KOMPENDIUM

# GRUNDLÄGGANDE DATORTEKNIK FÖR HÖGSKOLANS INGENJÖRSUTBILDNINGAR

ROGER JOHANSSON
INSTITUTIONEN FÖR DATA OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA, 2013

# *Innehåll*





# 9 Assemblerprogrammering

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', dvs. sätta ihop ett program utav en sekvens instruktioner). Utifrån sekvenser av instruktioner i assemblerspråk sätts alltså programmet samman till ett komplett program i maskinspråket för den använda processorn. Den omvända processen, att "dissassemblera" (engelskans 'disassembly', 'plocka i sär') är således att utgå från ett komplett maskinprogram, dvs. ettor och nollor, och dela upp detta i sekvenser av assemblerinstruktioner.

I detta kapitel använder vi även beskrivningar med ett programspråk med högre abstraktionsnivå än assemblerspråk. Detta språk har stora likheter med programspråket "C" men avsikten är inte att fullständigt och korrekt beskriva programspråket "C" i sig utan snarare illustrera kopplingen mellan programspråken.

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 speciella regler för hur assemblerprogrammet ska se ut. Programmet läses av assemblatorn, rad för rad. En rad vars första tecken är ett semikolon (;) tolkas helt och hållet som en kommentar, och assemblatorn ignorerar denna. Övriga rader i assemblerprogrammet delas in i maximalt 4 fält. Första fältet kan enbart användas för att markera ett *läge* (en symbol eller en "etikett"). Man väljer då ett *symboliskt namn* och kan därefter använda detta namn som exempelvis operand till instruktioner.

*Exempel 9.1 Assemblerprogrammets struktur*

| Symbolfält | Mnemonic eller direktiv | Operand | Kommentarsfält |
|---|---|---|---|
| | ORG | $20 | ; Program börjar |
| start: | LDA | $FB | ; Läs data |
| | STA | $FC | ; Skriv data |
| | JMP | start | ; Repetera... |

| | | | |
|---|---|---|---|
| Då symbolfältet används måste symbolnamnet börja i radens första position. Kolon (:) kan, men behöver inte anges. | I detta fält ska instruktioner eller assemblerdirektiv anges | Om instruktionen/direktivet har någon operand, ska denna anges här. | Det sista fältet kan användas för kommentarstext. Det är lämpligt att markera detta med ett inledande semikolon, men det är inte nödvändigt. |

Man behöver inte ställa upp texten på något speciellt sätt, det räcker med att använda ett blanktecken som avdelare till nästa fält. För instruktioner, direktiv och hexadecimala numeriska värden kan man använda både gemena och versaler, följande kod är alltså helt likvärdig med den ovanstående:

```
 org $20 ; Program börjar
start: lda $fb ; Läs data
 sta $fc ; Skriv data
 jmp start ; Repetera...
```

### *Symboler*

Varje symbolnamn måste väljas *unikt* dvs, får bara definieras *en* gång i programmet. Symbolnamnet får vara högst 128 tecken långt. Symbolens *första* tecken måste vara en bokstav (a-z eller A-Z) *eller* en "understrykning", därefter kan symbolnamnet dessutom innehålla något av tecknen . "punkt" eller $ "dollartecken". Observera att de svenska tecknen å,ä och ö *inte* får användas i symbolnamn. Små respektive stora bokstäver betraktas som *olika* i symbolnamn.

*Exempel 9.2 Symbolnamn*

Följande symbolnamn kan definieras i samma program:

```
start
Start
```

Det är dock olämpligt att göra så eftersom det lätt kan skapa förvirring hos den som läser programmet. Följande utgör ytterligare exempel på tillåtna symbolnamn:

```
_prog10
stopp
prog10
prog_1
prog_1.1
prog$1
```

Följande ger exempel på otillåtna symbolnamn:

```
.prog              ; symbolnamn får INTE inledas med "punkt"
$prog              ; symbolnamn får INTE inledas med "dollar"
Hållplats          ; symbolnamn får INTE innehålla å,ä eller ö.
```

Då en symbol används (refereras) *innan* den definierats kallas detta *framåtreferens*. Detta är vanligtvis problemfritt men det finns några enskilda undantag då framåtreferenser inte kan tillåtas. Vi återkommer till detta.

*Exempel 9.3 Symbolreferenser*

Då vi använder en symbol som operand eller i ett uttryck kallas detta *referens*:

```
                ORG     $20
start:          LDA     $FB     ; Symbolen 'start' definieras på denna rad
                STA     $FC
                JMP     start   ; Referens av symbolen 'start'
```

Följande visar hur symbolen kan refereras innan den definierats, *framåtreferens*:

```
                JMP     next    ; Referens av symbolen 'next'
                STA     $FC
next:           ...             ; Symbolen 'next' definieras på denna rad
```

Vi har nu sett exempel på hur symboler kan anges för att representera lägen (dvs. adresser) i programmet. Symboler är dock generellt användbara och kan också användas för att representera exempelvis datakonstanter, detta illustreras bland annat i Exempel 9.5 nedan.





### *Talprefix och uttryck*

Konstanter som anges med binär eller hexadecimal talbas föregås av ett *prefix*. Inget prefix används för att ange konstanter i det decimala talsystemet. Observera att teckensträngen inte får innehålla några blanktecken eller, för talbasen, andra ogiltiga tecken. Även ASCII-tecken kan användas som en numerisk konstant, då ersätts detta med det värde som ges av ASCII-tabellen.

| | |
|---|---|
| Decimal konstant | Decimala konstanter anges utan prefix, siffrorna [0..9] används. |
| Binär konstant | Binära konstanter anges med ett procenttecken som prefix (%), endast siffrorna 0 och 1 kan användas i talet. |
| Hexadecimal konstant | Hexadecimala konstanter anges med ett dollartecken som prefix ($), siffrorna [0..9] och tecknen [a..f, A..F] kan användas i talet. |
| Teckenkonstant | Ett enskilt ASCII-tecken som omges av enkla citationstecken.<br>Skrivsättet 'a' anger exempelvis ASCII-värdet $61_{16}$, för gemena A. |

TABELL 9.1 PREFIX FÖR TALBASER

*Exempel 9.4 Talprefix*

Det decimala talet 11, är entydigt så länge inga andra talbaser är aktuella. För att entydigt ange talet i närvaro av andra talbaser, skriver vi $11_{10}$. Samma talvärde uttrycks i det binära talsystemet som $1011_2$, och i det hexadeximala talsystemet som $B_{16}$. Alltså gäller det att:

$11 = 1011_2 = B_{16}$

För att förenkla skrivsättet i en assemblerkälltext använder vi i stället prefix, och kan då i stället enklare skriva:

```
11 = %1011 = $B
```

Ett *uttryck* är en beskrivning av ett numeriskt värde. Den enklaste formen av ett uttryck är alltså en konstant i någon definierad talbas. Även symboler, vilka omedelbart representerar något numeriskt värde, kan utgöra grundläggande element i ett uttryck. Symboler som används för lägesidentifiering (adresser) kan också ofta användas som grundläggande element i uttryck. Man skiljer då på symboler vars numeriska värde är bestämt från början (absolut) respektive symboler som representerar adresser vilka kan komma att ändras, sådana symboler kallas *relokerbara*.

*Exempel 9.5 Absoluta och relokerbara symboler*

Absoluta symboler kan definieras med assemblerdirektivet `EQU` (equate). Innebörden är att symbolen ges ett värde som sedan inte kan ändras. Värdet kan ges i form av ett uttryck men det måste kunna bestämmas omedelbart och kan därför inte innehålla relokerbara symboler.

| Symbolfält | Mnemonic eller direktiv | Operand | Kommentarsfält |
|---|---|---|---|
| | ORG | $20 | ; Program börjar |
| InPort | EQU | $FB | |
| UtPort | EQU | $FC | |
| start: | LDA | InPort | ; Läs data |
| | STA | UtPort | ; Skriv data |
| | JMP | start | ; Repetera... |

Det numeriska värdet för symbolen `start` bestäms av ORG-direktivet och eventuella instruktionskoder eller data som placerats före symbolen. Symbolens värde ändras om man exempelvis skjuter in nya instruktioner mellan ORG och symbolen, detta har ingen betydelse för användningen av `start` och och symbolen sägs därför vara relokerbar. Symbolerna `InPort` respektive `UtPort` representerar fasta port-adresser i minnet, dessa kan inte ändras och symbolerna sägs därför vara absoluta.

Assemblatorn tillåter en rad olika operatorer för att forma uttryck. De olika operatorerna kan delas in i olika grupper beroende på funktion.

***Aritmetikoperatorer***

Aritmetikoperatorer används för att representera de fyra räknesätten. Eftersom heltalsdivision faller i två olika operationer (heltalsdivision och restdivision) finns totalt fem olika aritmetikoperatorer:

| Operator | Betydelse |
|---|---|
| `A+B` | Aritmetisk summa av A och B. |
| `A-B` | Aritmetisk skillnad mellan A och B. |
| `A*B` | Aritmetisk produkt av A och B. |
| `A/B` | Resulterande heltalsdel efter division, "heltalsdivision", (*integer division*). |
| `A%B` | Resulterande bråktalsdel efter division, "restdivision", (*modulus division*). |

Addition, subtraktion och multiplikation kan bara resultera i heltal så länge de ingående operanderna är heltal. Detta gäller inte division och man definierar därför två olika heltalsdivisioner.

*Exempel 9.6 Heltalsdivision och restdivision*

Bestäm A/B, respektive A%B, om A=10 och B=3.

Lösning:

$\frac{10}{3}=3+\frac{1}{3}$ varför A/B=3 och A%B=1

***Bitvis logikoperatorer***

Används för att manipulera enstaka bitar hos operanden.

| Operator | Betydelse |
|---|---|
| `~A` | Bitvis komplement av A. |
| `A&B` | Bitvis logiskt ”OCH” av A och B. |
| `A\|B` | Bitvis logiskt ”ELLER av A och B”. |
| `A^B` | Bitvis ”EXKLUSIVT ELLER” av A och B. |

*Exempel 9.7 Bitvis operationer*

Bitvis operatorer används för att manipulera enstaka bitar av en operand, några exempel på användning är:

Bitvis komplement: $\sim(11001010)_2 = (00110101)_2$

Bitvis OCH: $(11101110)_2 \;\&\; (11110000)_2 = (11100000)_2$

Bitvis ELLER: $(10100000)_2 \mid (11010011)_2 = (11110011)_2$

Bitvis EXKLUSIVT ELLER: $(01011100)_2 \wedge 00111100)_2 = (01100000)_2$





***Skiftoperatorer***

Skiftoperationer används för att manipulera enstaka bitar men kan också ses som aritmetikoperationer (division eller multiplikation). Assemblatorn använder inte någon typinformation om de ingående operanderna. Det finns därför bara två skiftoperatorer, vänster och höger, båda utförs som ”logiska skift”. Operand B, som anger antalet positioner som ska skiftas, förutsätts alltid vara ett positivt tal. Om B är större än antalet bitar i A kommer resultatet alltid att bli 0 eftersom nollor alltid skiftas in i den ”nya” positionen vid logiska skift.

| Operator | Betydelse |
|---|---|
| `A<<B` | Operand A skiftas B steg till vänster. |
| `A>>B` | Operand A skiftas B steg till höger. |

*Exempel 9.8 Skiftoperationer*

Operationerna A<<B och A>>B, ger samma bitmönster som resultat oavsett om vi tolkar talet A med eller utan tecken. Man får vara uppmärksam på de olika spillsituationerna som kan uppstå.

