var+1;  vilket är ekvivalent med
var++;
```

eller

```
var = var -1;  vilket är ekvivalent med
var--;
```

under förutsättning att datatypen för `var` är `char`, dvs 8 bitar.

Den ”naiva” översättningen är:

```
LDAB  var
ADDB(SUBB)  #1 alternativt:  INCB(DECB)
STAB  var
```

medan den effektivare översättningen är:

```
INC   var
```

eller

```
DEC var
```

observera dock att detta endast gäller 8-bitars datatyper. För större typer måste den ”naiva” kodningen användas. Observera också att INC och DEC instruktionerna sätter flaggor i CCR på annat sätt än de vanliga ADD och SUB instruktionerna, (C-flaggan påverkas inte). Den utvidgade arkitekturen HCS12X tillför följande dock instruktioner för “increment” och “decrement”

Tabell 1.13: Instruktioner för inkrementering, HCS12X

Mnemonic	Funktion	Operation
INCW	Incrementera i minnet	(M:M+1)+$0001 → (M:M+1)
INCX	Inkrementera X anm: Ekv. med LEAX 1,X	(X)+ $0001 → X
INCY	Inkrementera Y anm: Ekv. med LEAY 1,Y	(Y)+ $0001 → Y

Tabell 1.14: Instruktioner för dekrementering, HCS12X

Mnemonic	Funktion	Operation
DECW	Dekrementera i minnet	(M:M+1)-$0001 → (M:M+1)
DECX	Dekrementera X anm: Ekv. med LEAX -1,X	(X)- $0001 → X
DECY	Dekrementera Y anm: Ekv. med LEAY -1,Y	(Y)- $0001 → Y

Utvidgningen tillför egentligen bara den förenklade kodningen för inkrementering/dekrementering av variabler i minnet. Vi kan med INCW hantera även 16-bitars datatyper.

Booleska logiska operationer (AND,OR,EOR)

AND (*and logical*), OR (*inclusive or logical*) och EOR (*exclusive or logical*) används för att utföra bitvis logiska operationer.

Instruktioner för bitoperationer utförs i HCS12 med 8-bitars operander.

Tabell 1.15: Logiska operationer

Mnemonic	Funktion	Operation
ANDA	Bitvis "och" A med minnesinnehåll	(A)•(M)⇒ A
ANDB	Bitvis "och" A med minnesinnehåll	(B)•(M)⇒ B
ANDCC	Bitvis "och" CC med minnesinnehåll	(CCR)•(M)⇒ CCR
EORA	Bitvis "exklusivt eller" A med minnesinnehåll	(A)⊕(M)⇒ A
EORB	Bitvis "exklusivt eller" B med minnesinnehåll	(B)⊕(M)⇒ B
ORAA	Bitvis "eller" A med minnesinnehåll	(A)+(M)⇒ A
ORAB	Bitvis "eller" B med minnesinnehåll	(B)+(M)⇒ B
ORCC	Bitvis "eller" CCR med minnesinnehåll	(CCR)+(M)⇒ CCR

Exempel 1.25: Logiskt OCH med konstant

Antag deklarationer:

```
char  ca,cb;
```

och uttrycket

```
ca = cb & 0xF;
```

En kodning av uttrycket skulle kunna vara:

```
LDAB  cb
ANDB  #$0F
STAB  ca
```

Om operanderna är 16 bitar med följande deklarationer:

```
int   ia,ib;
```

och uttrycket

```
ia = ib & 0xF004;
```

får vi i stället följande kodning av uttrycket:

```
LDD   ib
ANDB  #$04
ANDA  #$F0
STD   ia
```

Den utvidgade HCS12X medger bitvis logiska operationer även med 16-bitars operander. Något av registren X eller Y måste då användas.

Tabell 1.16: Logiska operationer, HCS12X

Mnemonic	Funktion	Operation
ANDX	Bitvis "och" X med minnesinnehåll	(X)•(M:M+1)⇒ X
ANDY	Bitvis "och" Y med minnesinnehåll	(Y)•(M:M+1)⇒ Y
EORX	Bitvis "exklusivt eller" X med minnesinnehåll	(X) ⊕ (M:M+1)⇒ X
EORY	Bitvis "exklusivt eller" Y med minnesinnehåll	(Y) ⊕ (M:M+1)⇒ Y
ORX	Bitvis "eller" X med minnesinnehåll	(X)+(M:M+1)⇒ X
ORY	Bitvis "eller" Y med minnesinnehåll	(Y)+(M:M+1)⇒ Y

Unära operationer (nollställ, bitvis invertering och 2-komplement)

För unära operatorer - (*unärt minus*) och ~ (*komplementering*) används instruktionerna NEG (*negate*) respektive COM (*logical complement*). CLR (*clear*) används för att *nollställa* operanden. Alla dessa instruktioner har *en* operand.

NEG används fär att bilda 2-komplementet av operanden. Detta betyder alltså att man helt enkelt byter tecken på talet.

Tabell 1.17: Unära operationer

Mnemonic	Funktion	Operation
CLC	Nollställ carryflaggan i CCR	0 ⇒ C
CLI	Nollställ avbrottsmask i CCR	0 ⇒ I
CLR	Nollställ minnesinnehåll	$00 ⇒ M
CLRA	Nollställ A	$00 ⇒ A
CLRB	Nollställ B	$00 ⇒ B
CLV	Nollställ overflowflaggan I CCR	0 ⇒ V
COM	Ettkomplementera minnesinnehåll	$FF-(M) ⇒ M
COMA	Ettkomplementera A	$FF-(A) ⇒ A
COMB	Ettkomplementera B	$FF-(B) ⇒ A
NEG	Tvåkomplementera minnesinnehåll	$00-(M) ⇒ M
NEGA	Tvåkomplementera A	$00-(A) ⇒ A
NEGB	Tvåkomplementera B	$00-(B) ⇒ B

Den utvidgade arkitekturen HCS12X tillför följande instruktioner för unära operationer

Tabell 1.18: Instruktioner för unära operationer, HCS12X

Mnemonic	Funktion	Operation
CLRW	Nollställ minnesinnehåll (16 bitar)	$0000⇒ M:M+1
CLRX	Nollställ X	$0000 ⇒ X
CLRY	Nollställ Y	$0000 ⇒ Y
COMW	Ettkomplementera minnesinnehåll	$FFFF-(M M+1) ⇒ M M+1
COMX	Ettkomplementera X	$FFFF-(X) ⇒ X
COMY	Ettkomplementera Y	$FFFF-(Y) ⇒ Y
NEGW	Tvåkomplementera minnesinnehåll	$0000-(M:M+1) ⇒ M:M+1
NEGX	Tvåkomplementera X	$0000-(X) ⇒ X
NEGY	Tvåkomplementera Y	$0000-(Y) ⇒ Y

Exempel 1.26: Teckenbyte för 8-bitars heltal

Antag deklarationer:

```
char  ca,cb;
```

och uttrycket

```
ca = -cb;
```

En kodning av uttrycket skulle kunna vara:

```
LDAB  cb
NEGB
STAB  ca
```

Exempel 1.27: Teckenbyte för 16-bitars tal

Antag deklarationer:

```
int   ia,ib;
```

och uttrycket

```
ia = -ib;
```

För HCS12 kan uttrycket kodas:

```
LDD   ib
COMA
COMB
ADDD  #1
STD   ia
```

Det kan verka onödigt komplicerat men kommer sig av det faktum att NEG-operationen inte kan utföras med register D (A:B). Om vi i stället använder HCS12X kan vi koda samma uttryck:

```
LDX   ib
NEGX
STX   ia
```

Det logiska komplementet, dvs 1-komplementet, bildas på motsvarande sätt genom att instruktionen `NEG` bytes mot instruktionen `COM`.

Exempel 1.28: Bitvis komplementering

Instruktionsföljden

```
LDAB  #%10101010
COMB
```

ger i register B:

0101 0101

Antag deklarationer:

```
char  ca,cb;
```

och uttrycket

```
ca = ~cb;
```

En kodning av uttrycket skulle kunna vara:

```
LDAB  cb
COMB
STAB  ca
```

Skiftinstruktioner

Skiftoperationer används för att flytta grupper av bitar ett eller flera steg. I programspråket '**C'** finns två olika skiftoperationer:

`A << B`	A skiftas vänster B steg
`A >> B`	A skiftas höger B steg

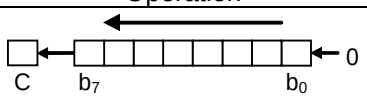
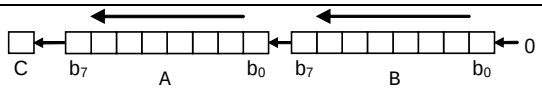
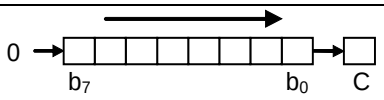
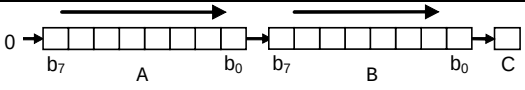
Medan A måste vara en variabel, kan B vara såväl en konstant som någon (annan) variabel. CPU12 stödjer skiftoperationer med tre olika instruktioner:

logiskt skift	(LS)
aritmetiskt skift	(AS)
rotation med Carry	(RO)

Såväl vänsterskift som högerskift kan utföras. Instruktionerna kan användas i någon av dessa former:

1. ***instruktion*** **`<ea>,`** "minnesshift", <ea> anger en minnescell, innehållet i denna skiftas 1 steg. Operandens storlek är här alltid 8 bitar.
2. ***instruktion*****`ACK`** "register-skift", innehållet i ett register (A,B,D) skiftas ett steg.

Tabell 1.19: Logiska skift

Mnemonic	Funktion	Operation
LSL	Logiskt vänsterskift i minnet	
LSLA	Logiskt vänsterskift A	
LSLB	Logiskt vänsterskift B	
LSLD	Logiskt vänsterskift D	
LSR	Logiskt högerskift i minnet	
LSRA	Logiskt högerskift A	
LSRB	Logiskt högerskift B	
LSRD	Logiskt högerskift D	

Exempel 1.29

Antag att följande 'C'-deklarationer är givna:

```
unsigned char uc, ucResult;
```

Koda följande sats i 'C', som assemblerkod:

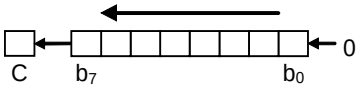
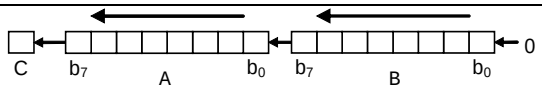
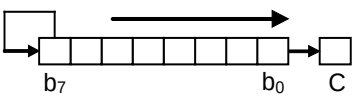
```
ucResult = ( uc >> 1 ) & 1;
```

1 Lösning:

```
LDAB  uc
LSRB
ANDB  #1
STAB  ucResult
```

Observera att logiska högerskift normalt inte används för tal med tecken. Detta beror på att teckeninformationen i så fall riskerar att förloras eftersom en nolla alltid skiftas in från höger till teckenbitens position. För att klara även teckenskift finns därför *aritmetiskt skift*. Observera också att aritmetiskt vänsterskift är det samma som logiskt vänsterskift.

Tabell 1.20: Aritmetiska skift

Mnemonic	Funktion	Operation
ASL	Aritmetiskt vänsterskift i minnet (ekv. med LSL)	
ASLA	Aritmetiskt vänsterskift A (ekv. med LSLA)	
ASLB	Aritmetiskt vänsterskift B (ekv. med LSLB)	
ASLD	Aritmetiskt vänsterskift D (ekv. med LSLD)	
ASR	Aritmetiskt högerskift i minnet	
ASRA	Aritmetiskt högerskift A	
ASRB	Aritmetiskt högerskift B	

Exempel 1.30

Antag att följande 'C'-deklarationer är givna:

```
signed char sc, scResult;
signed int si, siResult;
```

Koda följande satser i 'C', som assemblerkod:

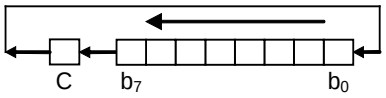
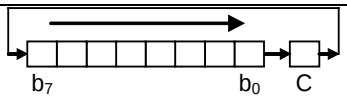
```
scResult = ( sc >> 1 ) & 0x80;
siResult = ( si >> 1 ) & 0x8000;
```

Lösning:

```
LDAB  sc
ASRB
ANDB  #$80
STAB  scResult
LDD   si
ASRA
LSRB
ANDA  #$80
STD   scResult
```

Den tredje typen av skiftinstruktion, *rotate*, kallas också i bland också för "carry-skift".

Tabell 1.21: Carry-skift

Mnemonic	Funktion	Operation
ROL	Rotation vänster via carry i minnet	
ROLA	Rotation vänster via carry A	
ROLB	Rotation vänster via carry B	
ROR	Rotation höger via carry i minnet	
RORA	Rotation höger via carry A	
RORB	Rotation höger via carry B	

Exempel 1.31

Antag att följande 'C'-deklaration är given:

```
signed long sl, slResult;
```

Koda följande C-sats i assemblerkod:

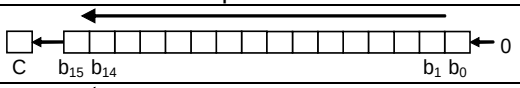
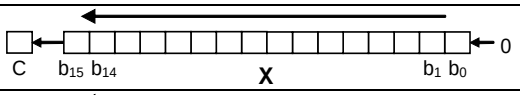
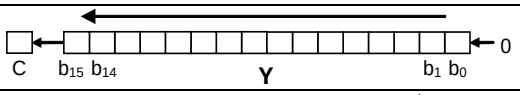
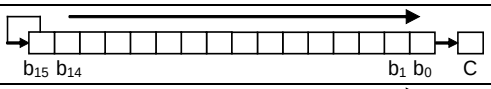
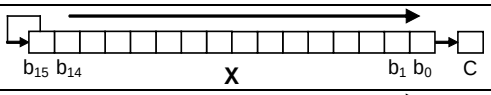
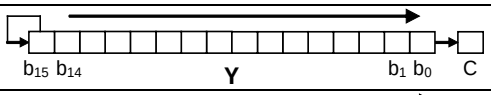
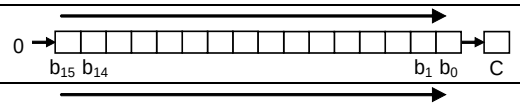
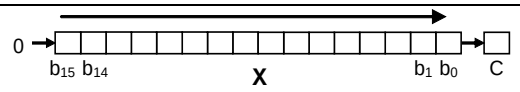
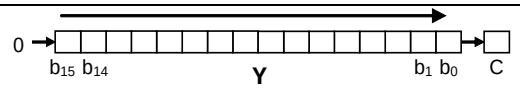
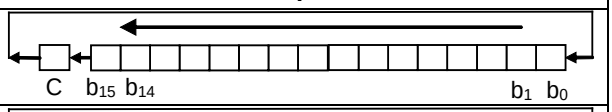
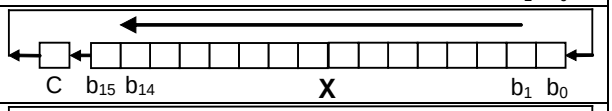
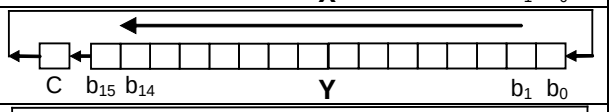
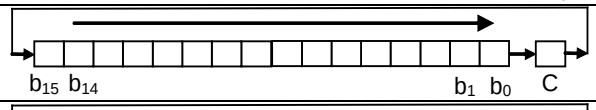
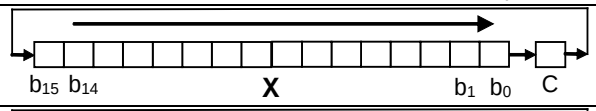
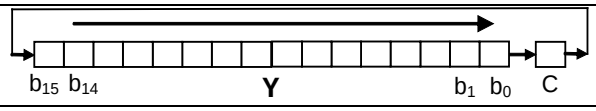
```
slResult  = sl >> 1 ;
```

Lösning:

```
LDD   sl          ; msw av 'sl'
ASRA
RORB
STD   slResult    ; anm. påverkar EJ carry…
LDD   sl+2        ; lsw av 'sl', anm. påverkar EJ carry…
RORA
RORB
STD   slResult +2
```

Ytterligare skiftinstruktioner tillkommer hos HCS12X, se Tabell 1.22 nedan..

Tabell 1.22: Skiftinstruktioner, HCS12X

Mnemonic	Funktion	Operation
ASLW LSLW	Aritmetiskt/Logiskt vänsterskift i minnet, 16 bitar	
ASLX LSLX	Aritmetiskt/Logiskt vänsterskift register X	
ASLY LSLY	Aritmetiskt/Logiskt vänsterskift register Y	
ASRW	Aritmetiskt högerskift i minnet, 16 bitar	
ASRX	Aritmetiskt högerskift register X	
ASRY	Aritmetiskt högerskift register Y	
LSRW	Logiskt högerskift i minnet, 16 bitar	
LSRX	Logiskt högerskift register X	
LSRY	Logiskt högerskift register Y	
ROLW	Carry-skift vänster i minnet, 16 bitar	
ROLX	Carry-skift vänster register X	
RORY	Carry-skift vänster register Y	
RORW	Carry-skift höger i minnet, 16 bitar	
RORX	Carry-skift höger register X	
RORY	Carry-skift vänster register Y	

Exempel 1.32

Exempel 1.31 ovan kan, för HCS12X, kodas:

```
        LDX   sl          ; msw av 'sl'
        LDY   sl+2        ; lsw av 'sl'
        ASRX
        RORY
        STX   slResult
        STY   slResult+2
```

Jämförelse och test

Jämförelseinstruktioner, ”compare”, liknar subtraktionsinstruktioner med skillnaden att resultatet inte skrivs tillbaks till destinationsoperanden. Endast flaggorna (N,Z,V och C) i CCR påverkas alltså av dessa instruktioner.

För att testa *en* operand används TST (*test an operand*). Resultatet av testen sätter flaggorna N och Z i CCR medan flaggorna V och C *alltid* nollställs av TST-instruktionen. Detta innebär att instruktionerna `TSTA` och `CMPA #0` inte är ekvivalenta.

Tabell 1.23: Jämförelseinstruktioner

Mnemonic	Funktion	Operation
CBA	Jämför B med A	(A)-(B)
CMPA	Jämför A med minne	(A)-(M)
CMPB	Jämför B med minne	(B)-(M)
CPD	Jämför D med minne	(A:B)-(M:M+1)
CPS	Jämför SP med minne	(SP)-(M:M+1)
CPX	Jämför X med minne	(X)-(M:M+1)
CPY	Jämför Y med minne	(Y)-(M:M+1)

Tabell 1.24: Testinstruktioner

Mnemonic	Funktion	Operation
TST	Testa minnesinnehåll	(M)-$00
TSTA	Testa register A	(A)-$00
TSTB	Testa register B	(B)-$00

Tabell 1.25: Jämförelseinstruktioner, HCS12X

Mnemonic	Funktion	Operation
CPED	Jämför D med minne och Borrow	(A:B)-((M:M+1)+C)
CPES	Jämför SP med minne och Borrow	(SP)-((M:M+1)+C)
CPEX	Jämför X med minne och Borrow	(X)-((M:M+1)+C)
CPEY	Jämför Y med minne och Borrow	(Y)-((M:M+1)+C)

Tabell 1.26: Testinstruktioner, HCS12X

Mnemonic	Funktion	Operation
TSTW	Testa minnesinnehåll, 16 bitar	(M M+1)-$0000
TSTX	Testa register X	(X)-$0000
TSTY	Testa register Y	(Y)-$0000

Vi exemplifierar jämförelse och test instruktioner i samband med villkorliga instruktioner för programflödeskontroll nedan.

Ovillkorlig programflödeskontroll

Instruktioner som JMP (*jump*) BRA (*branch always*) JSR (*jump to subroutine*) och BSR (*branch to subroutine*) används för att åstadkomma programflödesändringar oavsett flaggsättningen i **CCR**. Instruktionerna JSR och BSR har dessutom egenskapen att adressen till nästa instruktion lagras undan på stacken innan hoppet utförs. Dessa instruktioner kan användas tillsammans med instruktionen RTS, (*return from subroutine*) för att skapa subrutiner av programkod som skall utföras många gånger eller kanske delas av olika program.

Tabell 1.27: Ovillkorlig programflödeskontroll

Mnemonic	Funktion	Operation
BSR	Anrop av subrutin. PC-relativ operand	SP-2 ⇒ SP RetAdrL:RetAdrH ⇒ $M_{(SP)}$:$M_{(SP+1)}$ Adress ⇒ PC
BRA	"Hopp" till adress. PC-relativ operand	Adress ⇒ PC
CALL	Anrop av subrutin Absolut operand (20 bitar) Anm: Användes vid programflödesändring mellan olika minnesbankar ($8000-$BFFF)	SP-2 ⇒ SP RetAdrL:RetAdrH ⇒ $M_{(SP)}$:$M_{(SP+1)}$ Subrutinadress ⇒ PC SP-1 ⇒ SP (PPAGE) ⇒ $M_{(SP)}$ PAGE ⇒ PPAGE Subrutinadress ⇒ PC
JMP	"Hopp" till address. Absolut operand	Subrutinadress ⇒ PC
JSR	Anrop av subrutin Absolut operand	SP-2 ⇒ SP RetAdrL:RetAdrH ⇒ $M_{(SP)}$:$M_{(SP+1)}$ Subrutinadress ⇒ PC
RTC	Återvänd från subrutin. Returadress från STACK och PPAGE	$M_{(SP)}$ ⇒ (PPAGE) SP+1 ⇒ SP $M_{(SP)}$:$M_{(SP+1)}$ ⇒ PC_H:PC_L SP+2 ⇒ SP
RTS	Återvänd från subrutin. Returadress från STACK	$M_{(SP)}$:$M_{(SP+1)}$ ⇒ PC_H:PC_L SP+2 ⇒ SP

Att modularisera ett program innebär att dess olika funktioner placeras i block med överskådlig och sammanhållen programkod. Ett naturligt sätt att modularisera är att dela upp programkoden i subrutiner. En subrutin kännetecknas av sitt gränssnitt och gränssnittet utgörs av:

- Namn
- Indata
- Utdata

Om dessa egenskaper dokumenterats väl är det möjligt att använda subrutinen utan att samtidigt tvingas sätta sig in i den detaljerade funktionen. Det är därför viktigt att denna dokumentation finns med och att den utformats på rätt sätt.

Exempel 1.33 "Header" för subrutin

En text beskrivande subrutinen "COMMAND" kan lämpligen utformas på följande sätt:

```
*
* SUBRUTIN – COMMAND
* Beskrivning: Rutinen avgör vilken
* kommandosubrutin som skall
* utföras och anropar denna.
*
* Indata:              Kommandonummer i reg A
* Utdata:              Inga
*
* Registerpåverkan:    A,X
*
*
*
* Anropade subutiner:  SUB0, SUB1, SUB2
*
*                      LDAA    #cmd
* Anrop:               JSR     COMMAND
*
```

Namnet åtföljs av en kort beskrivning av subrutinens funktion.

Eventuella indata/utdata anges, dessutom vilka register som används för indata/utdata.

De registerinnehåll som ändras av subrutinen anges speciellt.

Ytterligare subrutiner som anropas från den dokumenterade rutinen bör anges.

Exempel på hur subrutinen anropas bör anges.

CPU12 stöder modularisering bland annat med instruktionerna BSR (*branch to subroutine*) och RTS (*return from subroutine*). BSR kan ses som en ovillkorlig hopp-instruktion, men till skillnad från exempelvis BRA, gäller att exekveringen ska fortsätta direkt efter BSR då subrutinen utförts. Följakligen måste *returadressen*, dvs adressen till instruktionen omedelbart efter BSR sparas på något sätt. På motsvarande sätt måste adressen kunna placeras i **PC** då instruktionen RTS exekveras. Stacken används för att spara returadresser, dvs vid:

BSR placeras adressen till nästa instruktion på stacken, stackpekaren minskas med 2 bytes, adressen till subrutinen placeras i **PC**

och vid:

RTS 2 bytes tas från stacken och placeras i **PC** stackpekaren ökas med 4 bytes

Som framgår av Tabell 1.27 ovan finns det olika varianter av instruktioner som används för subrutinanrop. De skiljer sig åt i kodningen (se instruktionslistan) och man väljer variant beroende på avståndet mellan anropet och den anropade subrutinen.

Exempel 1.34 Användning av BSR

Instruktionen BSR kodas:

`07 rr` där `rr` står för 8 bitars PC-relativ offset (med tecken)

Den anropade subrutinen kan därför finnas på någon adress maximalt 127 bytes framåt eller 128 bytes bakåt räknat från adressen till BSR-instruktionen+2.

Villkorlig programflödeskontroll

Instruktioner för kontroll av programflödet är centrala i varje mikroprocessors instruktionsuppsättning. Beteckningen ”branch” vilken kan översättas med ”förgrening” är gemensam för instruktioner för ”vägval” i programutförandet. För en villkorlig ”branch”-instruktion är vägvalet självfallet associerat till något villkor, ”condition” varför den den gemensamma beteckningen för dessa instruktioner är ”branch on condition”, dvs en villkorlig ändring av programmets flöde. Detta åstadkommes genom att något villkor testas, om villkoret är sant utförs en programflödesändring, dvs ett ”hopp” i programmet. Om villkoret däremot är falskt, fortsätter programutförandet genom att nästa instruktion i minnet exekveras.

Villkorliga instruktioner används alltså alltid tillsammans med någon (omedelbart föregående) instruktion som åstadkommit flaggsättning. Ofta är detta någon av instruktionerna `cmp` eller `tst` men det kan också vara i kombination med någon aritmetisk instruktion. Det är därför viktigt att du alltid kontrollerar hur flaggorna sätts av instruktionen som föregår branch-instruktionen.

En villkorlig (branch-) instruktion:

Testar villkoret mot innehållet i flaggregistret (**CCR**)
Om resultatet av testen är SANT, utförs instruktionen (hoppet)
Om resultatet är FALSKT fortsätter exekveringen med nästa instruktion (hoppet utförs inte).

16 olika villkor (hoppinstruktioner) kan anges och vi ska här titta närmare på dessa villkor och i vilka sammanhang de kan användas

Tabell 1.28: Villkorlig programflödeskontroll

Mnemonic	Funktion	Villkor
Enkla flaggtest		
BCS	"Hopp" om *carry*	C=1
BCC	"Hopp" om ICKE *carry*	C=0
BEQ	"Hopp" om *zero*	Z=1
BNE	"Hopp" om ICKE *zero*	Z=0
BMI	"Hopp" om *negative*	N=1
BPL	"Hopp" om ICKE *negative*	N=0
BVS	"Hopp" om *overflow*	V=1
BVC	"Hopp" om ICKE *overflow*	V=0
Test av tal utan tecken		
BHI	Villkor: R>M	C + Z = 0
BHS	Villkor: R≥M	C=0
BLO	Villkor: R<M	C=1
BLS	Villkor: R≤M	C + Z = 1
Test av tal med tecken		
BGT	Villkor: R>M	Z + (N ⊕ V) = 0
BGE	Villkor: R≥M	N ⊕ V = 0
BLT	Villkor: R<M	N ⊕ V = 1
BLE	Villkor: R≤M	Z + (N ⊕ V) = 1

Observera speciellt de villkorliga instruktionerna för realtionera "större än", större eller lika", etc. Här måste rätt instruktion väljas baserat på om vi jämför tal betraktade som *med* eller *utan* tecken.

Exempel 1.36 Jämförelser tal med/utan tecken

Vi jämför instruktionerna `BHI` (*branch higher*) och `BGT` (*branch greater than*). Antag att vi vill testa om innehållet i register B är *större* än -2, vi prövar med kodsekvensen

```
      CMPB      #-2
      BHI       St_n2
      ...       denna kod ska utföras om innehållet i B är mindre än -2
St_n2:
      ...       denna kod ska utföras om innehållet i B är större än, eller lika med -2
```

Vi antar att B's innehåll är 3, vilket bör resultera i att instruktionen (hoppet) *utförs* ty 3 > -2. Vi utför nu operation och flaggsättning:

```
            10 10 10 10 10 10 10
  (3)        0  0  0  0  0  0  1  1
-(-2)        1  1  1  1  1  1  1  0
=            0  0  0  0  0  1  0  1
```

Observera speciellt att en *Borrow* genereras vid operationen

I processorn utförs dock subtraktionen som *addition* av tvåkomplementet, dvs:

```
  (3)       0000 0011
 +(2)       0000 0010
=          0 0000 0101
```

Carry-biten blir 0 när vi räknar, Carry flaggan sätts dock till inversen av detta ty vid SUB och CMP-instruktioner representerar Carry-flaggan *Borrow*...

Flaggsättning:
Minns att C-flaggan vid denna operation representerar " Borrow"...

C = 1 ty borrow genererades
Z = 0 ty resultatet är skilt från 0
V = 0 ty inget tvåkomplementsspill genererades
N = 0 ty mest signifikanta biten är 0

I vårt exempel förväntar vi oss att hoppet till St_n2 ska utföras, men detta är inte fallet eftersom villkoret för `BHI` *ej* är uppfyllt, $(\overline{C} \wedge \overline{Z})$.

Om vi istället hade valt instruktionen `BGT,` ($N \wedge V \wedge \overline{Z} \vee \overline{N} \wedge \overline{V} \wedge \overline{Z}$), fungerar det korrekt (jämför flaggsättningen med villkoret för `BGT`). Förklaringen ligger i att vi betraktar jämförelse av tal *med* tecken.

Vi kan sammanfatta de villkorliga testoperationerna genom att koppla dessa till motsvarande *operatorer* i programspråket **C**.

Tabell 1.29: C-operatorer och villkorlig programflödeskontroll

C-operator	Betydelse	Datatyp	Instruktion
`==`	Lika med	signed/unsigned	`BEQ`
`!=`	Skild från	signed/unsigned	`BNE`
`<`	Mindre än	signed	`BLT`
		unsigned	`BCS`
`<=`	Mindre än eller lika	signed	`BLE`
		unsigned	`BLS`
`>`	Större än	signed	`BGT`
		unsigned	`BHI`
`>=`	Större än eller lika	signed	`BGE`
		unsigned	`BCC`

Av tabellen framgår hur operator *tillsammans* med den aktuella datatypen avgör villken villkorlig instruktion som ska användas.

Exempel 1.37 Kodning av jämförelseoperation

Antag att följande 'C'-deklaration är given:

```
unsigned char uc1,uc2;
```

Koda följande C-sats i assemblerkod:

```
if( uc1 > uc2 )
      S1;
```

Lösning:

Vi väljer villkorlig instruktion `BHI` eftersom denna svarar mot operator och datatyp:

```
      LDAB  uc1
      CMPB  uc2
      BHI   S1
      BRA   skip
S1:   ...
      ...
skip:
```

Observera att vi kan avlägsna `BRA-` instruktionen genom att välja komplementinstruktionen till `BHI`, dvs `BLS`:

```
      LDAB  uc1
      CMPB  uc2
      BLS   skip
S1:   ...
      ...
skip:
```

Bittest och villkorlig programflödesändring

Tabell 1.30: Villkorlig programflödeskontroll baserad på bittest

Mnemonic	Funktion	Villkor
BRCLR	"Hopp" om bit-operands alla bitar är 0	(M)•(mm) = 0
BRSET	"Hopp" om bit-operands alla bitar är 1	($\bar{M}$)•(mm) = 0

Dessa instruktioner kombinerar test och villkorlig programflödesändring. Speciellt är de användbara i programslingor med statustest av IO-enheter.

Exempel 1.38

Antag att en IO-enhet placerats på address $400 i datorn's mine. Antag vidare att bit 0 indikerar någon speciell händelse. Om bit 0 är 1 innebär detta att händelsen inträffat och att vänteslingan kan avslutas. Följande konstruktion implementerar då denna vänteslinga:

```
wait_here:
      BRCLR $0400,#%00000001,wait_here
```

BRCLR testar om data på address $400 efter en logisk AND-operation med %00000001 är noll. I så fall utförs "hopp" till positionen "wait_here", annars fortsätter programutförandet med efterföljande instruktion.

Instruktioner för räknande programslingor

Villkorliga instruktioner används för att styra programflödet baserat på en tidigare jämförelse. Vissa typer av programkonstruktioner är mycket vanliga och CPU12 omfattar därför en speciell uppsättning instruktioner utformade enbart för att stödja sådana programkonstruktioner.

Tabell 1.31: Instruktioner för räknande programslingor

Mnemonic	Funktion	Villkor
DBEQ	Dekrementera innehåll i register. "Hoppa" om resultatet = 0. (register: A,B,D,X,Y,SP)	(register) – 1 ⇒ register om(register)=0; "hoppa"; annars: nästa instruktion
DBNE	Dekrementera innehåll i register. "Hoppa" om resultatet ≠ 0. (register: A,B,D,X,Y,SP)	(register) – 1 ⇒ register om(register)≠0; "hoppa"; annars: nästa instruktion
IBEQ	Inkrementera innehåll i register. "Hoppa" om resultatet = 0. (register: A,B,D,X,Y,SP)	(register) + 1 ⇒ register om(register)=0; "hoppa"; annars: nästa instruktion
IBNE	Inkrementera innehåll i register. "Hoppa" om resultatet ≠ 0. (register: A,B,D,X,Y,SP)	(register) + 1 ⇒ register om(register)≠0; "hoppa"; annars: nästa instruktion
TBEQ	Testa innehåll i register. "Hoppa" om resultatet = 0. (register: A,B,D,X,Y,SP)	om(register)=0; "hoppa"; annars: nästa instruktion
TBNE	Testa innehåll i register. "Hoppa" om resultatet ≠ 0. (register: A,B,D,X,Y,SP)	om(register)≠0; "hoppa"; annars: nästa instruktion

Exempel 1.39 Kodning av "for-loop"

Antag deklarationen:

```
      int   i;
```

Koda C-programsekvensen:

```
      for( i = 10; i > 0; i-- )
      {
            ...
      }
dummy:
```

Lösning:

```
      MOVW  #10,i
      LDD   i
for_loop_iteration:
      DBEQ  D,dummy
```

```
      ...
      BRA   for_loop_iteration
dummy:
```

Vi inser igen att en och samma programkonstruktion kan kodas i assembler på flera olika sätt. Försök själv att koda ovanstående exempel utan att använda sammansatta instruktioner som dessa.

Ytterligare en vanlig programkonstruktion ”while(...) do” implementeras också enkelt, betrakta följande exempel:

Exempel 1.40 Kodning av ”while-loop”

Antag deklarationen:

```
      int   i;
```

Koda C-programsekvensen:

```
      while( i )
      {
            ...
      }
dummy:
```

Lösning:

```
      LDD   i
while_loop_iteration:
      TBEQ  D,dummy
      ...
      BRA   while _loop_iteration
dummy:
```

Multiplikation och division

Eftersom multiplikation utförs på olika sätt beroende på om operanderna betraktas som tal med eller utan tecken, finns det också flera varianter av multiplikations-instruktionen:

Tabell 1.32 Instruktioner för multiplikation

Mnemonic	Funktion	Operation
MUL	Multiplikation, utan tecken (8×8 bitar)	(A)×(B) ⇒ A:B
EMUL	Multiplikation, utan tecken (16×16 bitar)	(D)×(Y) ⇒ Y:D
EMULS	Multiplikation, med tecken (16×16 bitar)	(D)×(Y) ⇒ Y:D

Användning av instruktioner för multiplikation		
	Tal med tecken (signed)	Tal utan tecken (unsigned)
8 bitar	Ingen instruktion finns för detta. I stället används MUL kompletterad med s.k. ”teckenöverläggning”, se exempel nedan.	MUL (A)×(B)→(D) Innehållen i ackumulatorerna multipliceras. Resultatet, 16 bitar, finns efter instruktionen i ackumulator D.
16 bitar	EMULS (D)×(Y)→(Y:D) Innehållen i D och Y multipliceras. Resultatet, 32 bitar, finns efter instruktionen i register Y (mest sign. 16 bitar) och i ackumulator D (minst signifikanta 16 bitar)	EMUL Samma som EMULS men vid multiplikationen behandlas talen utan tecken.

Exempel 1.41 Kodning av ”unsigned character”-multiplikation

Antag att följande ’C’-deklarationer är givna:

```
unsigned char uc1, uc2, ucResult;
```

Koda följande operation, given i ’C’, i assemblerspråk:

```
ucResult = uc1 * uc2;
```

Lösning:

```
; ucResult = uc1 * uc2;
      LDAA uc1
      LDAB uc2
      MUL
      STAB ucResult
```

Exempel 1.42 Kodning av ”short”-multiplikation

Antag att följande ’C’-deklarationer är givna:

```
unsigned short us1, us2, usResult;
signed short ss1, ss2, ssResult;
```

Koda följande operationer, givna i ’C’, i assemblerspråk:

```
usResult = us1 * us2;
ssResult = ss1 * ss2;
```

Lösning:

```
; usResult = us1 * us2;
      LDD   us1
      LDY   us2
      EMUL
      STD   usResult

; ssResult = ss1 * ss2;
      LDD   ss1
      LDY   ss2
      EMULS
      STD   ssResult
```

Tabell 1.33 Instruktioner för division

Mnemonic	Funktion	Operation
IDIV	Division, utan tecken (16/16 bitar)	(D)÷(X) ⇒ X Resten ⇒ D
IDIVS	Division, med tecken (16/16 bitar)	(D)÷(X) ⇒ X Resten ⇒ D
FDIV	Bråkdelsdivision (16/16 bitar)	(D)÷(X) ⇒ X Resten ⇒ D
EDIV	Division, utan tecken (32/16 bitar)	(Y:D)÷(X) ⇒ Y Resten ⇒ D
EDIVS	Division, med tecken (32/16 bitar)	(Y:D)÷(X) ⇒ Y Resten ⇒ D

Stackoperationer

Operationer som påverkar stackpekaren (SP) kallar vi *stackoperationer*. Stacken används för att tillfälligt spara data och adresser. Det finns instruktioner som implicit använder stacken, exempelvis JSR, RTS vilka beskrivits tidigare nedan. I dessa fall hanteras stacken, undanlagring och återställning, utan att programmeraren behöver vidta speciella åtgärder. Ur programmerarens

synvinkel är stacken däremot intressant som lagringsplats för ”temporära data”, dvs ett sätt att reservera minnesutrymme för en kortare instruktionssekvens. Då sekvensen är avslutad kan stacken återställas och samma minnesutrymme kan på så sätt återanvändas av senare instruktionssekvenser som kräver plats för temporär undanlagring/återställning.

Stackoperationer kan indelas i två grupper

- Operationer som enbart påverkar stackpekaren.
- Operationer som implicit använder (och eventuellt påverkar) stackpekaren.

Flertalet operationer som inbegriper stackpekaren har vi behandlat i tidigare sammanhang,

Mnemonic	Funktion	Operation
PSHA	Placera innehållet i register A på stacken	(SP)-1 ⇒ SP, (A) ⇒ $M_{(SP)}$
PSHB	Placera innehållet i register B på stacken	(SP)-1 ⇒ SP, (B) ⇒ $M_{(SP)}$
PSHC	Placera innehållet i register CCR på stacken	(SP)-1 ⇒ SP, (CCR) ⇒ $M_{(SP)}$
PSHD	Placera innehållet i register D på stacken	(SP)-2 ⇒ SP, (A: B) ⇒ $M_{(SP)}$:$M_{(SP+1)}$
PSHX	Placera innehållet i register X på stacken	(SP)-2 ⇒ SP, (X) ⇒ $M_{(SP)}$:$M_{(SP+1)}$
PSHY	Placera innehållet i register Y på stacken	(SP)-2 ⇒ SP, (Y) ⇒ $M_{(SP)}$:$M_{(SP+1)}$
PULA	Återställ innehållet i register A från stacken	($M_{(SP)}$) ⇒ A, (SP)+1 ⇒ SP
PULB	Återställ innehållet i register B från stacken	($M_{(SP)}$) ⇒ <B, (SP)+1 ⇒ SP
PULC	Återställ innehållet i register CCR från stacken	($M_{(SP)}$) ⇒ CCR, (SP)+1 ⇒ SP
PULD	Återställ innehållet i register D från stacken	($M_{(SP)}$:$M_{(SP+1)}$) ⇒ A:B, (SP)+2 ⇒ SP
PULX	Återställ innehållet i register X från stacken	($M_{(SP)}$:$M_{(SP+1)}$) ⇒ X, (SP)+2 ⇒ SP
PULY	Återställ innehållet i register Y från stacken	($M_{(SP)}$:$M_{(SP+1)}$) ⇒ Y, (SP)+2 ⇒ SP

För HCS12X tillkommer följande stackoperationer:

Mnemonic	Funktion	Operation
PSHCW	Placera innehållet i register CCR_H:CCR på stacken.	(SP)-2 ⇒ SP, (CCR_H: CCR) ⇒ $M_{(SP)}$:$M_{(SP+1)}$
PULCW	Återställ innehållet i register CCR_H:CCR från stacken.	($M_{(SP)}$:$M_{(SP+1)}$)⇒ CCR_H: CCR, (SP)+2 ⇒ SP

Vi har minst tre olika situationer som kräver temporär lagring med användning av stacken. Det första fallet beskrivs enklast av att vi behöver använda någon speciell instruktion som kräver ett specifikt register och att detta register för tillfället används för att lagra ett tidigare delresultat.

Situationen kallas”register spill”, och kan enkelt hanteras med hjälp av en tillfällig undanlagring på stacken.

Exempel 1.43 Hantering av ”register spill”.

Låt föjande deklarationer vara givna:

```
unsigned short int _a,_b,_c,_d;
```

Skriv en sekvens assemblerinstruktioner som evaluerar följande uttryck och lämnar resultatet i register D.

```
(_a*_b)+(_c*_d);
```

Lösning:

För 16 bitars multiplikation använder vi EMUL-instruktionen. Denna förutsäller att operanderna finns i D respektive Y-registren.