För vänsterskift uppstår spill om den mest signifikanta biten (före skiftet) är 1:

$4 \;<<\; 1 = (00000100)_2 << 1 = (00001000)_2 = 8$

$135 << 1 = (10000111)_2 << 1 = (00001110)_2 = 14 \quad (spill)$

Vid högerskift blir resultatet fel om A tolkas som tal med tecken och dessutom är mindre än 0:

$4 \;>>\; 1 = (00000100)_2 >> 1 = (00000100)_2 = 2$

$-4 >> 1 = (11111100)_2 << 1 = (01111100)_2 = 124 \quad (spill)$

Utöver aritmetik-, skift- och logikoperatorer finns det operatorer vars resultat utgörs av ett ”sanningsvärde”. Ett uttrycks logikvärde kan vara 1 (*sant*) eller 0 (*falskt*). Dessa operatorer kan delas in i två grupper, relations- och logikoperatorer.

***Relationsoperatorer***

| Operator | Betydelse |
|---|---|
| `A==B` | Uttryckets värde är 1 om A och B är lika, 0 annars. |
| `A!=B` | Uttryckets värde är 1 om A och B är olika, 0 annars. |
| `A<=B` | Uttryckets värde är 1 om A är mindre än, eller lika med B, 0 annars. |
| `A>=B` | Uttryckets värde är 1 om A är större än, eller lika med B, 0 annars. |
| `A<B` | Uttryckets värde är 1 om A är mindre än B, 0 annars. |
| `A>B` | Uttryckets värde är 1 om A är större än B, 0 annars. |

*Exempel 9.9 Relationsoperatorer*

Följande tabell anger relationsoperatorernas resulterande värde för några olika operander A och B

| A | B | A==B | A!=B | A<=B | A>=B | A<B | A>B |
|---|---|---|---|---|---|---|---|
| 32 | 17 | 0 | 1 | 0 | 1 | 0 | 1 |
| 32 | 32 | 1 | 0 | 1 | 1 | 0 | 0 |
| 16 | 17 | 0 | 1 | 1 | 0 | 1 | 0 |

### *Logikoperatorer*

Logikoperatorer kan också användas för att bestämma sanningsvärden. Dessa operatorer ska inte förväxlas med de bitvisa logikoperatorerna vi tidigare har behandlat.

| Operator | Betydelse |
|---|---|
| `!A` | Uttryckets värde är 0 om A är 0, i övriga fall är uttryckets värde 1. |
| `A&&B` | Uttryckets värde är 0 om A någon av B är 0, i övriga fall är uttryckets värde 1. |
| `A\|\|B` | Uttryckets värde är 0 om A och B båda är 0, i övriga fall är uttryckets värde 1. |

*Exempel 9.10 Logikoperatorer*

Följande tabell anger logikoperatorernas resulterande värde för några olika operander A och B

| A | B | !A | A&&B | A\|\|B |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 0 | 17 | 1 | 0 | 1 |
| 32 | 0 | 0 | 0 | 1 |
| 32 | 17 | 0 | 1 | 1 |
| 32 | 32 | 0 | 1 | 1 |
| 16 | 17 | 0 | 1 | 1 |

### *Instruktioner och assemblerdirektiv*

Det andra fältet i en rad kan innehålla en *mnemonic* för någon processorinstruktion *eller* ett assemblerdirektiv. Assemblerdirektiv används för att instruera assemblatorn på olika sätt. Följande direktiv används tillsammans med FLISP:

| | |
|---|---|
| `ORG <Val>` | "ORIGIN": Anger startadress för påföljande kod/data. Om en symbol används för att ange startadressen måste symbolen vara definierad, dvs inga framåtreferenser är tillåtna här. |
| `Sym EQU <Val>` | "EQUATE": Symbolen 'Sym' representerar värdet <Val>. |
| `[Sym] FCB <Val>,<Val>...` | "FORM CONSTANT BYTE": Skapar en sträng med initierade data i minnet I detta fall kan ett symbolnamn anges, men det är inte nödvändigt. |
| `[Sym] FCS "<ASCII tecken>"` | "FORM CONSTANT STRING": Skapar en sträng med ASCII-tecken i minnet. Symbolnamn kan, men behöver inte, anges. |
| `[Sym] RMB <Val>` | "RESERVE MEMORY BYTES": Reservera <Val> bytes i minnet. Minnesinnehållet på dessa adresser är odefinierat. Symbolnamn kan, men behöver inte, anges. |

TABELL 9.2 ASSEMBLERDIREKTIV FÖR FLISP





*Exempel 9.11 Användning av assemblerdirektiv*

Följande exempel visar hur direktiven FCB, RMB och FCS kan användas för att reservera minnesutrymme och även initiera minnet med dess innehåll.

| Adress | Innehåll | Assemblerkod (disassemblering) | |
|---|---|---|---|
| | | `ORG` | `$20` |
| 20 | 38 | `FCB` | `$38` |
| 21 | 98 | `FCB` | `%10011000` |
| 22 | | `RMB` | `1` |
| 23 | 41 | `FCB` | `´A´` |
| 24 | 61 | `FCS` | `"abcd"` |
| 25 | 62 | | |
| 26 | 63 | | |
| 27 | 64 | | |
| 28 | | | |

Notera hur vi utelämnat minnesinnehåll på adress 22 eftersom direktivet endast innebär att 1 byte reserveras här. Vi kan därför inte veta dess värde.

Assemlerdirektiven `FCB`, `FCS` och `RMB` används alltså för att upplåta minnesutrymme vars adresser sedan inte kan ändras. Detta kallas *statisk* minnesallokering.

## 9.1 Enkla datatyper, lagringsklasser och synlighet

En variabeldeklaration används för att reservera minnesutrymme. Deklarationen anger ett unikt symbolnamn och för att kunna behandla symbolen korrekt i uttryck ges den också en distinkt "datatyp". Av datatypen framgår ofta vad symbolen är tänkt att representera, eller annorlunda sagt, vad man ska använda den till.

`char` är en beteckning på en datatyp som, i programspråket "C", ursprungligen implementerades med 8 bitar, motsvarande beteckning i "Java" är `byte`. Det är också med denna betydelse vi använder beteckningen `char` i denna text. *Unsigned* respektive *signed* är så kallade *typspecifierare* som anger hur en variabel deklarerad på detta sätt används vid jämförelseoperationer. Talområdet för en `unsigned char` är 0..255, medan talområdet för en `signed char` blir -128..127. Speciellt anger typspecifieraren vilken villkorlig branch-instruktion som ska användas efter en jämförelseoperation.

*Exempel 9.12 Enkla datadeklarationer, reservering av minnesutrymme*

En variabeldeklaration:

```
unsigned char counter;
```

anger att en *byte* ska reserveras för symbolen `counter`, då den används i jämförelseoperationer ska den betraktas som ett 8-bitars tal utan inbyggt tecken. I assemblerspråk motsvarar deklarationen:

```
counter:   RMB   1
```

Deklarationen:

```
signed char summa;
```

anger också att en byte ska reserveras, denna gång för symbolen summa, i jämförelseoperationer ska symbolen betraktas som ett 8-bitars tal med inbyggt tecken och i assemblerspråk motsvarar deklarationen:

```
summa:     RMB   1
```

Observera att direktivet RMB inte innehåller någon typinformation om minnesinnehållet utan bara anger storleken på det reserverade utrymmet.

Enkla datatyper kan sättas samman till *vektorer* i ett godtyckligt antal instanser av typen. I ”C” används exempelvis hakparenteser, där vektorns längd anges inom parenteserna, att just en vektor avses. Deklaration av en endimensionell vektor av typen **char**, dvs. godtyckligt antal tecken placerade konsekutivt i minnet har följande struktur:

```
char    symbolnamn[längd];
```

*symbolnamn* utgör den symboliska startadressen för vektorn, *längd* anger maximala antalet element (av typen **char**) i vektorn.

*Exempel 9.13 Vektorer i assemblerspråk*

En variabeldeklaration:

```
char char_array[8];
```

anger att 8 *bytes* ska reserveras för symbolen `char_array`. I assemblerspråk motsvaras deklarationen av:

```
char_array:     RMB   8
```

För att indexera i vektorn, dvs. referera något av vektorns element, anges något index, i detta fall 0 t.o.m 7. Observera speciellt att uppräkningen av elementen i vektorn alltid börjar med 0.

***Variablers synlighet och lagringsklasser***

En variabel sägs ha olika synlighet ("scope") beroende på i vilket sammanhang den är deklarerad. Variabler kan ha olika *lagringsklass*. Inledningsvis behandlar vi uteslutande *globala* variabler, *lokala* variabler behandlas längre fram i detta kapitel.

***Globala variabler***

En global variabel karaktäriseras av att den kan refereras (användas), av alla delar av programmet oavsett om det gäller huvudprogrammet eller någon subrutin. För deklaration av en global variabel ska därför något av direktiven `RMB`, `FCB` eller `FCS`, användas tillsammans med variabelns (unika) symbolnamn.

## 9.2 Programmeringsmodell

Beskrivningen av FLISP-datorns interna uppbyggnad (enligt kapitel 8) omfattar en stor mängd detaljer som är överflödiga då vi bara vill konstruera assemblerprogram för datorn. Vi använder i stället här en en beskrivningsnivå med de detaljer som är nödvändiga för assemblerprogrammeraren.

Med "assemblerprogrammerarens bild" av FLISP avses:

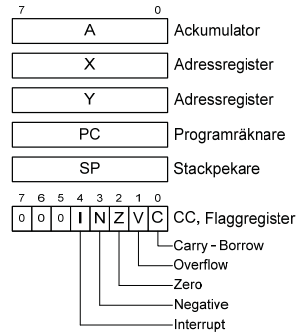
- register synliga för programmeraren
- primärminnets användning
- instruktionsuppsättning och adresseringssätt
- undantagshantering

***Programmerarens register***

I Figur 9.1 visas FLISP's registeruppsättning.








FIGUR 9.1 REGISTER SYNLIGA FÖR PROGRAMMERAREN

De olika registren är avsedda för speciella ändamål. Ackumulatorregistret (A) används vid aritmetik-/logik- och skiftperationer för att ”ackumulera” resultatet från operationerna. Registret är därför en ”implicit” operand vid binära operationer, samtidigt som resultatet från operationen återförs till registret, ”ackumuleras”.

Adressregistren (X och Y) används för att bestämma adresser till data i de fall de måste beräknas på något sätt. Eftersom vanliga operationer i en dator inbegriper såväl en källoperand som en destinationsoperand har FLISP utrustats med två likvärdiga adressregister.

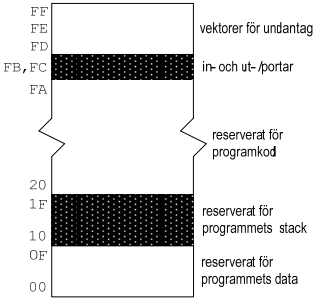
Programräknaren (PC) manipuleras ytterst sällan direkt av assemblerprogrammeraren, den ingår dock som adresseringssätt och är därför en självskriven komponent i programmerarens bild.

Delar av minnet måste anvisas för temporär lagring av återhoppsadresser vid subrutinanrop, undanlagring av registerinnehåll vid undantagshantering, temporär undanlagring styrd av programmeraren, etc. Stackpekaren (SP) används för sådana ändamål.

I Figur 9.1 finns även ett register med flaggbitar (CC). Många instruktioner laddar nya värden på flaggbitarna i flaggregistret. Normalt är det flaggvärdena från ALU:n som laddas i flaggregistret vid någon ALU-operation. De innehåller alltså information om resultatet av den senaste ALU-operationen som påverkade flaggregistret. Flaggbitarna används till exempel när villkorliga instruktioner skall utföras av processorn. Den speciella flaggan *interrupt-mask* (I) används endast vid *undantagshantering* som vi återkommer till.

***Primärminnets användning***

Bortsett från adresser avdelade för undantagshantering och in-/utmatning kan primärminnet användas på praktiskt taget godtyckligt sätt. Det är dock praktiskt att införa lämpliga konventioner och sedan vara så konsekvent det är möjligt. Den rekommenderade användningen av primärminnet hos FLISP-datorn framgår av Figur 9.2.




FIGUR 9.2 ANVÄNDNING AV PRIMÄRMINNET HOS FLISP

Observera de begränsningar som gäller. Programmet kan inte vara större än 218 bytes ($FA_{16}$-$20_{16}$=$DA_{16}$=218). Figuren antyder att det förutsätts att stacken aldrig växer mer än 16 bytes och att totala utrymmet för globala variabler inte överstiger 16 bytes. Egentligen är dock restriktionen att utrymme för globala variabler och stack tillsammans inte får överstiga 32 bytes eftersom de globala variablerna reserveras med start på adress 0 i minnet och stacken börjar på adress $20_{16}$ för att därefter "växa nedåt" i minnet.

*Exempel 9.14 Organisation av ett enkelt program för FLISP*

Med konventionerna för primärminnets användning kan vi skapa ett första ”skelett” för applikationsprogram för FLISP.

```
            ORG         0
; deklarationer av globala variabler följer här...
;      ...
            ORG         $20
; programkod följer här...
start:      LDSP        #$20
            ...
            ...
            ORG         $FF
            FCB         start       ; RESET-vektor
```

## 9.3 Instruktionsuppsättning för FLIS-processorn

En processors instruktionsuppsättning är sammansatt av intruktioner avsedda att utföra uppgifter av generell typ. Ett maskinprogram skall kunna konstrueras utgående från en algoritmisk beskrivning av hur en uppgift utförs. Maskinprogrammet i sig är också en algoritmisk beskrivning. Processorn skall därför ha instruktioner med vilka man kan uttrycka konstruktionerna *sekvens*, *selektion* och *iteration*, som är nödvändiga för utförande av de flesta algoritmer.

Instruktionsuppsättningen är ofta ganska primitiv, i synnerhet när det gäller att beräkna uttryck. I enkla processorer saknas exempelvis ofta instruktioner för operationer som multiplikation och division. Lite mer komplexa operationer implementeras då istället som följder av processorns primitiva operationer. När det gäller instruktionsuppsättningen skall programmeraren ges en god bild av vilka *operationer* som erbjuds. En betraktelse av processorns instruktionslista ger vid handen att merparten av instruktionerna tillhör någon av följande kategorier:

- datakopiering
- databearbetning, aritmetik- och logik- operationer
- jämförelser och tester
- programflödesändring

De två första används för att forma sekvenser, dvs göra tilldelningar och beräkna uttryck. De två sista används huvudsakligen för att uttrycka selektioner och iterationer dvs. skapa villkorliga och ovillkorliga programflöden.

Instruktioners operander kan anges som ett *läge* med ett *adresseringssätt*, De tre första kategorierna måste på något sätt specificera lägen för både operander och resultat. Dessa lägen kan vara interna processorregister eller externa adresser i primärminnet. I den sista operationskategorin måste den adress specificeras på vilken programexekveringen efter ett hopp skall fortsätta.

Programmeraren måste ges en god bild av hur ett operandläge eller en hoppadress kan specificeras via instruktioner. Specificeringen kan ske på ett flertal olika sätt. Processorn sägs ha olika *adresseringsätt*. Dessa framgår också av instruktionslistan. I texten illustreras och diskuteras adresseringssätten efter hand.





### 9.3.1 Instruktioner för datakopiering

FLISP-instruktioner för datakopiering förekommer i fyra olika varianter:

- *Load*, ladda ett register med data från minnet
- *Store*, kopiera data från ett register till minnet
- *Transfer*, kopiera data från ett register till ett annat
- *Exchange*, Skifta innehållet mellan två register

***Load***

Load-instruktionen kopierar data från minnet till något register. Det finns flera varianter och adresseringssätt för denna instruktion. Alla register kan laddas med någon load-instruktion men de tillgängliga adresseringssätten skiljer sig. De mest generella sätt att ange en operand gäller endast för LDA-instruktionen, medan övriga register kan använda de vanliga adresseringssätten.

För *load*-instruktionen gäller allmänt:

| | |
|---|---|
| RTN | M (EA)→ R<br>här anger M(EA) helt enkelt data på minnesposition som bestäms av det aktuella adresseringssättet.<br>R avser registret som anges av load-instruktionen, dvs. LDA, LDX, LDY eller LDSP. |
| Flaggor | N: Ettställs om resultatets teckenbit (bit 7) får värdet 1.<br>Z: Ettställs om samtliga åtta bitar i resultatet blir noll.<br>V: Nollställs.<br>C: Påverkas ej |
| Beskrivning | Laddar dataord från minnet till angivet register R (A,X,Y eller SP) |

***Store***

*Store*-instruktionen används för att kopiera data från något register till minnet, även här kan olika adresseringssätt förekomma och STA-instruktionen har flest olika adresseringsmöjligheter, tills vidare nöjer vi oss med att introducera de viktigaste. Allmänt gäller för *store*-instruktionen:

| | |
|---|---|
| RTN | R → EA<br>här anger EA en minnesposition som bestäms av det aktuella adresseringssättet.Till denna minnesposition kopieras data från registret.<br>R avser registret som anges av store-instruktionen dvs. STA, STX, STY eller STSP. |
| Flaggor | N: Påverkas ej.<br>Z: Påverkas ej.<br>V: Påverkas ej.<br>C: Påverkas ej |
| Beskrivning | Lagrar angivet registerinnehåll R (A,X,Y eller SP) i minnet på den effektiva adressen |

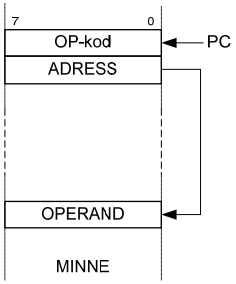
De enklaste varianterna av instruktionerna framgår av följande tabeller:

| Instruktion **ST** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| STA Adr | Absolute | E1 | 2 | 3 | A → M(Adr) | - | - | - | - |

| Instruktion **LD** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| LDA #Data | Immediate | F0 | 2 | 2 | Data → A | Δ | Δ | 0 | - |
| LDA Adr | Absolute | F1 | 2 | 3 | M(Adr) → A | | | | |

***Absolut adressering* (Adr)**

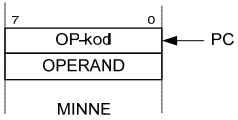
Absolut adressering används typiskt för variabler där minnesadressen är statiskt bestämd (om så som en symbol) redan vid översättningen av programmet (se även Exempel 9.12).




***Omedelbar adressering* (#Data)**

För att ange konstanter används adresseringssättet "omedelbar" (*immediate*), konstanten är placerad omedelbart efter operationskoden i minnet.

RTN: M(PC+1)
Assemblersyntax: #<DATA>




Låt oss nu, med några exempel belysa hur de olika adresseringssätten är avsedda att användas.

*Exempel 9.15 Tilldelningar av konstant till variabel*

Antag att vi har följande globala variabel:

```
char index;
```

I assemblerspråk motsvaras deklarationen av:

```
index:      RMB   1
```

Vi vill göra följande tilldelning till denna variabel:

```
index = 1;
```

för detta exempel finns det fler alternativ:

```
LDA   #1
STA   index
```

eller:

```
LDX   #1
STX   index
```

eller:

```
LDY   #1
STY   index
```

hade fungerat lika bra.





*Exempel 9.16 Tilldelning av variabel från variabel*

Antag att vi har följande globala variabler:

```
char index;
char counter;
```

I assemblerspråk motsvaras deklarationerna av:

```
index:      RMB   1
counter:    RMB   1
```

Exempelvis kan man nu koda tilldelningen:

```
counter = index;
```

som:

```
LDA   index
STA   counter
```

***Transfer***

TFR, datakopiering av register. Med en "transfer"-instruktion kopieras ett ord ifrån ett av processorns register till ett annat. Det register varifrån data hämtas kallas allmänt för *källregister*, och det register data kopieras till kallas *destinationsregister*. RTN-beskrivningen för operationen är:

$R_{källregister} \rightarrow R_{destinationsregister}$

Operationen påverkar inte flaggorna såvida inte flaggregistret är destinationsregister.

| Instruktion **TFR** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| TFR A,CC | Inherent | 18 | 1 | 2 | A → CC | Δ | Δ | Δ | Δ |
| TFR CC,A | Inherent | 19 | 1 | 2 | CC → A | - | - | - | - |
| TFR X,Y | Inherent | 1A | 1 | 2 | X → Y | - | - | - | - |
| TFR Y,X | Inherent | 1B | 1 | 2 | Y → X | - | - | - | - |
| TFR X,SP | Inherent | 1C | 1 | 2 | X → SP | - | - | - | - |
| TFR SP,X | Inherent | 1D | 1 | 2 | SP → X | - | - | - | - |
| TFR Y,SP | Inherent | 1E | 1 | 2 | Y → SP | - | - | - | - |
| TFR SP,Y | Inherent | 1F | 1 | 2 | SP → Y | - | - | - | - |

***Exchange***

EXG, data "utbytes" mellan två register, dvs. innehållen skiftas mellan registren. RTN-beskrivningen för operationen är:

$R_1 \leftrightarrow R_2$

Operationen påverkar flaggorna enbart om flaggregistret ingår.

| Instruktion **EXG** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| EXG A,CC | Inherent | 9F | 1 | 4 | A ↔ CC | Δ | Δ | Δ | Δ |
| EXG X,Y | Inherent | AF | 1 | 4 | X ↔ Y | - | - | - | - |
| EXG X,SP | Inherent | BF | 1 | 4 | X ↔ SP | - | - | - | - |
| EXG Y,SP | Inherent | CF | 1 | 4 | Y ↔ SP | - | - | - | - |

### 9.3.2 Bearbetning av data, aritmetik och logik

I detta avsnitt beskrivs instruktioner som utför aritmetik- resp. logikoperationer. Instruktionena används uteslutande för evaluering av uttryck. Ett uttryck sätts samman av konstanter, variabler och operatorer. Resultatet av ett uttryck är alltid ett värde (evaluerat uttryck) som antingen kan användas för att tilldelas någon variabel eller användas som test för någon villkorlig operation. Skillnaden är liten, i det första fallet sparas resultatet i någon minnescell (tilldelning) i det andra fallet testas bara det evaluerade uttryckets värde för att bestämma något programflöde. Vi behandlar nu uttrycksevaluering och tilldelningar.