```
LDD   _a
LDY   _b
EMUL        ; första parentesen evaluerad
PSHD        ; placera delresultat på stacken
LDD   _c
LDY   _d
EMUL        ; andra parentesen evaluerad
ADDD  0,SP  ; addera med första delresultatet
LEAS  2,SP  ; återställ stackpekaren
```

Efter instruktionssekvensen finns hela uttryckets värde i register D, stackpekaren har återställts till det värde den hade före instruktionssekvensen.

Ett annat fall är då vi vill reservera ”tillfällig plats” eller mera konkret, ”deklarera lokala variabler”. Sådana är synliga endast i den funktion (subrutin) de deklarerats och används aldrig före eller efter funktionen anropats. Det är därför en god tanke att använda temporär lagring för dessa. Minnet kan ju då återanvändas av en annan funktion. Vi återkommer till detta.

Exempel 1.44 Användning av ”tillfälligt variabelutrymme”

Antag att följande ’C’-deklarationer är givna:

```
signed char sc1, sc2, scResult;
```

Koda följande operation, given i ’C’, i assemblerspråk:

```
scResult = sc1 * sc2;
```

Lösning:

Eftersom det inte finns någon maskininstruktion för multiplikation av 8-bitars tal med tecken tvingas vi här göra teckenöverläggning för att bestämma tecken hos resultatet. Vi inför en temporär variabel `sign` för att hålla reda på resultatets tecken. Så här skulle då operationen kunna kodas i ’C’:

```
sign = 0;
if( (sc1 & 0x80 )&&( sc2 & 0x80 ) ) /* båda är < 0 */
      scResult = (-sc1) * (-sc2);
else if( (!(sc1 & 0x80 )) && (!( sc1 & 0x80 )) ) /* båda är => 0 */
      scResult = sc1 * sc2;
else if( (sc1 & 0x80 ) && (!( sc2 & 0x80 )) ){ /* sc1 < 0, sc2 => 0 */
      scResult = -sc1 * sc2;
      sign=1;
}
else { /* sc1 => 0, sc2 < 0 */
      scResult = sc1 * -sc2;
      sign = 1;
}

if( sign == 1 )
```

```
            scResult = - scResult;
```

Låt oss nu se hur vi kan koda detta som assemblerkod:

```
      LEAS  -1,SP       ; unsigned char   sign;
      CLR   0,SP        ; sign = 0;

; if( (sc1 & 0x80 )&&( sc2 & 0x80 ) ) /* båda är < 0 */
      LDAA  _sc1
      BPL   _2
      LDAB  _sc2
      BPL   _2
;     scResult = (-sc1) * (-sc2);
      NEGA
      NEGB
      BRA   _8
_2:
; else if( (!(sc1 & 0x80 )) && (!( sc1 & 0x80 )) ) /* båda är => 0 */
      TSTA
      BMI   _4
      TSTB
      BMI   _4
; scResult = sc1 * sc2;
      BRA   _8
_4:
; else if( (sc1 & 0x80 ) && (!( sc2 & 0x80 )) ){ /* sc1 < 0, sc2 => 0 */
      TSTA
      BPL   _6
      TSTB
      BMI   _6
; scResult = -sc1 * sc2;
      NEGA
      MOVB  #1,0,SP     ; sign=1;
      BRA   _8
_6:
; else { /* sc1 => 0, sc2 < 0 */
; scResult = sc1 * -sc2;
      NEGB
      MOVB  #1,0,SP     ; sign = 1;
_8:
      MUL
      STAB  _scResult

; if( sign == 1 )
      LDAA  0,SP
      BEQ   _9
; scResult = - scResult;
      NEGB
      STAB  _scResult
_9:
      LEAS  1,SP
```

En tredje situation är då värden ska överföras till och från en funktion. Beroende på antalet värden som ska överföras kan här stacken komma till användning. Vi återkommer även till detta nedan.

Om man programmerar i ett högnivåspråk, behöver man normalt inte bekymra sig för ”temporär lagring” eftersom kompilatorn då hanterar kodgenereringen. Om man däremot kodar i assemblerspråk, och kombinerar detta med kompilatorgenererad kod, är det ytterligt viktigt att man förstått kompilatorns konventioner i dessa sammanhang.

Table 5-26. Condition Code Instructions

Mnemonic	Function	Operation
ANDCC	Logical AND CCR with memory	(CCR) • (M) ⇒ CCR
CLC	Clear C bit	0 ⇒ C
CLI	Clear I bit	0 ⇒ I
CLV	Clear V bit	0 ⇒ V
ORCC	Logical OR CCR with memory	(CCR) + (M) ⇒ CCR
PSHC	Push CCR onto stack	(SP) – 1⇒ SP; CCR ⇒ $M_{(SP)}$
PSHCW	Push CCR_H:CCR onto stack	(SP) – 2 ⇒ SP; (CCR_H:CCR) ⇒ $M_{(SP)}$:$M_{(SP+1)}$
PULC	Pull CCR from stack	($M_{(SP)}$) ⇒ CCR; (SP) + 1 ⇒ SP
PULCW	Pull CCR_H:CCR from stack	($M_{(SP)}$:$M_{(SP+1)}$) ⇒ CCR_H:CCR; (SP) + 2⇒ SP
SEC	Set C bit	1 ⇒ C
SEI	Set I bit	1 ⇒ I
SEV	Set V bit	1 ⇒ V
TAP	Transfer A to CCR	(A) ⇒ CCR
TPA	Transfer CCR to A	(CCR) ⇒ A

Parameteröverföring

Vi har redan sett exempel på hur *parametrar* kan överföras till/från en subrutin med hjälp av processorns register. Det är ett enkelt och mycket effektivt sätt att överföra parametrar. Vi kan också införa *konventioner* alltså regler för hur parameterlistorna ska översättas, dvs utgående från ordningsföljden av parametrar tilldelas register efter ett förutbestämt mönster.

Exempel Parametrar i register

Antag deklarationer:

```
int   la,lb,lc;
```

Antag vidare att vi alltid använder register `D`, `X`, `Y` (i denna ordning) för parametrar som skickas till en subrutin. Då kan funktionsanropet

```
dummyfunc(la,lb,lc);
```

översättas till:

```
LDD   la
LDX   lb
LDY   lc
BSR   dummyfunc
```

Då vi kodar subrutinen `dummyfunc` vet vi (på grund av våra regler) att den första parametern skickas i D, den andra i X och den tredje i Y (osv).

Det visade exemplet indikerar dock en rad problem för mer generella fall. Exempelvis kan parameterlistor vara långa och hur gör vi om inte processorns register räcker till? För att hantera det generella fallen kommer de följande avsnitten kommer också att behandla några andra metoder för parameteröverföring till och från subrutiner. I avsnitten om *kombinerad programmering* dvs hur man konstruerar program i *såväl* assemblerkod som ett högnivåspråk, återkommer vi till detta och visar metoder som vanligtvis används av C-kompilatorer.

Stacken används för temporär lagring

"Stacken" dvs en minnesarea som upplåtits för tillfällig mellanlagring kan användas för att *spara* registerinnehåll. Registren kan därefter användas för såväl uttrycksevaluering som parametrar, stacken *återställes* därefter och de ursprungliga registerinnehållen återställes samtidigt.

Då vi sparar ett registerinnehåll på stacken (detta kallas av tradition "push") kan vi föreställa oss att vi lägger detta, överst, på en "hög" (eng. stack), då vi återställer (kallas av tradition "pop") innebär detta att vi tar, det som ligger överst, på "högen". Jämför detta med hur CPU12 placerar *returadressen* vid subrutinanrop, på stacken där register **SP** utgör stackpekare. Av resonemanget framgår vikten av att vi *lägger på* och *plockar av* stacken i rätt ordning.

CPU12 stödjer stackhantering med adresseringsmoderna

preautodecrement `n,-SP`

och

postautoincrement `n,SP+`

Exempel

Vi vill spara initiala värden i register D och X på stacken för att kunna använda dessa värden i upprepade beräkningar:

```
STD   2,-SP
STX   2,-SP
```

Antag att stackpekarens innehåll före instruktionerna är $3000. Stackens utseende *efter* instruktionerna blir då:

Adress	Innehåll	SP före	SP efter
3000		◄	
2FFF	D.lsb		
2FFE	D.msb		
2FFD	X.lsb		
2FFC	X.msb		◄
2FFB			

Antag vidare att D och X nu används för beräkningar och att vi därefter åter igen behöver de initiala värdena. Vi ser att SP nu pekar på den sist lagrade byten. För att återföra de ursprungliga värdena till D och X utan att påverka stacken kan vi nu använda:

```
LDD   2,SP
LDX   0,SP
```

Då beräkningarna är slutförda kan vi välja mellan att antingen återställa såväl de initiala värdena till D och X samt stackpekaren med:

```
LDX   2,SP+
LDD   2,SP+
```

eller, om vi inte vill modifiera D,X utan bara återställa stackpekaren:

```
LEAS  4,SP
```

Parametrar överförda via stacken

Det mest generella sättet att överföra parametrar är via stacken. Metoden har fördelen att *antalet* parametrar inte är beroende av antalet register i processorn. Ett subrutinanrop föregås då av ett antal instruktioner som placerar parametrarna på stacken. Efter subrutinanropet måste stacken återställas. I subrutinen refereras parametrarna via den offset de får i förhållande till stackpekaren.

Exempel Parametrar via stack

Antag deklarationer:

```
int   la,lb,lc;
```

Antag vidare att listan av parametrar som skickas till en subrutin behandlas från höger till vänster. Då kan funktionsanropet

```
dummyfunc(la,lb,lc);
```

översättas till:

```
LDD   lc
PSHD        (alternativt STD  2,-SP)
LDD   lb
PSHD
LDD   la
PSHD
BSR   dummyfunc
LEAS  6,SP
```

Då vi kodar subrutinen `dummyfunc` vet vi nu dess parametrar är åtkomliga via stacken enligt följande.

Innehåll	Kommentar	Adressering via SP i subrutinen
`lc.lsb`	Parameter `lc`	
`lc.msb`		`6,SP`
`lb.lsb`	Parameter `lb`	
`lb.msb`		`4,SP`
`la.lsb`	Parameter `la`	
`la.msb`		`2,SP`
`PC.lsb`	Återhoppsadress,	
`PC.msb`	placeras här vid `BSR`	`0,SP`

Parametrarna kan nu refereras enligt följande:

```
dummyfunc:
      . .
      LDD   2,SP  parameter la till register D
      . .
      LDD   4,SP  parameter lb till register D
      . .
      LDD   6,SP  parameter lc till register D
```

Parametrar i programkod (In Line)

Ett annat sätt att överföra parametrar är *direkt i koden*. Metoden förutsätter då att parametrarna är konstanta. Metoden är ovanlig men förekommer exempelvis vid implementering av så kallade ”systemanrop”.

Exempel

“In line” parameteröverföring, värdet 10 ska överföras till en subrutin:

```
      BSR   dummyfunc
      FCB   10
      NOP
```

I `dummyfunc` måste nu återhoppsadressen (på stacken) modifieras. Annars kommer konstanten 10 att tolkas som en instruktion omedelbart efter återhoppet. Följande instruktionssekvens illustrerar, dels hur parametern tas fram och dels justering av återhoppsadress:

```
dummyfunc:
      LDAB  [0,SP]      parameter->B
      LDX   0,SP        återhoppsadress->X
      INX               modifiera ..
      STX   0,SP        .. tillbaks till stack
      . . .
      . . .
      . . .
      RTS
```

Positionsoberoende kod

Med *positionsoberoende kod* menar man maskinkod som fungerar korrekt *oberoende av var den placeras i primärminnet*. Låt oss belysa detta med följande rader assemblerkod och den maskinkod som assemblatorn skapar av instruktionssekvenserna:

```
      ORG         $1000
main:
      NOP
      JMP         main
```

Genererad kod:
`A7 06 10 00`
Den *absoluta* adressen till symbolen main är kodad i instruktionen.

Programkoden är *inte* positionsoberoende ty *maskinkoden* kan inte flyttas (*relokeras*) i primärminnet och fortfarande fungera korrekt om inte den absoluta adressen $1000 samtidigt modifieras i koden.

Betrakta nu följande kod i stället, observera att *funktionen* är identisk med föregående sekvens:

```
      ORG         $1000
main:
      NOP
      BRA         main
```

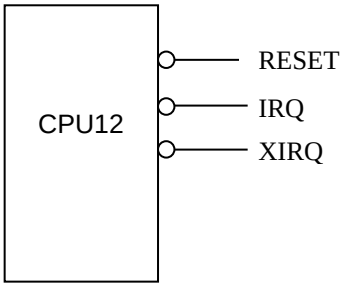
Genererad kod:
`A7 20 FD`
Adressen till main anges som en offset till programräknaren (FD=-3, PC-relativ)

Programkoden är positionsoberoende ty *maskinkoden* kan flyttas i primärminnet och programmet kommer fortfarande att fungera som avsett.

Egenskaperna hos positionsoberoende kod kan speciellt utnyttjas av *operativsystem*, eftersom ett positionsoberoende program kan flyttas utan att koden kräver modifiering kan operativsystemet bättre utnyttja datorsystemets primärminne.

Undantagshantering hos CPU12

Med ”undantag” (*exception*) menar vi speciella händelser som föranleder avbrott i sekventiellt utförande av instruktioner. Sådana händelser kan vara någon form av extern styrning (RESET, IRQ eller XIRQ, dvs. avbrott) men de kan också föranledas av något internt fel som uppstår under instruktionsexekvering.

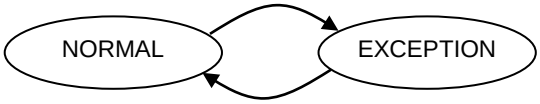


Extern styrning av CPU12, reset respektive avbrott

Processorns tillstånd

Processorn befinner sig alltid i något av tillstånden:

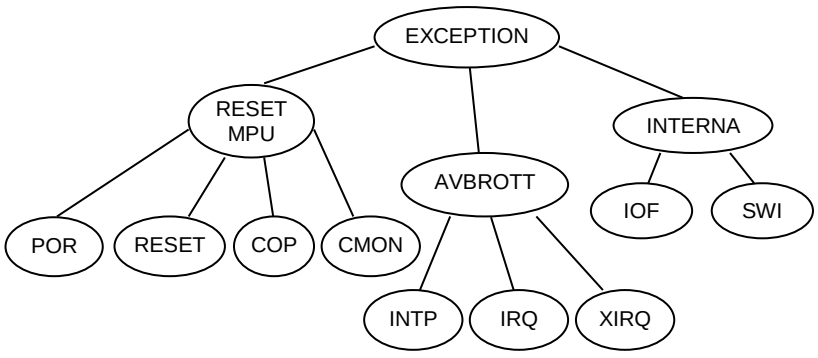
- *Normal*, processorn hämtar och utför instruktioner, dvs. normal exekvering.
- *Exception*, något “undantag” har inträffat som gör att processorn inte kan (eller ska) fortsätta normal exekvering.



Figur: Processorns olika tillstånd

Vi använder begreppet ”undantagshantering” (*exception handling*) för alla sorters händelser som tar processorn ut ur tillståndet “Normal”. Dessa händelser kan delas in i tre olika grupper: se även figur.

- *RESET MPU*, händelser som alltid föranleder återstart (RESET) av processorn.
- *AVBROTT*, externa händelser, dvs. utanför processorn, detta kan alltså vara enheter på samma krets som processorn (sammanbyggda periferienheter), det kan också vara en speciell insignal (IRQ eller XIRQ) som aktiveras.
- *INTERNA*, händelser som uppträder under programexekvering, exempelvis att en otillåten instruktion avkodas eller den speciella instruktionen SWI.



Olika typer av exceptions

RESET MPU

Det finns fyra olika händelser som föranleder återstart av processorn:

- POR, *Power On Reset*, vid spänningstillslag
- RESET, insignal till processorn aktiveras.
- COP, *Computer Operating Properly*. Detta är en så kallad *watchdog-funktion* som fungerar så att processorn måste skriva något värde till ett speciellt register med jämna mellanrum. Om programmet inte klarar av att genomföra detta så återstartas processorn (RESET) automatiskt. Typiska intervall för sådan uppdatering ligger mellan 15 och 500 ms. Funktionen kan alltså användas för att återstarta processorn då ett program av någon anledning hänger sig. Funktionen kan stängas av genom att en speciell bit i ett styrregister nollställs strax efter återstart.
- CMON, *Clock Monitor Reset*, är en annan tidsstyrd kontrollfunktion hos. Denna övervakar klockfrekvensen (E-klockan) och om frekvensen sjunker under 10 kHz genereras RESET.

AVBROTT

Avbrott kan komma från tre olika typer av källor. Exempelvis kan någon av de sammanbyggda periferienheterna (portar, räknare etc.) generera avbrott. För dessa finns förutbestämda autovektorer, dvs. vid avbrott från någon specifik enhet hämtar processorn avbrottsvektorn från en adress som avdelats för just denna enhet. Det finns också en speciell avbrottsingång, IRQ, som man ansluter till externa periferienheter, detta avbrott har endast en speciell avbrottsvektor. Ytterligare en typ av avbrottsingång XIRQ kan också anslutas externt. Avbrottet har också en speciell avbrottsvektor men skiljer sig på viktiga punkter från IRQ. Medan IRQ kan maskeras (behandlas under avbrottsprioriteter nedan) är XIRQ en form av icke maskerbart avbrott (*non-maskable interrupt*).

INTERNA

Undantagshantering kan också föranledas av interna händelser. Om processorn avkodar en otillåten operationskod kallas detta *Illegal Opcode Fetch* (IOF). Processorn avbryter då, sparar registerinnehåll på stacken, läser autovektorn för IOF och utför undantagshantering. Instruktionen *SoftWare Interrupt* (SWI) fungerar på samma sätt, men har en annan autovektor och en bestämd operationskod.

Följande tabeller, *Tabell: 1.34* och *Tabell 1.35* anger autovektorer hos MC68HCS12. I tabell *Tabell: 1.34* finns de autovektorer som är gemensamma för alla HCS12-varianter. Tabell *Tabell 1.35* listar autovektorer för varianten MC9S12DG256. Observera att andra varianter kan ha olika autovektortabeller.

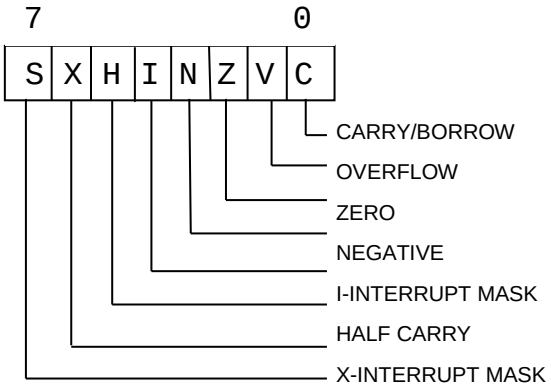
Adress (hex)	Funktion
FFFE	RESET, Startvektor
FFFC	Clock Monitor Fail
FFFA	COP Watchdog Timeout
FFF8	Illegal Op Code (ej impl i simulator)
FFF6	SWI
FFF4	XIRQ
FFF2	IRQ
FF00-FFF0	Enhetsspecifika vektorer, skiljer sig något beroende på de olika varianterna

Tabell: 1.34 MC68HCS12, gemensamma autovektorer

Adress (hex)	Funktion
FFF0	Real Time Interrupt
FFEE	Enhanced Capture Timer channel
FFEC	Enhanced Capture Timer channel 1
FFEA	Enhanced Capture Timer channel 2
FFE8	Enhanced Capture Timer channel 3
FFE6	Enhanced Capture Timer channel 4
FFE4	Enhanced Capture Timer channel 5
FFE2	Enhanced Capture Timer channel 6
FFE0	Enhanced Capture Timer channel 7
FFDE	Enhanced Capture Timer overflow
FFDC	Pulse accumulator A overflow
FFDA	Pulse accumulator input edge
FFD8	SPI0
FFD6	SCI0
FFD4	SCI1
FFD2	ATD0
FFD0	ATD1
FFCE	Port J
FFCC	Port H
FFCA	Modulus Down Counter underflow
FFC8	Pulse Accumulator B Overflow
FFC6	PLL lock
FFC4	CRG Self Clock Mode
FFC2	Används ej (BDLC)
FFC0	IIC Bus
FFBE	SPI1
FFBC	Reserverad
FFBA	EEPROM I-Bit
FFB8	FLASH I-Bit
FFB6	CAN0 wake-up
FFB4	CAN0 errors
FFB2	CAN0 receive
FFB0	CAN0 transmit
FFAE	Används ej (CAN1 wake-up)
FFAC	Används ej (CAN1 errors)
FFAA	Används ej (CAN1 receive)
FFA8	Används ej (CAN1 transmit)
FFA6	Används ej (ByteFlight Rx FIFO not empty)
FFA4	Används ej (ByteFlight receive)
FFA2	Används ej (ByteFlight general)
FFA0	Används ej (ByteFlight Synchronisation)
FF9E-FF98	Reserverade
FF96	CAN4 wake-up
FF94	CAN4 errors
FF92	CAN4 receive
FF90	CAN4 transmit
FF8E	Port P Interrupt
FF8C	PWM Emergency Shutdown
FF8A-FF80	Reserverade

Tabell 1.35 MC9S12DG128B/256B*, autovektorer*

Då undantagshantering påbörjas sparas först samtliga registerinnehåll på stacken. Bit I i CCR sätts till 1 för att förhindra ytterligare ett omedelbart avbrott. Därefter laddas PC med den autovektorn för den aktuella undantagshanteringen i avbrottsrutinen. Figur nedan visar processorns flaggregister Condition Code Register (CCR) och figur illustrerar hur registerinnehållen placerats på stacken inför utförandet av avbrottsrutinen.



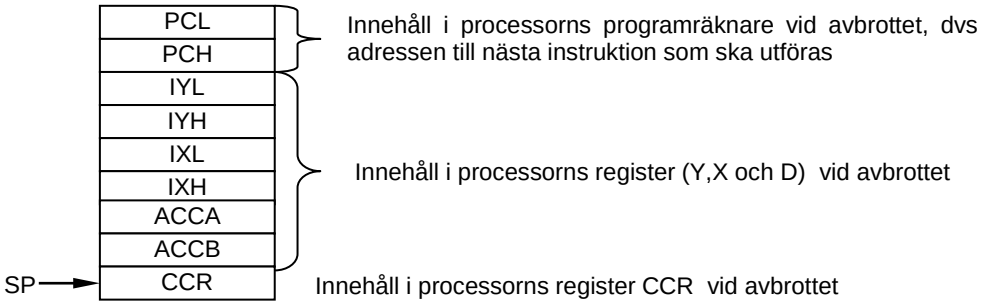
Processorns CCR

Bitar H,N,Z,V,C är statusbitar som sätts till 0 eller 1 vid exekvering av olika instruktioner.

S – Har endast betydelse vid utförande av STOP instruktion. Om bit S är 1 behandlas STOP som en NOP instruktion. Om bit S är 0 och STOP exekveras placeras processorn i stopp-tillstånd, dvs. alla operationer upphör och återupptas först vid ett IRQ eller XIRQ.

X – Bit X är en speciell typ av avbrottsmask för XIRQ. Bit X är 1 vid RESET och nollställs vanligtvis av programmet som startas omedelbart efter RESET. Då bit X nollställts kan den inte ett-ställas igen och XIRQ fungerar då som icke-maskerbart avbrott. XIRQ är verkningslös då bit X är 1.

I – Avbrottsmask, om bit I är 0 betjänas avbrott IRQ, annars inte.



Stackordning i avbrottsrutin, HCS12

Avbrottsprioriteter

Då flera avbrott uppträder samtidigt avgör *avbrottsarbitreringen* vilket avbrott som ska betjänas först. Ordningen kan i någon grad påverkas för de olika processorerna. Detta varierar mellan olika varianter inom respektive processorfamilj HC11 och HCS12. Generellt gäller dock att RESET MPU utförs alltid, IOF och SWI, XIRQ betjänas endast om bit X är noll, IRQ betjänas endast om bit I i CCR är noll.

Instruktioner för undantagshantering

Vid undantag sparas samtliga processorns register på stacken av hårdvaran.

HCS12

Mnemonic	Funktion	Operation
RTI	Återgå från undantagsrutin.	$(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + 2 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + 4 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) – 2 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + 4 \Rightarrow SP$
SWI	”Software Interrupt”	$(SP)-2 \Rightarrow SP, RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP, Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP, X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP, B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-1 \Rightarrow SP, CCR \Rightarrow (M_{(SP)} : M_{(SP+1)})$
TRAP	Icke implementerad instruktion	$(SP)-2 \Rightarrow SP, RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP, Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP, X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP, B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-1 \Rightarrow SP, CCR \Rightarrow (M_{(SP)} : M_{(SP+1)})$

HCS12X

Mnemonic	Funktion	Operation
RTI	Återgå från undantagsrutin.	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow CCR_H : CCR_L; (SP) + 2 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + 2 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow XH : XL; (SP) + 4 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PCH : PCL; (SP) – 2 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow YH : YL; (SP) + 4 \Rightarrow SP$
SWI	”Software Interrupt”	$(SP)-2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; CCR_H : CCR \Rightarrow (M_{(SP)} : M_{(SP+1)})$
TRAP	Icke implementerad instruktion	$(SP)-2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$ $(SP)-2 \Rightarrow SP; CCR_H : CCR \Rightarrow (M_{(SP)} : M_{(SP+1)})$

Exempel 1.45 Placering av Exceptionvektorer, assemblerkod

Följande programskelett illustrerar hur några avbrottsrutiner respektive avbrottsvektorer kan definieras i en fristående HCS12-applikation.

```
     ORG       $FFF2
     FDB       irq_service_routine
     FDB       xirq_service_routine
     FDB       software_interrupt_service_routine
     FDB       illegal_opcode_service_routine
     FDB       cop_service_routine
     FDB       clock_monitor_fail_service_routine
     FDB       Application_Start


; Symbolen “Application_Start_Address” kan vara godtycklig.
     ORG       Application_Start_Address
Application_Start:
     LDS       #TopOfStack
     ...
     ...
     ANDCC     #$FE      ; nollställ I-flagga
     JSR       _main
     ...