***Aritmetikinstruktioner***

FLIS-processorn har instruktioner för addition och subtraktion. För att utföra dessa operationer på 8-bitars datatyp (`char`) används instruktionerna `ADDA` respektive `SUBA`. För att utföra operationerna på större ordbredd, måste varianterna med ackumulerad carrybit (`ADCA` och `SBCA`) användas.

*Exempel 9.17 Uttrycksevaluering*

Uttryck används ofta tillsammans med tilldelningar. Följande satser visar hur ett uttryck (högerledet) leder till ett evaluerat resultat, som tilldelas en variabel. Antag deklarationerna:

```
char  sum,op1,op2,op3;
```

motsvarande assemblerdeklarationer blir:

```
      ORG   0
sum   RMB   1
op1   RMB   1
op2   RMB   1
op3   RMB   1
```

Tilldelningen:

```
      sum = op1+op2-op3;
```

kan då kodas:

```
      LDA   op1
      ADDA  op2
      SUBA  op3
      STA   sum
```

***Logikinstruktioner***

Processorn har instruktioner för att bitvis bilda AND, OR och EXCLUSIVE-OR mellan operandernas bitar. När en operation utförs bitvis finns ingen koppling i sidled mellan operandernas bitar utan varje resultatbit är enbart en funktion av motsvarande bitar i de två operanderna

*Exempel 9.18 Bitvis AND, OR och EOR*

En enskild bit kan ettställas med en OR-operation och nollställas med en AND-operation. Vid ettställning bildar man typiskt en ”bitmask”, med ’1’ för de bitar som ska ettställas och ’0’ för de bitar som inte ska påverkas. utför logiskt OR och återför detta resultat.

Om register A från början innehåller något värde X kan exempelvis varannan bit ettställas med instruktionen:

```
      ORA   #%10101010
```

Operationen som utförs är:

```
        xxxxxxxx
OR      10101010
        1x1x1x1x
```





Enskilda bitar inverteras med en bitmask där ’1’ innebär att motsvarande bit inverteras, medan ’0’ innebär att biten inte påverkas. Om exempelvis A innehåller värdet $11110000_2$ och följande instruktion utförs:

```
      EORA  #%10101010
```

utförs operationen:

```
        11110000
EOR     10101010
        01011010
```

Slutligen kan enskilda bitar nollställas genom att man i stället bildar en inverterad bitmask, i denna bitmask innebär ’1’ nu att motsvarande bit inte påverkas, medan ’0’ innebär att biten nollställs. Därefter utför man logiskt AND och återför resultatet.

Antag exempelvis att vi vill nollställa bitarna $b_0$, $b_1$, $b_2$ och $b_3$. Vi bildar då den inverterade bitmasken, det vill säga med ettor för de bitar vi inte vill påverka som,

```
        11110000
```

Om register A från början nu innehåller något värde Y kan bitarna nollställas med instruktionen:

```
      ANDA  #%11110000
```

Operationen som utförs är:

```
        yyyyyyyy
OR      11110000
        yyyy0000
```

### 9.3.3 IO-portar med bitfunktioner

Beteckningen "variabel" är bara användbar då minnesplatsen kan ändras godtyckligt av ett program utan att andra aspekter behöver beaktas. Detta är inte alltid sant. Adressen kan också utgöra en port (in- eller ut), även här använder man absolut adressering men till skillnad från en "variabel" kan man inte utgå från att minnespositionen, dvs. porten, är samtidigt läs- och skrivbar. Man kan därför inte förutsätta att det till en port sist skrivna värdet, är det samma som resulteras av en läsning från samma adress. I de fall man bara vill ändra enstaka bitar på en icke läsbar utport kan man införa en så kallad "skuggvariabel" vars syfte är att innehålla exakt samma värde som utporten hade haft om den varit läsbar. Bitmanipulation kan då först göras på skuggvariabeln, därefter kopieras skuggvariabelns värde till den icke läsbara utporten.

*Exempel 9.19 Användning av "skuggvariabel" för icke läsbara utportar*

FLISP's utport på adress $FB_{16}$ är inte läsbar, dvs. en LOAD-operation från denna adress resulterar alltid i värdet $FF_{16}$, oavsett vad som tidigare skrivits till porten. För att kunna ändra enstaka bitar kan man införa en "skuggvariabel", där bitoperationer kan utföras korrekt, och där resultatet sedan kopieras till utporten.

```
          ORG   0
shadow    RMB   1
          ORG   $20
...
; Ettställ bit 0
          LDA   shadow
          ORA   #1
          STA   shadow
          STA   $FB

; Nollställ bit 1
          LDA   shadow
          ANDA  #~1
          STA   shadow
          STA   $FB
```

### 9.3.4 Instruktioner för att förbättra prestanda

En operation ingående i ett uttryck kan vara *binär* eller *unär*. En unär operation kräver exakt en operand medan en binär operation alltid har två operander. Vanligt förekommande binära operationer kan ges separata instruktioner med unär form av prestandaskäl. I FLISP har vi exempelvis `CLR` ("clear"), `NEG` ("negate"), `DEC` ("decrement") och `INC` ("increment").

*Exempel 9.20 Instruktioner för att öka prestanda*

Tilldelningar av typen.

```
var = 0;
```

är vanliga, kodningen av en sådan tilldelning blir:

```
LDA   #0
STA   var
```

totalt omfattar kodningen 4 bytes men en ekvivalent kodning är

```
CLR   var
```

som upptar 2 bytes och ger en kortare total exekveringstid.

Som ytterligare exempel på sådana prestandahöjande instruktioner kan man ta:

```
var = -var;
```

dvs. teckenbyte på en variabel. Rättfram kodning blir:

```
LDA   #0
SUBA  var
STA   var
```

totalt 6 bytes, men samma operation utförs av instruktionen

```
NEG   var
```

vilken endast upptar 2 bytes.

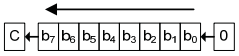
### 9.3.5 Skiftinstruktioner

FLISP stödjer tre olika typer av skift: logiskt skift, aritmetiskt skift och rotation. Såväl vänsterskift som högerskift kan utföras och totalt innebär detta då sex operationer. Logiskt skift är avsett för operationer på tal utan tecken. Aritmetiskt skift används för operationer på tal med tecken och rotation är avsett för olika bitmanipulationer. Eftersom logiskt och aritmetiskt vänsterskift är samma sak implementeras dessa som *en* operationskod. Assemblatorn tillåter dock mnemonics för båda instruktionerna.

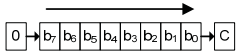
Man löper alltid en risk att tappa signifikanta siffror vid vänsterskift, på motsvarande sätt kan innehållet förlora precision (noggrannhet) vid högerskift.

***Logiskt skift***

`LSL`, *Logical Shift Left*, vänsterskift, multiplikation med 2, tal utan inbyggt tecken. C-flaggan anger om spill uppstod, dvs. förlorad signifikant bit.




`LSR`, *Logical Shift Right*, högerskift, division med 2, tal utan inbyggt tecken. C-flaggan anger om spill uppstod.(Förlorad precision).

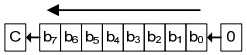




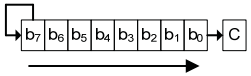



***Aritmetiskt skift***

`ASL`, *Arithmetic Shift Left*, vänsterskift, multiplikation med 2, tal med inbyggt tecken. C-flaggan anger om spill uppstod. Operationen är den samma som logiskt vänsterskift.



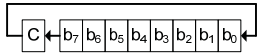

`ASR`, *Arithmetic Shift Right*, högerskift, division med 2, tal med inbyggt tecken. C-flaggan anger om spill uppstod.(Förlorad precision). Observera hur den mest signifikanta biten i talet behålls för att förhindra teckenbyte vid operationen.



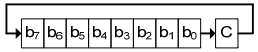

***Rotationskift***

Vid rotationsskift skiftas operandens bitar ett steg. Operandens två ändar sammanlänkas med C-flaggan. Detta innebär att C-flaggans värde skiftas in i en operandända och att den bit som skiftas ut i den andra ändan placeras i C-flaggan.

`ROL`, *ROtate Left*




`ROR`, *ROtate Right*




*Exempel 9.21 Kodning med skift-instruktioner*

Skiftoperationer väljs baserat på datatyp, antag deklarationerna:

```
unsigned   char  uop;
signed     char  sop;
```

motsvarande assemblerdeklarationer blir:

```
       ORG   0
uop    RMB   1
sop    RMB   1
```

satserna:

```
uop = uop << 1;
sop = sop << 1;
```

kodas:

```
LSL   uop
ASL   sop
```

satserna:

```
uop = uop >> 1;
sop = sop >> 1;
```

kodas:

```
LSR   uop
ASR   sop
```

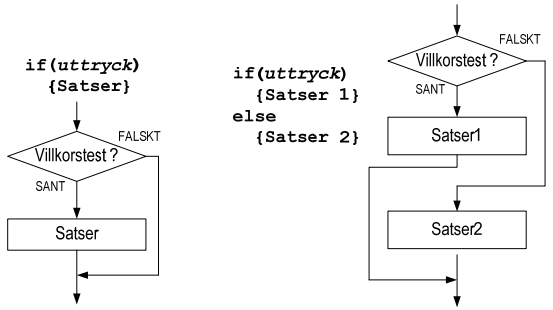
## 9.4 Styrning av programflöde

Ett programflöde kan styras av två principiellt olika konstruktioner, *selektion* eller *iteration*. I det första fallet utför programmet ett unikt val av (möjligtvis flera) alternativ medan i det senare fallet styrs programmet att upprepas (itereras), tills dess något villkor är uppfyllt.

En selektion- eller iterations- konstruktion är baserad på ett villkor. Ett villkor kan bara ha två värden, antingen sant eller falskt.

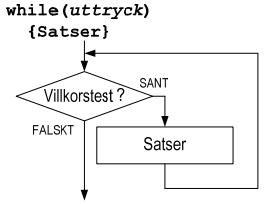
Villkoret är baserat på resultatet av ett uttryck, uttrycksevalueringen kan vara mer eller mindre omfattande men resultatet kan i detta fall vara antingen 0 och i så fall FALSKT, eller också är uttryckets värde skilt från 0 och i så fall SANT.

De enklaste selektionskonstruktionerna tillåter villkorlig exekvering av ett block med programkod.




FIGUR 9.3 TVÅ KONSTRUKTIONER FÖR SELEKTION

En vanlig iterationskonstruktion går ut på att ett block med programkod exekveras så länge ett villkor är uppfyllt.




FIGUR 9.4 ITERATIONSKONSTRUKTION

Ett uttryck som används i villkorstest skapas temporärt. Kodningen av uttrycket sker på samma sätt som exempelvis tilldelningssatser men resultatet, dvs. det evaluerade uttrycket används sedan i en test- eller jämförelseoperation, snarare än att tilldelas någon variabel.

### 9.4.1 Tester

Selektions- och iterationskonstruktioner är baserade på villkor. På processornivå är dessa typiskt formade ur undersökningen av en operand eller resultatet av ett beräknat uttryck. För att just undersöka en operand har processorn en eller flera testinstruktioner. Dessa operationer har vanligtvis två operander, en som är föremål för undersökning och en annan som den testas mot för att undersökningen skall ge resultat. Tre typer av testinstruktioner, *compare*, *test* och *bittest*, är vanliga, de kännetecknas av att de endast påverkar flaggregistret.





***Jämförelse med operand***

Instruktionen `CMP` (compare) använder subtraktion för att avgöra om en operand är mindre än, lika med eller större än en annan operand. Resultatet finns i flaggregistret.

**CMP Compare register and data**

| | | |
|---|---|---|
| RTN | R – Opr | där R = A, X, Y eller SP |
| Flaggor | N: | Får värdet hos skillnadens teckenbit (bit 7). |
| | Z: | Ettställs om skillnaden blir noll. |
| | V: | Ettställs om 2-komplementoverflow uppstår vid subtraktionen |
| | C: | Ettställs om borrow uppstår vid subtraktionen. |
| Beskrivning | Operanden subtraheras från innehållet i det angivna registret. Skillnaden lagras ej, utan påverkar endast flaggorna. | |

*Exempel 9.22*

Om ackumulatorn innehåller ett uppmätt temperaturvärde $12_{16}$ (18 °C) så utför instruktionen

```
CMPA  #20
```

operationen

```
  00010010      innehåll i A
- 00010100      jämförelsevärde
  11111110      skillnad (sparas ej)
```

flaggvärdestilldelningen blir:

1→C 0→V 0→Z 1→N

Operationen jämför alltså det uppmätta värdet med ett känt referensvärde 20 °C. Detta kan exempelvis göras i avsikt att starta uppvärmning när N-flaggan blir "1".

***Jämförelse med noll***

Instruktionen `TST` (*test*) är det specialfall av `CMP` då operanden är konstanten 0. Instruktionen subtraherar noll ifrån operanden för att avgöra om den är noll, positiv eller negativ. Resultatet finns i flaggregistret, skillnaden bevaras ej.

**TST Test**

| | | |
|---|---|---|
| RTN | A – 0 eller M – 0 | M = data från minnesadress |
| Flaggor | N: | Ettställs om resultatets teckenbit (bit 7) får värdet 1. |
| | Z: | Ettställs om samtliga åtta bitar i resultatet blir noll. |
| | V: | Nollställs. |
| | C: | Nollställs |
| Beskrivning | Låter datavärdet i A eller M passera ALU:n och sätter flaggvipporna N och Z så att man kan avgöra datavärdets tecken eller om det är noll. Endast flaggvipporna påverkas. | |

*Exempel 9.23*

Om ackumulatorn innehåller värdet $45_{16}$ så utför instruktionen `TSTA` operationen

```
  01000101   innehåll i A
- 00000000   jämförelsevärde
  01000101   skillnad (sparas ej)
```

och ger flaggorna värdena

0→C 0→V 0→Z 0→N

utan att påverka något annat registers innehåll.

I båda dessa jämförelseoperationer betraktas operanden som ett numeriskt värde som jämförs med ett testobjekt. Relationerna är beroende av om operand och testobjekt ses som tal med eller utan tecken. Helt allmänt gäller att relationen mellan operand och jämförelseobjekt kan uttryckas med flaggor enligt Tabell 9.3.

| relation | tal utan tecken | tal med tecken |
|---|---|---|
| > | C'·Z' | (N⊕V)'·Z' |
| ≥ | C' | (N⊕V)' |
| = | Z | Z |
| ≠ | Z' | Z' |
| ≤ | (C'·Z')' = C+Z | ((N⊕V)'·Z')' = (N⊕V)+Z |
| < | C | (N⊕V) |

TABELL 9.3

Flaggvärdena är alltså resultat av subtraktioner och lånebiten $b_8$ finns då i C-flaggan. Vid tal utan tecken avgörs relationerna av C- och Z-flaggorna.

Tal med tecken har 2-komplementrepresentation och tillhör intervallet [-128, +127]. Eftersom talen nu har teckenbit måste flaggorna N, V och Z användas för att bestämma storleksrelationerna. Vi vet att när "overflow" inträffar vid subtraktion upptäcks detta genom att teckenbiten N får fel värde. Trots detta kan man skapa en korrekt teckenbit, som gäller vare sig "overflow" inträffar eller ej, genom att bilda N⊕V. Om overflow inträffar vet man ju att teckenbiten N får fel värde. Eftersom V samtidigt får värdet 1 inverterar man N genom att bilda N⊕V (N⊕1 = N'). Om overflow inte inträffar ger N⊕V värdet N eftersom N⊕0 = N.

***Test av operandbitar***

I den tredje testoperationen betraktas operanden ej som ett värde utan enbart som ett bitmönster.

### BITA Bit test register A

| | |
|---|---|
| RTN: | A ∧ Opr |
| Flaggor: | N: Ettställs om resultatets teckenbit (bit 7) får värdet 1.<br>Z: Ettställs om samtliga åtta bitar i resultatet blir noll.<br>V: Nollställs.<br>C: Påverkas ej |
| Beskrivning: | Utför bitvis AND-operation mellan dataordet i minnet och innehållet i register A. Resultatet lagras ej, utan påverkar endast flaggorna |

`BITA` är en instruktion som utför logiskt OCH mellan bitarna i en operand och bitarna i ett testmönster, en mask. Testet resulterar enbart i påverkan av N- och Z-flaggorna. Detta är huvudsakligen ett sätt att med en mask ta reda på om alla bitarna i den icke maskerade bitgruppen av operanden är noll eller ej. Vanligen används denna operation för att undersöka en av operandens bitar genom att maskera de övriga bitarna. Denna operation erbjuder således ett enklare och bättre sätt att undersöka individuella bitar än att göra det med skiftoperationer och C-flaggan.

*Exempel 9.24*

Bit 3 av ackumulator A undersöks med `BITA #$08`

```
    x7 x6 x5 x4 x3 x2 x1 x0      operand i A
∧   0  0  0  0  1  0  0  0       mask
    0  0  0  0  x3 0  0  0       logisk produkt (sparas ej)
```

N-flaggan får värdet

N = 0

Z- flaggan får värdet

Z = 1 om $x_3 = 0$

eller

Z = 0 om $x_3 = 1$





## 9.4.2 Ändring av programflödet

Denna kategori av instruktioner är nödvändiga för selektions- och iterationskonstruktioner. I båda dessa fall måste en rak exekveringsföljd kunna brytas upp i ett "vägval". I ett maskinprogram görs detta med någon instruktion som åstadkommer en programflödesändring dvs. när programräknaren laddas med en ny adress som bryter det sekventiella instruktionsflödet. En programflödesändring kan vara villkorlig eller ovillkorlig.

Det finns två olika instruktioner för ovillkorlig programflödesändring i FLISP, `BRA` (*branch always*) och `JMP` (*jump*), medan operationen är den samma skiljer de sig åt i olika adresseringssätt.

### BRA Branch always

| | |
|---|---|
| RTN | PC+Offset → PC |
| Flaggor | Påverkas ej |
| Beskrivning | Ett hopp utförs till adressen ADRESS = PC+Offset. Offset räknas från adressen efter branchinstruktionen, dvs vid uträkningen av hoppadressen pekar PC på operationskoden som (eventuellt) finns direkt efter branchinstruktionen i minnet |

Detaljer:

| Instruktion **BRA** | Adressering | | | | Operation | Flaggor | | | |
|---|---|---|---|---|---|---|---|---|---|
| | metod | OP | # | ~ | | N | Z | V | C |
| `BRA Adr` | Relativ | 21 | 2 | 4 | PC+Offset → PC | - | - | - | - |

### JMP Jump

| | |
|---|---|
| RTN | EA → PC |
| Flaggor | Påverkas ej |
| Beskrivning | Ovillkorlig programflödesändring, nästa instruktion hämtas från effektiva adressen EA. |

Detaljer (ej fullständigt):

| Instruktion **JMP** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| `JMP Adr` | Absolute | 33 | 2 | 2 | Adr → PC | - | - | - | - |

*Exempel 9.25*

Följande programexempel som vi såg inledningsvis använder den ovillkorliga programflödesinstruktionen JMP.

```
Symbolfält   Mnemonic eller direktiv   Operand     Kommentarsfält
             ORG                       $20         ; Program börjar
InPort       EQU                       $FB
UtPort       EQU                       $FC
start:       LDA                       InPort      ; Läs data
             STA                       UtPort      ; Skriv data
             JMP                       start       ; Repetera...
```

Vi hade också kunnat använda BRA-instruktionen:

```
start:       LDA                       InPort      ; Läs data
             STA                       UtPort      ; Skriv data
             BRA                       start       ; Repetera...
```

*Exempel 9.26*

Om programflödesändringen i Exempel 9.25 görs med JMP-instruktionen blir maskinprogrammet:

| Adress | Maskin-kod | | Assemblerkod | |
|---|---|---|---|---|
| 20 | F1 | start | LDA | InPort |
| 21 | FB | | | |
| 22 | E1 | | STA | UtPort |
| 23 | FC | | | |
| 24 | **33** | | **JMP** | **start** |
| 25 | **20** | | | |

Om vi i stället väljer BRA-instruktionen blir maskinprogrammet:

| Adress | Maskin-kod | | Assemblerkod | |
|---|---|---|---|---|
| 20 | F1 | start | LDA | InPort |
| 21 | FB | | | |
| 22 | E1 | | STA | UtPort |
| 23 | FC | | | |
| 24 | **21** | | **BRA** | **start** |
| 25 | **FA** | | | |

***Positionsoberoende kod***

Genom att i maskininstruktionen specificera programflödesändringen med *avståndet* till destinationsadressen görs instruktionen *positionsoberoende*, dvs. programflödesändringen sker till rätt adress inom programmet oberoende av var det placerats i minnet. Denna egenskap är värdefull om man vill *relokera* programmet, dvs. flytta maskinkod mellan olika ställen i minnet utan att behöva assemblera programmet på nytt.

*Exempel 9.27 Positionsoberoende kod*

Den PC-relativa offseten för BRA-instruktionen hos FLISP bestäms enligt:

```
OFFSET = OPERAND ADRESS -((OP-kod adress)+2)
```

Om vi betecknar:

lägesangivelsen för `OPERAND ADDRESS` **?+*M***
`OP-kod adress` betecknas ?+*N*,
och offseten betecknas *V*

där '?' är en godtycklig absolut basadress i minnet, *M* och *N* är bara offseter till denna basadress,
då kan vi skriva:

$V = ? + M - (?+N+2) = M-(N+2)$

dvs. absolut adressinformation krävs inte för att koda instruktionen, som därför är positionsoberoende.





Vid villkorliga programflödesinstruktioner utförs programflödesändringen endast om ett villkorstest befunnits vara sant, då laddas programräknaren med en destinationsadress. Om testet däremot ger ett falskt resultat så fortsätts den raka exekveringsföljden. Man kan alltså säga att exekveringsföljden via en villkorlig instruktion har en *förgrening* (*eng. branch*). Därför benämns instruktioner för villkorliga programflödesändringar *branch*-instruktioner.