; Avbrottshanterare
irq_service_routine:
     RTI

xirq_service_routine:
     RTI

software_interrupt_service_routine:
     RTI

illegal_opcode_service_routine:
     RTI

cop_service_routine:
     RTI

clock_monitor_fail_service_routine:
     RTI
```

Kombinerad programmering

Programmering i assemblerspråk har fördelar men också stora nackdelar. Det är ett långsamt, och därmed också kostsamt sätt att programmera. Det färdiga assemblerprogrammet kan dessutom bara användas till den typ av maskin det skrivits för. Redan under 1950-talet började man utveckla programspråk som dels skulle göra programmeringsarbetet lättare, dvs språket ska ha konstruktioner som ligger närmre de algoritmer man vill att datorn ska utföra. och samtidigt skulle programspråket vara oberoende av den underliggande hårdvaran, dvs då nya datortyper introducerades skulle äldre programvara snabbt kunna flyttas till dessa bättre maskiner.

Programspråket *C* skapades av Dennis Ritchie (Bell Laboratories) i början av 1970-talet. Även om *C* är ett generellt användbart språk har det traditionellt använts som systemprogramspråk. Speciellt är operativsystemet *UNIX* och dess i dag mer bekanta efterföljare *LINUX,* skrivet i programspråket *C*. Den ursprungliga versionen av *C* blev snabbt populär, skälen till detta var flera: *C* tillhandahåller programkonstruktioner som gör det enkelt att implementera algoritmer på ett effektivt sätt. Alla vanliga datatyper finns representerade såväl som pekare och strängar. Det finns en rikhaltig uppsättning operatorer och ett "standard I/O" (input/output) bibliotek som täcker in- och utmatning till filer och terminaler. *C*-program är "effektiva", *C*-operatorer och programflödes-konstruktioner är nära relaterade till instruktioner som tillhandahålls av flertalet processorer. Ett annat sätt att uttrycka det: Det *semantiska gapet* mellan *C* och datorns hårdvara är litet. *C* skapade stora möjligheter att skriva portabla program, dvs applikationer som enkelt kunde kopieras till nya system.

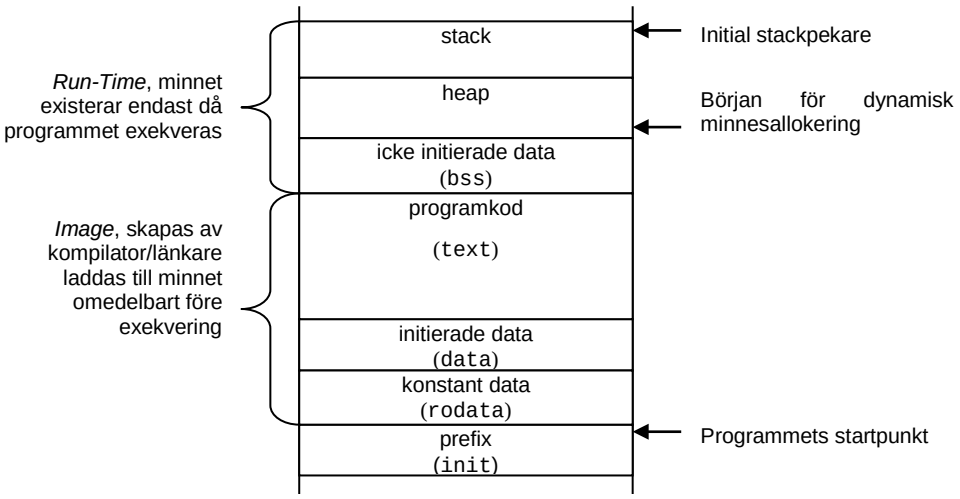
Populariteten hos *C* innebar dock att områden som inte hade beaktats av Kernighan/Ritchie blottlades, dvs brister hos språket identifierades och åtgärdades, ofta lokalt. Som en direkt följd skapades flera olika "dialekter" av språket. Utvecklingen *av UNIX System V* (AT and T) respektive *Berkeley UNIX* accelererade divergensen hos *C*-dialekterna. 1983 skapades kommite'n ANSI X3J11 (*American National Standards Institute*) med målsättning att inrätta en standard för programspråket *C*. Den standard som då definierades populärt för *ANSI-C*, medan den ursprungliga definitionen av *C* har kommit att kallas *K/R C* (Kernighan/Ritchie C). Standardiseringen av *C* har därefter tagits upp av ISO (International *Standard Organisation*) som därefter kontinuerligt drivit standardiseringen av *C*.

XCC - korskompilator

XCC är en ANSI-C korskompilator för flera olika typer av mikroprocessorer. Vi beskriver här speciellt *XCC12* för 68HCS12(X). I själva verket består *XCC* av flera delar: En *preprocessor* som hanterar alla preprocessor-direktiv i *C*, en *översättare*/*kodgenerator* som kontrollerar syntaxen i C-programmet och genererar assemblerkod för 68HC12, en *assembler* som assemblerar koden och genererar relokerbar kod i objektfiler och slutligen en *länkare* som kombinerar flera olika objektfiler till en, och skapar en laddfil som kan laddas i en 68HC12-baserad dator eller en simulator. Under detta moment *relokeras* koden, dvs alla symboliska namn ersätts med med absoluta adresser i måldatorns minne.

Minnesdisposition

Programkod och data indelas i olika segment, betrakta följande figur som beskriver hur minnesdispositionen för ett komplett program, under exekvering, kan se ut:



Figur 1.1: Minnesdisposition vid programexekvering

Figur 1.1 förstås bäst mot bakgrund av hur ett program översätts, sparas (eventuellt på en hårddisk), laddas till primärminnet och exekveras.

prefix

Prefixet, eller som det också kallas, *startupsekvens*, placeras först i varje C-program. Detta görs för att programmet alltid ska ha en enkelt identifierad startpunkt. Av konvention använder *XCC* segmentnamnet `init` för startupsekvensen. Den enklaste formen av prefix startupsekvens kan exempelvis vara:

```
SEGMENT   init
LDS       #TopOfStack
JSR       _main
```

Vi känner igen symbolen ”main” som namnet på det huvudprogram som måste finnas i varje C-program. Vi använder "underscore" framför symbolnamnet för att skilja C-funktionen "main" från (den översatta) assemblerfunktionen.

programkod

Här placeras all programkod. Den får inte vara självmodifierande, dvs segmentet förutsätts vara *read-only*. Kompilatorn gör en "bild" av maskinkod som laddas i minnet. Av konvention kallas detta segment för `text`.

konstant data

Deklarationer som exempelvis:

```
const     int c = 2;
```

innebär att symbolen `c` alltid kommer att ha värdet 2 under programmets exekvering. Detta kan ge kompilatorn värdefull information. Exempelvis kan kontroll utföras, att `c` inte av misstag tilldelas andra värden i programmet. Informationen kan också användas för effektivare kodgenerering med användning av *omedelbart* (*immediate*) adresseringssätt. I de fall konstant data tilldelas minne placeras de av konvention i segmentet `rodata`.

initierade data

Deklarationer som exempelvis:

```
int a = 2;
char array[] = {"Detta är en text"};
```

kan också användas för deklarera och initiera variabler. Innehållet är definierat från start, men kan komma att ändras under exekvering. Kompilatorn måste göra en "bild" av detta segment för att dessa initialvärden ska kunna laddas till minnet före exekvering. Eftersom sådana variabler kan komma att modifieras måste data-segmentet placeras i *read/write* minne.

Exempel 1.46

Beroende på hur en textsträng deklareras kommer kompilatorn att placera den i olika segment:

Satsen

```
printf("Denna text ...");
```

ger samma resultat på bildskärmen som:

```
char reftext[]={"Denna text ..."};
printf("%s", reftext);
```

dvs en textsträng skrivs ut.

Kompilatorn betraktar dock textsträngarna på helt olika sätt. I det första fallet är det en konstant sträng, som inte kan refereras av programmet från någon annan punkt än just i printf-satsen. Eftersom den inte kan refereras med någon tilldelningssats kan den heller inte ändras, textsträngen är därför *read-only*, och kan placeras i något av `rodata` eller `text`-segmenten.

I det andra fallet är det omedelbart klart att denna texsträng kan refereras även från andra ställen i programmet, t.ex:

```
strcpy(reftext, "Annan text...");
```

Textsträngen kan därför inte placeras i `text` segmentet, i stället hamnar den i `data` segmentet.

icke initierade data

Deklarationer som:

```
int  a;
char array[34];
```

osv, har inte något definierat innehåll från start. Det behövs alltså ingen "bild" av detta segment, till skillnad från `data/rodata` segmenten. Variabler som deklareras på detta sätt hamnar i segmentet `bss`.

stack

Stacken används av program under exekvering. Stackpekaren initieras under startupsekvensen. Stackens storlek bestäms av olika faktorer som hur mycket *read/write*-minne som finns tillgängligt i maskinen, hur mycket utrymme som reserverats för ”heap”, samt utrymmet som upplåtits för variabler (`bss`).

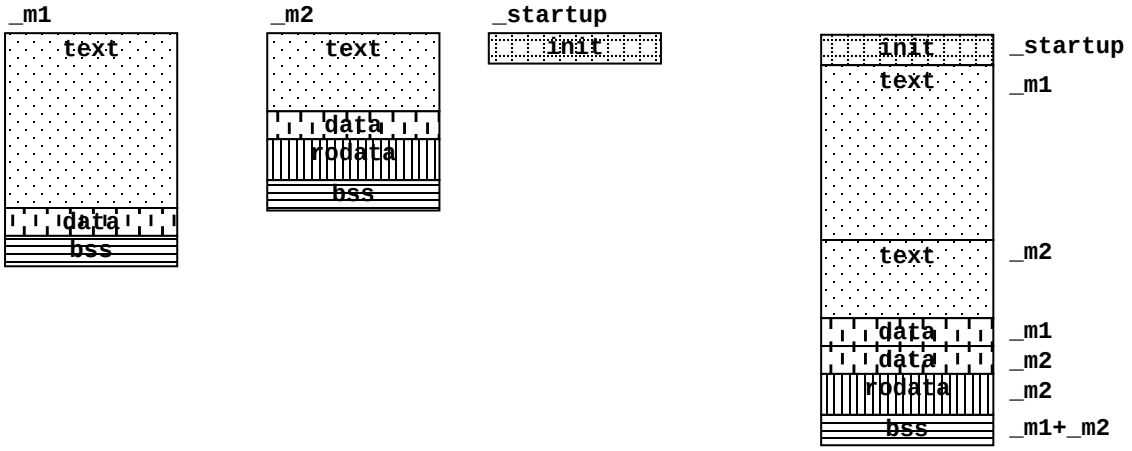
heap

Heapen benämns ofta det minnesutrymme som reserverats för programmets dynamisk minneshantering `malloc()`, `free()` etc. Även storleken av detta utrymme bestäms som regel automatiskt. I *XCC*, exempelvis kommer allt tillgängligt *read/write* minne utöver `bss` att användas för stack och heap vid programmets exekvering.

Låt oss sammanfatta detta. Vid kompilering skapas objektmoduler med följande information/innehåll:

- ”Read-Only”-sektion innehållande en "bild" av segmenten `init`, `text`, `data` och `rodata`.
- Information om storleken av `bss`-segmentet.
- Symboltabell innehållande alla globala symbolers relativa adresser (offset till segmentets början) i respektive segment. Observera att alla symboler är relokerbara, dvs absoluta adresser har ännu ej bestämts.

Vid länkningsproceduren kombineras nu innehållen från alla objektmoduler segmentsvis. Totala storleken av `bss`-segmentet bestäms och alla globala symbolers relativa adresser modifieras. Se, som exempel Figur 1.2.



Figur 1.2: modulerna ”_startup”, ”_m1” och _m2” kombineras till en ny objektmodul

Slutligen, måste alla segment tilldelas absoluta startadresser. Symbolerna kan därefter ges absoluta adresser och en sista relokering utför innan laddfilen skapas.

I/O-programmering

I detta avsnitt ska vi ge konkreta exempel på hur grundläggande funktioner i en HCS12 kan programmeras.

Inledningsvis behandlar vi klock-modulen. Vi visar först hur vi programstyrt bestämmer kretsens busfrekvens, dvs tidbasen i systemet. Vi ger därefter exempel på två olika implementeringar av en realtidsklocka. Den första implementeringen duger bra för många ändamål och ger en noggrannhet om c:a 1 ms. Vi visar ytterligare en implementering, med betydligt bättre noggrannhet, några tiotals nanosekunder.

Vi behandlar parallell in- och utmatning via portar A och B i HCS12, vi diskuterar också hur detta kan generaliseras att gälla även andra parallellportar hos HCS12.

Vi visar seriell kommunikation med HCS12’s ”SCI”-moduler.

Vi ger också några enkla exempel på pulsbreddsmodulering och A/D omvandling.

Genomgående illustreras programmeringen med implementering i assemblerspråk såväl som i C. I flera fall illustreras också applikationer med avbrott.

Slutligen ger vi exempel på hur en komplett fristående applikation för HCS12 bör utformas, dvs hur initiering och start procedur utförs och hur avbrottstabeller tillhandahålls.

Klockmodulen i HCS12

Klockmodulen (*CRG, Clock/Reset Generator*) tillhandahåller flera funktioner, bland annat:

- PLL (Phase Locked Loop) oscillator för programmerbar busfrekvens.
- ”Watchdog”-funktion, med programmerbar time-out.
- Periodisk avbrottsgenerator (*RTI, Real Time Interrupt*)

I HCS12-familjen har CRG-modulen vanligtvis basadress $34 men man ska alltid kontrollera vad som gäller för den specifika variant man använder. Modulens register visas i Figur 1.3. Utrymmet här medger inte att vi ger detaljerad behandling av samtliga register. För en fullständig beskrivning hänvisas till fabrikantens dokumentation.

Clock Reset Generator (CRG)											
Offset		7	6	5	4	3	2	1	0	Mnemonic	Namn
$00	R	0	0	SYN5	SYN4	SYN3	SYN2	SYN1	SYN0	SYNR	Synthesizer Register
	W										
$01	R	0	0	0	0	REFDV3	REFDV2	REFDV1	REFDV0	REFDV	Reference Divide Register
	W										
$02	R	0	0	0	0	0	0	0	0	CTFLG	*)Test Flags Register
	W										
$03	R	RTIF	PORF	LVRF	LOCKIF	LOCK	SCMIE	SCMIF	SCM	CRGFLG	Flags Register
	W										
$04	R	RTIE	0	0	LOCKIE	0	0	SCMIE	0	CRGINT	Interrupt Enable Register
	W										
$05	R	PLLSEL	PSTP	SYSWAI	ROAWAI	PLLWAI	CWAI	RTIWAI	COPWAI	CLKSEL	Clock Select Register
	W										
$06	R	CME	PLLON	AUTO	AOQ	0	PRE	PCE	SCME	PLLCTL	PLL Control Register
	W										
$07	R	0	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0	RTICTL	RTI Control Register
	W										
$08	R	WCOP	RSBCK	0	0	0	CR2	CR1	CR0	COPCTL	COP Control Register
	W										
$09	R	0	0	0	0	0	0	0	0	FORBYP	*)Force and Bypass Test Register
	W										
$0A	R	0	0	0	0	0	0	0	0	CTCTL	*)Test Control Register
	W										
$0B	R	0	0	0	0	0	0	0	0	ARMCOP	COP Arm/Timer Reset
	W	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0		

Anm: Skuggade fält utmärker bitar som ej kan skrivas
*) Registren används endast för fabrikstest av kretsen.

Figur 1.3: Register i CRG-modulen

Kontroll av busfrekvens PLL, SYNR och REFDV

Systemets klocka (busfrekvens) kan väljas mellan PLLCLK och OSCCLK. Sambanden mellan de olika klockorna beskrivs av följande ekvation:

$$PLLCLK = 2 \times OSCCLK \times \frac{(SYNR + 1)}{(REFDV + 1)}$$

Genom att ett-ställa biten PLLSEL i register CLKSEL väljs PLLCLK som bas för arbetstakten. Den effektiva busfrekensen bestäms då av OSCCLK och kvoten mellan SYNR och REFDV. Det finns dock två viktiga begränsningar i valet av värden hos SYNR och REFDV.

- PLLCLK får aldrig vara *mindre än* OSCCLK eftersom detta äventyrar stabilitetsvillkoren i oscillatorn.
- PLLCLK/2 får aldrig vara *större än* nominella arbetsfrekvensen hos kretsen. För första generationens HCS12 innebär detta att PLLCLK/2 < 25 MHz.

Exempel 1.47

Antag vi vill maximera arbetstakten hos en HCS12 med 8 MHz kristall. Vi får:

$$50MHz > 2 \times 8MHz \times \frac{(SYNR+1)}{(REFDV+1)}$$

Den närmsta arbetstakt vi kan skapa är 48 MHz med valen SYNR = 5 och REFDV = 1, vi får då:

$$2 \times 8MHz \times \frac{(5+1)}{(1+1)} = 2 \times 8 \times 3MHz = 48MHz$$

För att byta systemklocka ska vi alltså modifiera registren SYNR och REFDV. Därefter sätter vi biten PLLSEL till 1. Innan vi ändrar denna bit till ett måste vi dock kontrollera att PLL-kretsen genererar en stabil PLLCLK, vi säger att den då är *låst*. Detta kontrolleras via biten `LOCK` i registret `CRGFLG.`

Exempel 1.48: Busfrekvens hos HCS12, assemblerspråk