Ett villkorstest utgörs alltid av en utsaga om flaggorna i CC-registret. Utsagan är formad med någon eller några av flaggorna i flaggregistret, baserat på relationerna i Tabell 9.4. Utsagan används för att testa ett resultat som just erhållits i instruktionsexekveringen. Utsagorna förekommer i par, om det finns en "branch"-instruktion för en utsaga så finns det en annan "branch"-instruktion för en motsvarande negerad utsaga.

| 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 |

TABELL 9.4 VILLKORSTEST

*Exempel 9.28*

Korrekt branch-instruktion väljs baserat på relationsoperator och datatyp. Antag följande deklarationer:

```
signed     char  sindex;
unsigned   char  uindex;
...
      if( sindex < 10 )
            { satser }
kodas:
      LDA   sindex
      CMPA  #10
      BLO   satser
medan
...
      if( uindex < 10 )
            { satser }
ska kodas:
      LDA   sindex
      CMPA  #10
      BLT   satser
```

### 9.4.3 Indirekt indexerad adressering

Vid absolutadressering adresseras en operand genom att adressen explicit ingår i instruktionen. Absolutadressering kan därför bara användas när data finns på fasta adresser, kända vid assembleringen, i systemet. Vi skall nu se på alternativ till absolutadressering, adresseringssätt som konstruerats för att tillmötesgå krav som:

- att programmet skall kunna arbeta med operandvärden oberoende av deras placering i primärminnet, dvs en instruktions operand skall kunna ha olika adress från körning till körning
- att programmet skall kunna arbeta med och inom datastrukturer av varierande slag, som tabeller, stackar etc
- att dataarean tillhörande ett positionsoberoende program också skall vara positionsoberoende

Ett program kan använda sig av operander med varierande lägen i minnet genom instruktioner med indirekt adressering. I dessa fall innehåller instruktionen ej operandens effektivadress utan anger istället platsen, i vilket effektivadressen finns placerad. När effektivadressen finns lagrad i primärminnet på en adress som explicit anges i instruktionen, säger man helt generellt att adresseringssättet är *indirekt*. I instruktionen ges en minnesadress och på den finns adressen till operanden.

Den vanligaste formen för indirekt adressering är att låta ett processorregister hålla effektivadressen och tjäna som *pekare* till operanden. Processorn har då maskininstruktioner som utgår ifrån att effektivadressen finns i detta register. Pekarregistret anges explicit i instruktionen. Detta adresseringssätt kallas allmänt för *register-indirekt*. För att ytterligare utöka användbarheten läggs ofta en så kallad offset, som adderas till pekarregistrets innehåll. Ett sådant adresseringssätt kallas *indexerat*.

Indexering har sitt ursprung i matematikens sätt att numrera element tillhörande en mängd. Här används indexering på liknande sätt för att numrera operander i en datastruktur lagrad på en följd av konsekutiva adresser i adressrummet. En sådan datastruktur kännetecknas av tre parametrar:

- *basadress*, dvs. datastrukturens startadress i minnet
- *operandavstånd* (*eng. offset*), som är operandens adressavstånd från basadressen
- *storlek*, som är datastrukturens totala storlek

Adressen till en operand i strukturen skapas genom att addera dess avstånd till basadressen. Man ser tydligt en likhet mellan avstånd och index.

*Exempel 9.29 Lagring av vektor*

Datastrukturer som *vektorer* ger en direkt tillämpning för indexerad adressering. Enkla datatyper, som exempelvis **char**, kan grupperas i vektorer av godtycklig längd. En vektor kan deklareras enligt:

```
char namn[längd];
```

som anger att symbolen *namn* består av *längd* antal element av typen **char**. En vektor lagras med elementen i påföljande minnesadresser. Vektorns symboliska namn ger adressen till det första elementet. Följande deklaration:

```
char  char_array[4];
```

anger att symbolen `char_array` upptar 4 bytes i minnet. Vektorns innehåll är odefinierat. En vektor kan också initieras, dvs. ges initialvärden, följande deklaration:

```
char  init_array[4] = {1,2,3,4};
```

anger att symbolen init_array innehåller de på varandra följande värdena 1,2,3 och 4. Vektorerna lagras på följande sätt i minnet:





| Adress | Innehåll | Assemblerkod (disassemblering) | | |
|---|---|---|---|---|
| | | | ORG | $0 |
| 20 | | char_array: | RMB | 4 |
| 21 | | | | |
| 22 | | | | |
| 23 | | | | |
| 24 | 1 | init_array | FCB | 1 |
| 25 | 2 | | FCB | 2 |
| 26 | 3 | | FCB | 3 |
| 27 | 4 | | FCB | 4 |
| 28 | | | | |

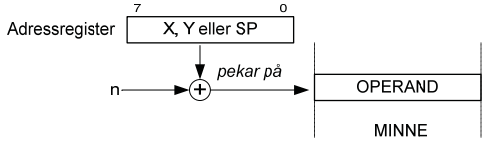
FLIS-processorn tillhandahåller två varianter av indexerad adressering. De skiljer sig åt i sättet att specificera basadress och index:

- basadress i ett register och index i instruktionen (konstant index)
- basadress i ett register och index i ett annat register (variabelt index)

***Indexerad adressering med konstant offset***

Indexerad adressering är väl lämpad vid användning av vektorer. Ett indexregister (X eller Y) kan då tilldelas vektorns symboliska basadress, dvs. adressen till det första elementet. Adressen till operanden, dvs. det element i vektorn som avses, bildas sedan genom att en offset adderas till innehållet i registret.

Adresseringssättet finns för registren X, Y och SP:




Detta sätt att bilda adressen gör att man namnger vektorns index från 0 och uppåt. Då motsvarar indexet exakt den offset som används vid adressbildningen Indexet för det sista elementet i en vektor med längden *n* element blir därför *n*-1.

*Exempel 9.30 Indexerad adressering med konstant offset*

Antag variabeldeklarationerna:

```
char char_array[4];
char item;
```

Vi vill göra tilldelningen:

```
item = char_array[2];
```

dvs. kopiera element med ordningsnummer 3 i vektorn till variabeln `item`.

```
                ORG    0
char_array      RMB    4
item            RMB    1
                ORG    $20
```

```
        LDX   #char_array
        LDA   2,X
        STA   item
```

Den omvända tilldelningen är lika enkel, om vi vill göra tilldelningen:

```
char_array[2] = item;
```

dvs. kopiera variabeln `item` till element 3 i vektorn, blir detta.

```
        ORG   $20
        LDX   #char_array
        LDA   item
        STA   2,X
```

*Exempel 9.31*

Ett medlemsregister där varje medlem registreras med en identifierare (`id`), förnamn (`firstname`), efternamn (`surname`) och ålder (`age`) kan, i programspråket "C", se ut på följande sätt. Av praktiska skäl begränsas namnsträngarnas längd:

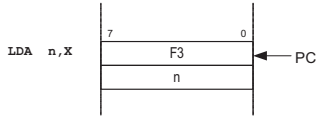
```
struct member{
      unsigned char      id;
      char               firstname[5];
      char               surname[9];
      unsigned char      age;
};
```

Datastrukturens totala storlek fås genom att summera de ingående komponenternas storlekar, dvs 1+5+9+1=16 bytes. De ingående komponenterna ligger konsekutivt i minnet, operandavstånden fås därför genom att succesivt addera storleken av de föregående komponenterna:

- `id` får operandavstånd 0.
- `firstname` får operandavstånd 1 byte
- `surname` får operandavstånd 1+5= 6 bytes
- `age` får operandavstånd 1+5+9=15 bytes

*Exempel 9.32*

Antag att register X innehåller basadressen till en variabel av typen `member`, beskriven i Exempel 9.31. Vi kan då enkelt referera komponenterna `id` och `age`:




Operationen M(X+id)→A kodas:

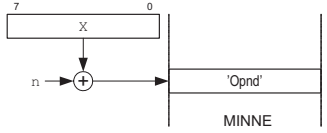
```
LDA    0,X
```

ty komponent `id` har offset 0.

Operationen M(X+age)→A kodas:

```
LDA    15,X
```

ty komponent `age` har offset 15.




Om man i stället söker en textsträng i datastrukturen (`firstname` eller `surname`) då är det ju adressen som ska anges. I detta fall används instruktionen LEA och namnsträngen `firstname` refereras med användning av register Y. Operationen X+firstname→Y kodas då exempelvis:

```
TFR    X,Y
LEAY   1,Y
```



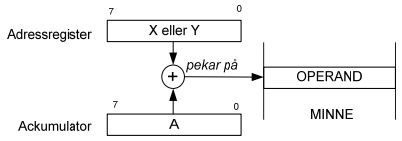


### 9.4.4 Register indirekt med ackumulatoroffset

Om ett vektorindex inte är känt vid assembleringen kan inte heller indexerad adressering med konstant offset användas. I stället utnyttjar vi då indexerad adressering med offset i register. Registerrelativ adressering (A,R) innebär att innehållen i registren R och A adderas för att bilda adressen till operanden. R kan här vara något av registren X eller Y.

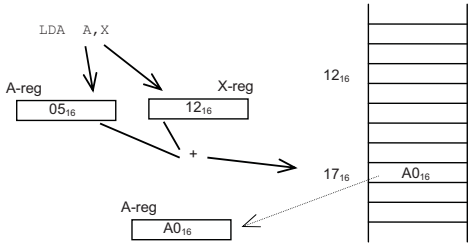
***Indexerad adressering med registeroffset***

Operandens adress bildas genom att innehållet i register A adderas till ett adressregister (X eller Y), resultatet blir den effektiva adressen till operanden. Vi har då alltså ett lämpligt adresseringssätt för uttryck där ett vektorindex utgörs av en variabel:




*Exempel 9.33 Adressberäkning vid ackumulatorindex*

Ackumulatorindexerad adressering sker ex vis med FLISP-instruktionen `LDA A,X`. Den läser data på adressen X + A och placerar det i ackumulator A.




*Exempel 9.34 Tabelluppslagning*

I detta exempel visas en enkel "tabelluppslagning". Vi vill översätta det binära värdet 0..9 till motsvarande ASCII-tecken. En tabell med ASCII-tecknen för siffrorna 0 t.o.m 9 skapas av följande deklaration:

```
char  ascii_table[]={'0','1','2','3','4','5','6','7','8','9'};
```

En sådan initierad vektor motsvaras i assemblerspråk av:

```
ascii_table: FCB    '0','1','2','3','4','5','6','7','8','9'
```

Observera hur citationstecken används för att ange att vi menar ASCII-kod. Vi hade också kunnat göra följande deklaration i assemblerspråk:

```
ascii_table: FCB    $30,$31,$32,$33,$34,$35,$36,$37,$38,$39
```

Låt oss nu anta register A innehåller ett binärt värde, 0 t.o.m 9. Vi vill ersätta värdet i A med det motsvarande ASCII-värdet. Det kan göras med följande sekvens:

```
        LDX   #ascii_table ; startadressen till tabellen laddas till X
        LDA   A,X          ; värdet i A adderas till X, bildar ny adress till ett tecken i tabellen
                           ; detta tecken laddas till A
```

*Exempel 9.35 Referens i vektor med variabelt index*

Antag variabeldeklarationerna:

```
char char_array[4];
char item;
char index;
```

Vi vill göra tilldelningen:

```
item = char_array[index];
```

dvs. kopiera elementet som anges av variabeln `index` till `item`.

```
                ORG   0
char_array      RMB   8
item            RMB   1
index           RMB   1
                ORG   $20
                LDX   #char_array       ; basadressen till vektorn
                LDA   index             ; variabelns värde
                LDA   A,X               ; det indexerade värdet
                STA   item
```

Den omvända tilldelningen kräver att vi använder ytterligare ett register (Y), om vi vill göra tilldelningen:

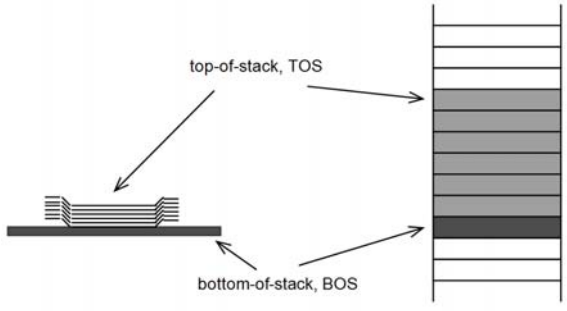
```
char_array[index] = item;
```

dvs. kopiera variabeln `item` till elementet som anges av variabeln `index`, blir detta.

```
                ORG   $20
                LDX   #char_array
                LDA   index
                LDY   item
                STY   A,X
```

### 9.4.5 Stackbyggnad och användning

På en stack lagras binära ord på precis samma sätt som man lägger hö på en höstack eller staplar tallrikar på en hylla, dvs bildligt talat ovanpå varandra, se Figur 9.5. Basläget kallas för *bottom-of-stack, BOS* och läget av det sist pålagda elementet kallas *top-of-stack, TOS*.



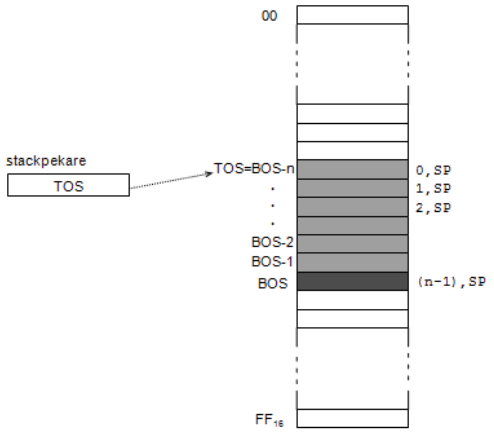

FIGUR 9.5 STACKSTRUKTUR

Nya element tillförs ovanifrån, dvs läggs på toppen. Principen för de binära ordens, höets och tallrikarnas åtkomst är *sist in - först ut* (*eng. last in - first out, LIFO*). Processen att tillföra element benämns *push* och processen att ta element ifrån stacken benämns *pull* (ibland *pop*).





Stacken är alltså en *LIFO-struktur*, som företrädesvis används för att på ett enkelt sätt temporärt lagra data. FLISP har försetts med ett speciellt pekarregister, SP, *stackpekare* som får hålla reda på TOS-läget enligt Figur 9.6. Stacken initieras när stackpekaren laddades med BOS-adressen, detta sker vanligtvis alldeles i inledningen av ett program. Praktiskt taget alla FLISP-instruktioner har försetts med adresseringssätt (stack-relativt) som gör det enkelt att snabbt lagra och återställa data direkt via stackpekaren.




FIGUR 9.6 STACKENS TILLVÄXT OCH ADRESSERING VIA STACKPEKAREN.

För snabb lagring/återställning av register till/från stacken finns det speciella "push"- och "pull"-instruktioner:

| **PSH** | **Push register on Stack** |
|---|---|
| RTN | SP–1 → SP<br>R → M(SP) |
| Flaggor | Påverkas ej |
| Beskrivning | Stackpekaren uppdateras först. Angivet registerinnehåll skrivs sedan på stacken |

Detaljer:

| Instruktion **PSH** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| PSHA | Inherent | 10 | 1 | 3 | SP–1 → SP, A → M(SP) | - | - | - | - |
| PSHX | Inherent | 11 | 1 | 3 | SP–1 → SP, X → M(SP) | | | | |
| PSHY | Inherent | 12 | 1 | 3 | SP–1 → SP, Y → M(SP) | | | | |
| PSHCC | Inherent | 13 | 1 | 3 | SP–1 → SP, CC → M(SP) | | | | |

## PUL Pull register from stack

| | |
|---|---|
| Varianter | |
| RTN | M(SP) → R<br>SP+1 → SP |
| Flaggor | Flaggorna påverkas endast vid PULC, då flaggorna får värden från stacken |
| Beskrivning | Översta dataordet på stacken läses och placeras i angivet register. Stackpekaren uppdateras sedan |

Detaljer:

| Instruktion **PUL** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| PULA | Inherent | 14 | 1 | 3 | M(SP) → PC, SP+1 → SP | - | - | - | - |
| PULX | Inherent | 15 | 1 | 3 | M(SP) → X, SP+1 → SP | - | - | - | - |
| PULY | Inherent | 16 | 1 | 3 | M(SP) → Y, SP+1 → SP | - | - | - | - |
| PULCC | Inherent | 17 | 1 | 3 | M(SP) → CC, SP+1 → SP | Δ | Δ | Δ | Δ |

### *Exempel 9.36 Registerspill*

Registerspill kan uppträda vid evaluering av mer komplexa uttryck. Ta följande exempel:

```
char  sum,index;
char  array[5];

sum = array[index] + array[index+1];
```

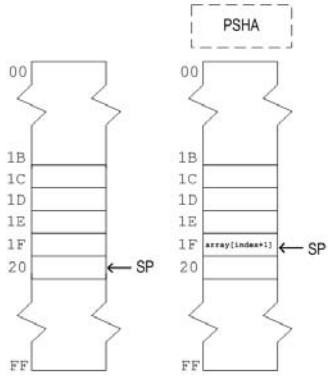
Högerledet evalueras:

```
LDA   index
INCA
LDX   #array
LDA   A,X
```

Vi har nu laddat första operanden i register A, men vi kommer strax att tvingas använda detta register igen för att kunna evaluera vänstra ledet, denna situation ger exempel på så kallat *registerspill*. Vi löser problemet genom att spara det temporära värdet på stacken:

```
PSHA
```

dvs. *spiller* register A till stacken. Se figuren till höger som visar effekten på stacken av operationen. Vi har nu vårt evaluerade högerled överst på stacken varför vi kan övergå till att bestämma vänsterledet:




```
LDA   index
LDX   #array
LDA   A,X
```

Vi adderar nu värdet av vänsterledet med värdet från stacktoppen:

```
ADDA  0,SP
```

och utför tilldelningen:

```
STA   sum
```

Slutligen återställer vi stackpekaren, till värdet den hade före push-operationen, det kan vi göra på olika sätt:

```
PULA
```

dvs. återställ register A från stacktoppen. Men eftersom vi inte längre har någon användning av värdet i fråga kan vi nöja oss med att helt enkelt modifiera stackpekaren i stället:

```
LEASP 1,SP
```

En av dessa instruktioner låter vi alltså avsluta kodsekvensen.





### 9.4.6 Subrutiner

Ett icke trivialt program delas in i *huvudprogram* och *subrutiner*. Subrutinerna utformas baserat på programmets funktion. Implementering av subrutiner sker med hjälp av instruktionerna BSR (Branch to SubRoutine), `JSR` (*Jump to SubRoutine*) respektivet `RTS` (*ReTurn from Subroutine*).

## BSR Branch to subroutine

| | |
|---|---|
| RTN | SP–1 → SP<br>PC → M(SP)<br>PC+Offset → PC |
| Flaggor | Påverkas ej |
| Beskrivning | PC-värdet (återhoppsadressen) skrivs först på stacken. Ett hopp utförs sedan till adressen ADRESS = PC+Offset. Offset räknas från adressen efter BSR-instruktionen, dvs vid uträkningen av hopp-adressen pekar PC på operationskoden direkt efter BSR-instruktionen (= återhoppsadressen) |

Detaljer:

| Instruktion **BSR** | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| BSR Adr | Relativ | 20 | 2 | 5 | SP–1 → SP, PC → M(SP), PC+Offset → PC | - | - | - | - |

## JSR Jump to subroutine

| | |
|---|---|
| RTN | SP–1 → SP; PC → M(SP); EA → PC |
| Flaggor | Påverkas ej |
| Beskrivning | PC-värdet, som är adressen till instruktionen efter JSR-instruktionen, dvs återhoppsadressen, skrivs först på stacken. Ett hopp utförs sedan till adressen EA. |

Detaljer:

| Instruktion **JSR** Variant | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| JSR Adr | Absolute | 34 | 2 | 4 | SP–1 → SP, PC → M(SP), Adr → PC | - | - | - | - |
| JSR n,X | Indexed | 54 | 2 | 5 | SP–1 → SP, PC → M(SP), n+X → PC | | | | |
| JSR A,X | Indexed | 64 | 1 | 5 | SP–1 → SP, PC → M(SP), A+X → PC | | | | |
| JSR n,Y | Indexed | 74 | 2 | 5 | SP–1 → SP, PC → M(SP), n+X → PC | | | | |
| JSR A,Y | Indexed | 84 | 1 | 5 | SP–1 → SP, PC → M(SP), A+X → PC | | | | |

## RTS Return from Subroutine

| | |
|---|---|
| RTN | M(SP) → PC; SP+1 → SP |
| Flaggor | Påverkas ej |
| Beskrivning | Återhopp från en subrutin utförs genom att översta dataordet på stacken, dvs återhoppsadressen, läses och placeras i PC. Stackpekaren uppdateras sedan |

Detaljer:

| Instruktion **RTS** | Adressering metod | OP | # | ~ | Operation | Flaggor N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|
| RTS | Inherent | 43 | 1 | 2 | M(SP) → PC, SP+1 → SP | - | - | - | - |

*Exempel 9.37 Subrutin 'mul' för multiplikation*

FLISP har ingen maskininstruktion för att multiplicera två tal så denna operation måste implementeras som en subrutin. Här visar vi hur enklast tänkbara algoritm kan användas, observera att subrutinen förutsätter att båda operanderna är större än 0 och att ingen hänsyn tas till eventuellt spill.

```
; subrutin mul
; multiplicerar två 8-bitars tal (utan tecken)
; returnerar 8-bitars resultat
; Indata
;     register A: operand 1
;     register Y: operand 2
; Utdata
;     register A: produkten (operand 1 * operand 2)
mul:
      PSHA                ; spara operand 1 som multiplikand
mul_loop:
      CMPY    #1          ; färdig då multiplikator=1
      BEQ     mul_exit
      ADDA    0,SP        ; addera multiplikanden
      LEAY    -1,Y        ; minska räknare (multiplikator)
      BRA     mul_loop
mul_exit:
      LEASP   1,SP        ; återställ stack
      RTS
```

Nu kan följande operation:

```
unsigned char      prod,op1, op2;
prod = op1 * op2;
```

kodas som:

```
      LDA   op1
      LDY   op2
      JSR   mul
      STA   prod
```

***Anropskonventioner***

Ofta, men inte alltid, kommuniceras data till/från subrutinen, detta sker i form av *parametrar* till och *returvärde* från subrutinen. I det inledande exemplet (Exempel 9.37) har subrutinen två parametrar, *operand 1* respektive *operand 2*, och ett returvärde, *produkten*. Parametrar kan skickas till en subrutin på olika sätt, i exemplet använder vi processorns register (A och Y) men om parametrarna är många kan man i stället övergå till att använda stacken för detta ändamål. Oavsett vilken metod man väljer måste det alltid vara väldigt tydligt hur data kommuniceras, vi kallar detta *anropskonvention*.