```
* Adressdefinitioner för register
REFDV       EQU   $35
SYNR        EQU   $34
CLKSEL      EQU   $39
* Bitdefinitioner
PLLSEL      EQU   $80
LOCK        EQU   8

* Registervärden enligt Exempel 1.47
REFDVVal:   EQU   1
SYNRVal:    EQU   5

*     Generisk kod för programmerad arbetstakt...
      MOVB  #REFDVVal,REFDV
      MOVB  #SYNRVal,SYNR
wait:
      BRCLR CRGFLG,#LOCK,wait       ; vänta tills PLL låst...
      BSET  CLKSEL,#PLLSEL          ; växla systemklocka till PLL.
```

Exempel 1.49: Busfrekvens hos HCS12, motsvarande i C:

Här är det lämpligt att göra en typdeklaration för CRG-modulen, exempelvis enligt följande:

```
typedef struct sCRG{
      volatile unsigned char synr;
      volatile unsigned char refdv;
      volatile unsigned char ctflg;
      volatile unsigned char crgflg;
      volatile unsigned char crgint;
      volatile unsigned char clksel;
      volatile unsigned char pllctl;
      volatile unsigned char rtictl;
      volatile unsigned char copctl;
      volatile unsigned char forbyp;
}CRG, *PCRG ;

#define     CRG_BASE    0x34        /* Basadress för CRG-modulen */
#define     REFDVVal    1     /* Registervärden enligt Exempel 1.47.. */
#define     SYNRVal     5


#define     PLLSEL      0x80  /* Bitdefinitioner */
#define     LOCK        8
```

```
/*    Generisk kod för programmerad arbetstakt... */

      ( ( ( PCRG ) ( CRG_BASE ))->refdv ) = REFDVVal;
      ( ( ( PCRG ) ( CRG_BASE ))->synr ) = SYNRVal;
      /* vänta tills PLL låst... */
      while( (( ((volatile PCRG) (CRG_BASE))->crgflg ) & LOCK )== 0);
      /* växla systemklocka till PLL */
      ( ((PCRG) (CRG_BASE))->clksel ) |= PLLSEL;
```

Observera speciellt sekvensens sista tilldelningssats som valts för att endast påverka biten PLLSEL och låta övriga bitar i registret vara opåverkade.

Realtidsklockan i CRG-modulen

CRG-modulen innehåller även en RTI- (*Real Time Interrupt*) funktion med vars hjälp vi enkelt implementerar en noggrann klocka (realtidsklocka). RTI-funktionen utgörs av en enkel räknare kopplad till centralenhetens avbrottsingång. Funktionen kontrolleras via tre register: `CRGINT` för att kontrollera avbrottsfunktionen, `RTICTL` för att bestämma avbrottsfrekvensen och `CRGFLG` för att kvittera avbrott (jämför Figur 1.3).

`CRGINT` (basadress + 4)

Registret används för att aktivera avbrott

$04	R	RTIE	0	0	LOCKIE	0	0	SCMIE	0
	W								

- `RTIE`: Aktivera avbrott från RTI-funktionen. Denna bit måste sättas till 1 för att avbrott ska genereras.
- `LOCKIE`, `SCMIE`, används ej här.

`RTICTL` (basadress + 7)

Registret används för att initiera en tidbas för den periodiska räknaren. En skrivning till detta register aktiverar RTI-funktionen.

$07	R	0	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0
	W								

RTR-bitarna bestämmer avbrottsintervallet från räknaren. Systemets klockfrekvens (kristalloscillatorns frekvens) delas med ett tal specificerat av RTR-bitarna enligt följande tabell:

RTR [3:0]	RTR[6:4]							
	000 (OFF)	001	010	011	100	101	110	111
0000	OFF	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0001	OFF	$2x2^{10}$	$2x2^{11}$	$2x2^{12}$	$2x2^{13}$	$2x2^{14}$	$2x2^{15}$	$2x2^{16}$
0010	OFF	$3x2^{10}$	$3x2^{11}$	$3x2^{12}$	$3x2^{13}$	$3x2^{14}$	$3x2^{15}$	$3x2^{16}$
0011	OFF	$4x2^{10}$	$4x2^{11}$	$4x2^{12}$	$4x2^{13}$	$4x2^{14}$	$4x2^{15}$	$4x2^{16}$
0100	OFF	$5x2^{10}$	$5x2^{11}$	$5x2^{12}$	$5x2^{13}$	$5x2^{14}$	$5x2^{15}$	$5x2^{16}$
0101	OFF	$6x2^{10}$	$6x2^{11}$	$6x2^{12}$	$6x2^{13}$	$6x2^{14}$	$6x2^{15}$	$6x2^{16}$
0110	OFF	$7x2^{10}$	$7x2^{11}$	$7x2^{12}$	$7x2^{13}$	$7x2^{14}$	$7x2^{15}$	$7x2^{16}$
0111	OFF	$8x2^{10}$	$8x2^{11}$	$8x2^{12}$	$8x2^{13}$	$8x2^{14}$	$8x2^{15}$	$8x2^{16}$
1000	OFF	$9x2^{10}$	$9x2^{11}$	$9x2^{12}$	$9x2^{13}$	$9x2^{14}$	$9x2^{15}$	$9x2^{16}$
1001	OFF	$10x2^{10}$	$10x2^{11}$	$10x2^{12}$	$10x2^{13}$	$10x2^{14}$	$10x2^{15}$	$10x2^{16}$
1010	OFF	$11x2^{10}$	$11x2^{11}$	$11x2^{12}$	$11x2^{13}$	$11x2^{14}$	$11x2^{15}$	$11x2^{16}$
1011	OFF	$12x2^{10}$	$12x2^{11}$	$12x2^{12}$	$12x2^{13}$	$12x2^{14}$	$12x2^{15}$	$12x2^{16}$
1100	OFF	$13x2^{10}$	$13x2^{11}$	$13x2^{12}$	$13x2^{13}$	$13x2^{14}$	$13x2^{15}$	$13x2^{16}$
1101	OFF	$14x2^{10}$	$14x2^{11}$	$14x2^{12}$	$14x2^{13}$	$14x2^{14}$	$14x2^{15}$	$14x2^{16}$
1110	OFF	$15x2^{10}$	$15x2^{11}$	$15x2^{12}$	$15x2^{13}$	$15x2^{14}$	$15x2^{15}$	$15x2^{16}$
1111	OFF	$16x2^{10}$	$16x2^{11}$	$16x2^{12}$	$16x2^{13}$	$16x2^{14}$	$16x2^{15}$	$16x2^{16}$

Tabell 1.36

Exempel 1.50: Bestämning av avbrottsintervall

Antag vi önskar avbrott med 10 ms intervall och att vår kristallfrekvens är 8 MHz.
Det gäller att:

$$\frac{OSCCLK}{RTR} = RTIfreq$$

dvs

$$\frac{8\times10^6}{RTR} = \frac{1}{10^{-2}} \Rightarrow$$

$$RTR = x \times 2^y = 8\times10^4 \text{ där: } 1\le x\le 16 \text{ och } 10\le y\le 16$$

Identifiera $x = 8$ ger:

$$2^y = 10^4 \Rightarrow \log 2^y = \log(10^4) \Rightarrow$$

$$y = \frac{\log 10^4}{\log 2} = \frac{4}{\log 2} \approx 13{,}29$$

Vi provar oss slutligen fram till det bästa värdet (approximativt 8×10^4)
$8\times2^{13} = 65536$
$9\times2^{13} = 73728$
$10\times2^{13} = 81920$
Den bästa approximationen har vi för
RTR = 100 1001 = $49, som medför: $10x2^{13} = 81920$
Eftersom detta värde är något större än det optimala, kommer vi att få en något längre periodtid, nämligen:
avbrottsfrekvens = $8\times10^6 / 81920 = 97.656$ Hz
vilket ger periodtiden:
0.01024 s = 10,24 ms.
Klockan kommer alltså att "gå för sakta" som en följd av detta systematiska fel.

`CRGFLG` (basadress + 3)

Statusregister, alla bitar är läsbara och varje bit representerar någon händelse.

$03	R W	RTIF	PORF	LVRF	LOCKIF	LOCK	SCMIE	SCMIF	SCM

- `RTIF`: Biten sätts till 1 vid avbrott från RTI-funktionen. Avbrottsignalen kvitteras genom att en etta skrivs till RTIF-biten.
- Övriga bitar, används ej här, nollor kan skrivas till dessa bitar utan att påverka någon funktion.

Då räknaren initierats/aktiverats, kommer den att räkna ned ett intervall och därefter begära avbrott, räknarvärdet initieras därefter på nytt automatiskt av kretsen och ett nytt intervall påbörjas. Avbrottet måste kvitteras för att återställa IRQ-signalen till en passiv nivå, detta görs i avbrottsrutinen. I följande exempel illustreras hur en enkel realtidsklocka implementeras i ett HCS12 system.

Exempel 1.51: Implementering av realtidsklocka med HCS12

I detta exempel visas en mycket enkel implementering av en realtidsklocka. Om det använda systemet är försett med en 8 MHz kristall så kommer avbrott att genereras med 10,24 ms intervall. Vi implementerar först med HCS12 assemblerspråk

```
* Adressdefinitioner
CRGFLG      EQU         $37
CRGINT      EQU         $38
RTICTL      EQU         $3B

            SEGMENT     text
timer_init:

* Initiera RTC avbrottsfrekvens (se Exempel 1.50 )
            MOVB        #$49,RTICTL ; För MC12/8MHz
* Anmärkning ang. avbrottsfrekvens.
* Om programexemplet används i simulatorn kommer denna
* perioden att bli mycket lång. Det kan vara bättre att
* använda kortast tänkbara intervall för tester i simulatorn
*           MOVB        #$10,RTICTL ; För simulator

* Aktivera avbrott från CRG-modul
            MOVB        #$80,CRGINT

* Avbrottsvektor
            LDX         #timer_interrupt
* För laborationssystem 'MC12' med 'DBG12'
            STX         $3FF0
* För simulator (ETERM och XCC)
            STX         $FFF0

* nollställ CPU'ns I-flagga (tillåt avbrott)
            CLI
            RTS
*
* Avbrottsrutin
* Normalt sett ska vi här underhålla en mjukvaruklocka
* men i detta exempel gör vi minsta möjliga...

timer_interrupt:
* Kvittera avbrott från RTC
            BSET        CRGFLG,#$80
            RTI
```

Implementering av samma realtidsklocka, i 'C'

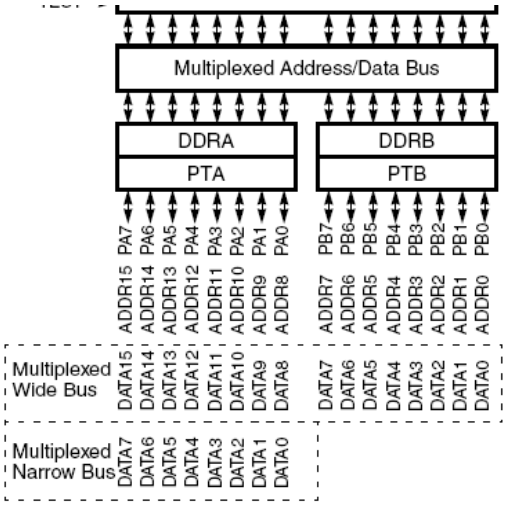
```
void timer_init( void )
{     /* RTC avbrottsfrekvens ...*/
      ( ( ( PCRG ) ( CRG_BASE ))->rtictl ) = 0x49;
      /* Avbrottsvektor för laborationssystem 'MC12' med 'DBG12' */
      *(unsigned short *) 0x3FF0 = (unsigned short) timer_interrupt;
      /* Avbrottsvektor för simulator */
      *(unsigned short *) 0xFFF0 = (unsigned short) timer_interrupt;
      /* Aktivera avbrott från CRG-modul */
      ( ( ( PCRG ) ( CRG_BASE ))->crgint ) = 0x80;
      __asm(" CLI");    /* nollställ CPU'ns I-flagga (tillåt avbrott) */
}

__interrupt void timer_interrupt( void )
{     // Kvittera avbrott från RTC
      ( ( ( PCRG ) ( CRG_BASE ))->crgflg ) |= 0x80;
}
```

Observera dock att varje enhet i ‘clock’ motsvarar 10,24 ms. För att få en exakt klocka måste alltså även dennatid korrigeras med jämna mellanrum.

Parallell kommunikation via portar A och B

Funktionen hos merparten av de fysiska anslutningarna (pinnarna) hos en HCS12 är programmerbar. Exempel på det är portar A och B. Betrakta Figur 1.4, som är tagen från blockbeskrivningen av en DG256. Figuren illustrerar hur de 16 anslutningarna, som går under namnen Port A resp Port B, kan användas på tre olika sätt.

1. Multiplexed Wide Bus – 16 bitars extern adressbuss och 16 bitars databus.
2. Multiplexed Narrow Bus – 16 bitars extern adressbuss och 8 bitars databus
3. Single chip – Ingen extern bus. Port A och Port B kan användas som generella IO-portar.



Figur 1.4: Alternativ användning av PA/PB

Vi tittar nu på hur vi kan använda dessa portar i ”Single Chip”-mode. Se Figur 1.5 som visar de första registren i MEBI-modulen. PORTA och PORTB är två identiska 8-bitars portar. Riktningen (in eller ut) kan för varje bit väljas oberoende av de övriga bitarna. Detta görs genom initiering av *Data Direction Register A* (DDRA) respektive *Data Direction Register B* (DDRB). MEBI-modulen har vanligtvis basadressen 0 i alla HCS12-system.

Multiplexed External Bus Interface (MEBI)										
Offset		7	6	5	4	3	2	1	0	Mnemonic
$00	R	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	PORTA
	W									
$01	R	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	PORTB
	W									
$02	R	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	DDRA
	W	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	
$03	R	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	DDRB
	W	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	
$04	R	.....								
	W									

Figur 1.5: Register för Port A/B som generell IO

DDRA bestämmer riktning för varje bit i PORTA. ”1” i DDRA innebär att motsvarande bit i PORTA fungerar som en utport. ”0” i DDRA innebär att motsvarande bit fungerar som en inport.
En läsning från PORTA kommer att returnera den logiska nivån för en ports pinne om biten är en inport, annars returneras det senaste värde som skrevs till biten. PORTB och DDRB fungerar på identiskt sätt.

Exempel 1.52

Ange i såväl assemblerspråk som C, programkonstruktioner som initierar port A för användning som inport samt port B för användning som utport.
Lösning:

```
PORTA       EQU   0
PORTB       EQU   1
DDRA        EQU   2
DDRB        EQU   3
      ...
            CLR   DDRA
            MOVB  #$FF,DDRB
      ...

typedef struct sMEBI{
      volatile unsigned char porta;
      volatile unsigned char portb;
      volatile unsigned char ddra;
      volatile unsigned char ddrb;
}MEBI, *PMEBI;

#define     MEBI_BASE   0

( ( ( PMEBI )( MEBI_BASE ))-> ddra ) = 0;
( ( ( PMEBI )( MEBI_BASE ))-> ddrb ) = 0xFF;
```

Seriell kommunikation via SCI

SCI-modulen är konstruerad för olika serieprotokoll som punkt-till-punkt, exempelvis *RS232*, och buss-protokoll som exempelvis *Local Interconection Network* (LIN). SCI-modulens registeruppsättning visas i Figur 1.6.

Serial Communication Interface (SCI)											
Offset		7	6	5	4	3	2	1	0	Mnemonic	Namn
$00	R	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8	SCIBDH	Baud Rate Register High
	W										
$01	R	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0	SCIBDL	Baud Rate Register Low
	W										
$02	R	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	SCICR1	Control Register 1
	W										
$03	R	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCICR2	Control Register 2
	W										
$04	R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCISR1	Status Register 1
	W										
$05	R	0	0	0	0	0	BRK13	TXDIR	RAF	SCISR2	Status Register 2
	W										
$06	R	R8	T8	0	0	0	0	0	0	SCIDRH	Data Register High
	W										
$07	R	R7	R6	R5	R4	R3	R2	R1	R0	SCIDRL	Data Register Low
	W	T7	T6	T5	T4	T3	T2	T1	T0		

Figur 1.6: SCI Register

Dataöverföringen sker via `SCIDRH` och `SCIDRL`. `SCIDRH` används endast vid 9 bitars dataöverföring. Observera att tecken kan skickas och tas emot samtidigt (*full duplex*) eftersom det finns separata register (*T, transmitter* och *R, receiver*).

Innehållen i `SCIBDH` och `SCIBDL` används för att bestämma överföringshastigheten, *baud rate*, som beräknas enligt följande:

$$baudrate = \frac{PLLCLK}{16 \times BR}$$

där: *BR* är innehållet i `SCIBDH-SCIBDL` (1-8191).
Vi kan också skriva sambandet som:

$$BR = \frac{PLLCLK}{16 \times baudrate}$$

vilket ger oss ett enkelt samband för att bestämma initieringsvärden till `SCIBDH`/`SCIBDL` för någon önskad *baudrate*.

Exempel 1.53

Bestäm initieringsvärden *BR* för olika baudrates, 9 600, 57 600 och 256 kbaud i ett HCS12-system med 48 MHz systemklocka.

Lösning:

Vi använder samband enligt ovan och sammanställer resultaten i följande tabell:

9 600	$\frac{48\times10^6}{16\times9600} = 312{,}5$	$\frac{48\times10^6}{16\times312} \approx 9615$, $\frac{48\times10^6}{16\times313} \approx 9585$
57 600	$\frac{48\times10^6}{16\times57600} \approx 52{,}08333$	$\frac{48\times10^6}{16\times52} \approx 57692$
256 000	$\frac{48\times10^6}{16\times256000} = 11{,}71875$	$\frac{48\times10^6}{16\times12} \approx 250000$

Det framgår omedelbart att vi inte kan besämma *BR* för en exakt överensstämmelse. Vi ser att vi tvingas använda 9615 (eller 9585) i stället för 9600 vilket vanligtvis fungerar eftersom det finns vissa toleranser i samplingen.

Låt oss nu studera SCI-modulen genom att bygga upp några typiska användarfall. Utgående från Figur 1.6 deklarerar vi först en lämplig datatyp för modulen:

```
typedef     struct sSCI{
      volatile unsigned short scibd;
      volatile unsigned char scicr1;
      volatile unsigned char scicr2;
      volatile unsigned char scisr1;
      volatile unsigned char scisr2;
      volatile unsigned char scidrh;
      volatile unsigned char scidrl;
}SCI, *PSCI;
```

RS232 kommunikation, 9600 baud, 8 bitar data, ingen paritet, utan avbrott

Betrakta kontrollregister 2, bitar 7-4 används enbart då vi vill använda *avbrott*. I detta fall ska därför dessa bitar vara 0.

Offset		7	6	5	4	3	2	1	0	Mnemonic	Namn
$03	R	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCICR2	Control Register 2
	W										

Biten RWU används då vi utnyttjar energibesparande moder och SBK (Send BreaK character) har speciell funktion som vi återkommer till senare. Bitar TE och RE har vi dock användning för. Deras respektive funktion är:

Bit 3 TE	**Transmitter Enable Bit** — TE aktiverar SCI'ns sändare och konfigurerar pinnen TXD så att den kontrolleras av SCI'n. 0 Transmitter inaktiv 1 Transmitter aktiv
Bit 2 RE	**Receiver Enable Bit** — RE aktiverar SCI'ns mottagare. 0 Receiver inaktiv 1 Receiver aktiv

Av funktionsbeskrivningen drar man slutsatsen att SCI'n kan användas som *enbart sändare*, som *sändare/mottagare*, dock *ej* som *enbart mottagare*.

Vi finner att i detta fall ska bitar TE och RE sättas till 1 medan övriga bitar ska vara 0.

```
( (( PSCI )( SCI_BASE ))->scicr2 ) = 0x0C;
```

Vi använder baudrate-beräkningar från Exempel 1.53 vilket ger oss följande initiering av *baudrate* registren:

```
( (( PSCI )( SCI_BASE ))->scibd) = 312;
```

Låt oss nu titta närmre på statusregister 1.

$04	R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCISR1	Status Register 1
	W										

Följande tabell beskriver kortfattat statusbitarnas betydelse:

Bit	Beskrivning
7 TDRE	**Transmit Data Register Empty Flag** — Då TDRE är satt till 1 kan ett nytt tecken skrivas till SCIDR(T). Biten nollställs av att SCISR1 läses och SCIDRL skrivs, i denna ordning. Då tecknet skiftats ut från SCIDRL sätter SCI TDRE till 1 igen. 0 SCIDRL(T) upptaget 1 SCIDRL(T) ledigt
6 TC	**Transmit Complete Flag** — TC nollställs då en sändning påbörjas och är noll så länge sändningen pågår. 0 Sändning pågår 1 Ingen sändning pågår
RDR F	**Receive Data Register Full Flag** — RDRF sätts till 1 då ett nytt tecken finns i SCIDRL. Flaggan nollställs av att SCISR1 läses och SCIDRL läses, i denna ordning. 0 Inget nytt tecken 1 Nytt tecken
4 IDLE	**Idle Line Flag** — IDLE sätts till 1 då 10 konsekutiva logiska ettor (om M=0) eller 11 konsekutiva logiska ettor (om M=1) mottagits. Funktionen används i buss-konfigurationer för att detektera "ledig buss". 0 Mottagaren är aktiv 1 Mottagaren är inaktiv, bussen är ledig
3 OR	**Overrun Flag** — OR sätts till 1 om ett nytt tecken anländer till SCIDRL(R) innan det tidigare tecknet lasts. Nollställs igen då SCISR1 läses. 0 Inget förlorat tecken 1 Förlorat tecken
2 NF	**Noise Flag** — NF sätts till 1 om störande brus förekommer på mottagarens ingång. Nollställs igen då SCISR1 läses. 0 Ingen störning 1 Störande brus
1	**Framing Error Flag** — FE sätts till 1 då en stoppbit läses som en logisk nolla. Nollställs igen då SCISR1

FE	läses. Observera att inget nytt tecken kan mottas innan FE nollställs. Typiskt inträffar detta fel om sändare/mottagare arbetar med olika baudrate. 0 Inget ramfel 1 Ramfel
0 PF	**Parity Error Flag** — PF sätts till 1 då ett paritetsfel upptäcks och PE samtidigt är 1. Nollställs igen då SCISR1 och SCIDRL läses efter varandra. 0 Inget paritetsfel 1 Paritetsfel

För att kunna *sända* ett tecken måste `SCIDRL(T)` vara tomt, annars förstör vi ett tidigare försök att skicka ett tecken. Följande programkonstruktion utför en upprepad statustest:

```
while( (( ((PSCI) (SCI_BASE))->scisr1 ) & 0x80 )== 0);
```

den är ekvivalent med följande sekvens instruktioner:

```
_1:
      LDAB  SCISR1
      ANDB  #$80
      BEQ   _1
```

Dvs. programmet ”hänger” till `TDRE` blir 1 och `SCIDRL(T)` är redo för ett nytt tecken

På motsvarande sätt indikerar `RDRF` att `SCIDRL(R)` innehåller ett nytt tecken som anlänt på seriegränssnittet. Statustesten blir den samma fast en annan bit testas:

```
while( (( ((PSCI) (SCI_BASE))->scisr1 ) & 0x20 )== 0);
```

Vi kan nu sätta samman en uppsättning funktioner enligt följande:

```
      void serial_init( void ); /* initiera gränssnittet */
      void serial_out( char c ); /* skicka ett tecken via gränssnittet */
      char serial_in( void );    /* ta mot ett tecken från gränssnittet */
```

Implementeringen, ”busy wait” av dessa följer här:

Exempel 1.54: RS232 kommunikation, “busy-wait”, i C

Vi förutsätter att `SCI_BASE` (basadress till SCI-modulen) är definierad.

```
void serial_init( void )
{
 ( (( PSCI )( SCI_BASE ))->scicr2 ) = 0x0C;
 ( (( PSCI )( SCI_BASE ))->scibd  ) = 312;
}

void serial_out( char c )
{
      while( (( ((PSCI) (SCI_BASE))->scisr1 ) & 0x80 )== 0);
      ( ((PSCI) (SCI_BASE))->scidrl ) = c;
}

char serial_in( void )
{
      while( (( ((PSCI) (SCI_BASE))->scisr1 ) & 0x20 )== 0);
      return ( ((PSCI) (SCI_BASE))->scidrl );
}
```

Exempel 1.55: RS232 kommunikation, "busy-wait", i HC12 assemblerspråk

Vi förutsätter här att SCI (basadressen till SCI-modulen) är definierad.

```
SCIBD       EQU         SCI
SCICR1      EQU         SCI+2
SCICR2      EQU         SCI+3
SCISR1      EQU         SCI+4
SCISR2      EQU         SCI+5
SCIDRH      EQU         SCI+6
SCIDRL      EQU         SCI+7

            SEGMENT     text
            EXPORT      _serial_init
            EXPORT      _serial_in
            EXPORT      _serial_out

_serial_init:
            LDAB        #$0C
            STAB        SCICR2
            LDD         #312
            STD         SCIBD
            RTS

_serial_out:
            BRCLR       SCISR1,#0x80,_serial_out
            LDAB        2,SP
            STAB        SCIBD
            RTS

_serial_in:
            BRCLR       SCISR1,#0x20,_serial_in
            LDAB        SCIBD
            RTS
```

Vi har nu en uppsättning funktioner som hanterar seriekommunikation i dess enklaste form. Det finns dock flera kompletteringar man kan göra för att öka robusthet och användbarhet av funkktionerna. Vi kan exempelvis lägga till felkontroll i mottagarrutinen så att inte felaktiga tecken returneras, det skulle kunna se ut på följande sätt:

```
char serial_in_errorcheck( void )
{
      char  c;
      while(1)
      {
            while( (( ((PSCI) (SCI_BASE))->scisr1 ) & 0x20 )== 0);
            c = ( ((PSCI) (SCI_BASE))->scidrl );      /* Läs tecken */
            if(((((PSCI)( SCI_BASE ))->scisr1) & 0xF )==0)
            {     /* allt är Ok, returnera tecknet */
                  return ( ((PSCI) (SCI_BASE))->scidrl );
            }
            /* Felaktigt tecken, kassera och vänta på nytt... */
      }
}
```

”Busy-wait” strategin är tilltalande i sin enkelhet men är inte alltid användbar i praktiken. En variant är då att använda ”Polling” (rundfrågning). Vi kan utöka vår uppsättning med två funktioner vars uppgift blir att undersöka status hos SCI’n men inte bli ”hängande” i någon vänteslinga. Ett gränssnitt bestående av fem funktioner skulle då kunna implementeras av följande:

```
      void serial_init( void ); /* initiera gränssnittet */
      int check_serial_out( void ); /* kontrollera om tecken kan skickas */
      void serial_out( char c ); /* skicka ett tecken via gränssnittet */
      int check_serial_in( void ); /* kontrollera om tecken finns */
      char serial_in( void );    /* ta mot ett tecken från gränssnittet */
```

Exempel 1.56: RS232 kommunikation, “polling”, i C

Vi förutsätter att `SCI_BASE` (basadress till SCI-modulen) är definierad.

```
void serial_init( void )
{
 ( (( PSCI )( SCI_BASE ))->scicr2 ) = 0x0C;
 ( (( PSCI )( SCI_BASE ))->scibd  ) = 312;
}

int check_serial_out( void )
{
      /* returnera 1, om sändaren ledig,returnera 0 annars */
      return( (( ((PSCI) (SCI_BASE))->scisr1 ) & 0x80 )!= 0);

}

void serial_out( char c )
{
       ( ((PSCI) (SCI_BASE))->scidrl ) = c;
}

int check_serial_in( void )
{
      /* returnera 1, om tecken finns hos mottagaren,
            returnera 0 annars */
      return( (( ((PSCI) (SCI_BASE))->scisr1 ) & 0x20 )!= 0);
}

char serial_in( void )
{
      return ( ((PSCI) (SCI_BASE))->scidrl );
}
```