En subrutin kan ha godtyckligt antal parametrar men det är vanligt att man inskränker sig till *ett* returvärde vilket vi också avgränsar oss till i denna text.

Precis som för deklarationer och uttrycksevaluering kan vi använda en beskrivande notation för att specficera subrutiner, grundformen för denna (inga parametrar, inget returvärde) visas i Figur 9.7.

| Beskrivande notation: | Implementering i assemblerspråk: |
|---|---|
| `sub_namn`<br>`{`<br>`  .... programkod`<br>`}` | `sub_namn:`<br>`  ....`<br>`  .... programkod`<br>`  ....`<br>`  RTS` |
| beskrivande subrutinanrop: | |
| `sub_namn();` | `  JSR  sub_namn` |

FIGUR 9.7 IMPLEMENTERING AV SUBRUTINER





*Exempel 9.38 Notation för subrutin*

Vi kan ge en beskrivande notation av subrutinen `mul` ovan genom att låta symbolnamnet föregås av datatypen för returvärdet (`unsigned char`) och ange parametrarna med respektive datatyp, i tur och ordning, inom parentes:

```
unsigned char mul(unsigned char operand1, unsigned char operand2)
```

Nu kan följande operation:

```
unsigned char      prod,op1, op2;
prod = op1 * op2;
```

kodas som:

```
prod = mul( op1, op2);
```

*Exempel 9.39 Organisation av ett komplett program för FLISP*

Med konventionerna för primärminnets användning (se Figur 9.2) kan vi nu skapa ett bättre ”skelett” för applikationsprogram för FLISP.

```
Code        EQU         $20
Data        EQU         0

            ORG         Data
; deklarationer av globala variabler följer här...
;     ...
            ORG         Code
; programkod följer här...
start:      LDSP        #Code       ; initiera stackpekare
            JSR         main
exit:       JMP         exit

main:       ...
            ...
            RTS
; övriga subrutiner...

            ORG         $FF
            FCB         start       ; RESET-vektor
```

## 9.4.7 Lokala variabler

Lokala variabler karaktäriseras av att de bara kan refereras (är synliga) i den begränsade omgivning i vilken de är deklarerade. Mer precist innebär detta att dessa variabler bara används under en begränsad tid, i en speciell subrutin. Det sistnämnda får konsekvenser för hur symbolnamnen kan väljas. Exempelvis måste alla globala variabler ha unika namn medan lokala variablers namn endast måste skiljas åt inom den subrutin i vilken de är deklarerade. Detta kan vara en fördel då man ofta vill välja talande symbolnamn för variabler, men kan i bland verka förvirrande då samma namn uppträder i olika sammanhang och därför har olika innebörd. Vi förtydligar nu detta med några exempel.

### Exempel 9.41 Datatyp och lagringsklass

Exemplet illustrerar skillnaden mellan implementering av den globala variabeln `glob_a` och den lokala variabeln `loc_a`. Följande beskrivning:

```
char glob_a;
sub_namn
{
   char loc_a;
}
```

ger upphov till följande assemblerprogram:

```
glob_a      RMB   1     ;  1 byte reserveras för symbolen.
sub_namn:
; 1 byte reserveras på stacken för symbolen loc_a
      LEASP -1,SP
      ...
; I subrutinen har nu den lokala variabeln adressen 0,SP
      ...
; stacken återställs, variabeln loc_a finns inte längre
      LEASP 1,SP
      RTS
```

Variabler hanteras lämpligen i den ordning de påträffas i källtexten. För globala variabler har detta ingen praktisk betydelse men är desto viktigare då det gäller lokala variabler.

### Exempel 9.42 Adressering av lokala variabler

Detta exempel illustreras hur plats för flera lokala variabler åstadkoms på stacken och hur deras adresser bildas då vi behandlar variablerna i den ordning de påträffas.

```
sub_namn
{
   char a, b, c, d;
}
```

ger upphov till följande assemblerprogram:

```
sub_namn:
; 4 bytes reserveras på stacken för symbolerna a,b,c och d
      LEASP      -4,SP
```

Här kan nu de lokala variablerna användas. Eftersom vi har behandlat dom i ordning kan vi enkelt tilldela respektive variabel adresserna i form av stack-relativt adresseringssätt.

- Variabel d behandlades sist och får adress `0,SP`
- variabel c behandlades näst sist och får adressen `1,SP`
- variabel b får adressen `2, SP`
- variabel a får adressen `3,SP`.

```
; stacken måste återställas inför återgång
      LEASP      4,SP
      RTS
```





### Exempel 9.43 Tilldelningar av lokala variabler

Följande exempel illustrerar användning av lokala variabler:

```
sub_namn
{
   char a, b, c, d;
   a = 10;
   b = c + a;
   ...
}
```

Bestäm adresser för variablerna:

```
d = 0,SP
c = 1,SP
b = 2,SP
a = 3,SP.
```

Koda därefter tilldelningssatserna på vanligt sätt:

```
sub_namn:
      LEASP      -4,SP
      LDA        #10              ; a = 10;
      STA        0,SP
      LDA        1,SP
      ADDA       0,SP             ; c + a
      STA        2,SP
      ...
      LEASP      4,SP
      RTS
```

### Exempel 9.44 Kodning av funktionsanrop

Ett anrop av funktionen `mul` kan beskrivas detaljerat enligt:

```
unsigned char     prod,op1, op2;      deklarationer
prod = mul(op1, op2);                 funktionsanrop
```

Själva funktionsanropet kodas på följande sätt, observera att parameterlistan behandlas från höger till vänster:

```
LDA   op2          Lägg första parameter på stacken
PSHA
LDA   op1          Lägg andra parameter på stacken
PSHA
JSR   mul          Anropa subrutinen
LEAS  2,sp         Återställ stackpekaren till värdet före anropssekvensen
STA   prod         Utför tilldelningen
```

*Exempel 9.45 Funktionen 'mul', beskrivning och implementering*

Funktionen `mul` kan beskrivas detaljerat enligt:

```
unsigned char mul(unsigned char operand1, unsigned char operand2)
{
   unsigned char result;

   result = 0;
   while( operand2 > 0 )
   {
         result = result + operand1;
         operand2 = operand2 - 1;
   }
   return result;
}
```

Vi kan systematiskt översätta från denna beskrivning, dvs. implementera i assemblerkod. Vi börjar med att bestämma adresserna till parametrar och lokala variabler. Enklaste sättet att göra detta är att tilldela dessa adresser (relativt stacken) genom att behandla först variabler sedan parametrar, bakifrån:

Bestäm adress för variabel:

```
      result      =  0,SP
```

Då listan av lokala variabler behandlats måste vi ta hänsyn till återhoppsadressen (*returPC*) som vi inte ser i beskrivningen av funktionen men som hamnar här vid själva anropet av funktionen:

```
      returPC     =  1,SP
```

Parameterlistan läser vi nu från *vänster till höger*, detta är viktigt och resultatet av att vi vid anrop, lägger parametrar på stacken i den omvända ordningen:

Adresser för parametrar blir då:

```
      operand1  =  2,SP
      operand2  =  3,SP
```

Vi är nu klara att implementera beskrivningen:

```
; unsigned char mul(unsigned char operand1, unsigned char operand2)
; {
mul:
; unsigned char result;
      LEAS  -1,SP
; result = 0;
      LDA   #0
      STA   0,SP
; while( operand2 > 0 )
mul_while:
      LDA   3,SP
      CMPA  #0
      BLS   mul_return
;   result = result + operand1;
      LDA   0,SP
      ADDA  2,SP
      STA   0,SP
;   operand2 = operand2 - 1;
      LDA   3,SP
      SUBA  #1
      STA   3,SP
; return result;
mul_return:
      LDA   0,SP
      LEASP 1,SP
      RTS
```





### 9.4.8 Autoincrement och autodekrement

De adresseringssätt som används med stackpekaren SP för stackhantering (*push* och *pull*), kan användas med godtyckliga register avsedda att hantera minnesadresser.

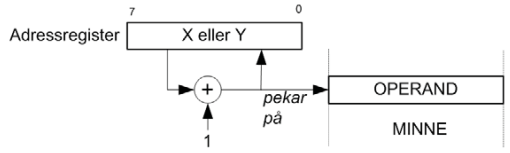
Register-indirekt adressering har en naturlig utvidgning som är lämplig att använda när samhörande operander lagrats på konsekutiva adresser och man i programmet vill flytta sig från operand till operand. Detta är ex vis vanligt när man arbetar med tabelliknande datastrukturer. Med "store"- och "load"-instruktioner som har register-indirekt adressering med autoinkrement och autodekrement (dvs. STA och LDA) kan man generellt implemebtera den datastruktur som går under namnet *stack*.

Den naturliga utvidgningen är att låta pekarregistrets innehåll ökas med ett eller minskas med ett varje gång instruktionen utförs. På detta sätt kan man i program röra sig framåt och bakåt i datastrukturen. Instruktioner med denna typ av automatisk modifiering av pekarregistrets innehåll har adresseringsmoden register-indirekt med *autoinkrement* eller *autodekrement*.

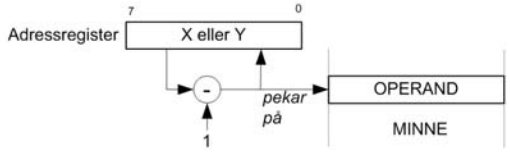
*Postinkrementering* innebär att instruktionen efter det att adresserad operand använts inkrementerar pekarregistret. Det senare förbereder användning av nästa element i strukturen.

*Predekrementering* innebär att instruktionen först dekrementerar pekarregistret och därefter använder den operand det pekar på.

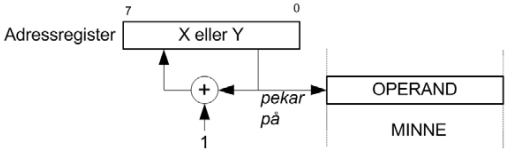
***Pre auto increment*** **[+R]**
RTN: R+1→R; M(R)
Assemblersyntax: ,+R



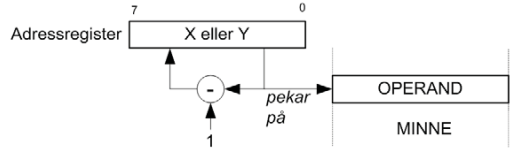

***Pre auto decrement*** **[–R]**
RTN: R-1→R; M(R)
Assemblersyntax: ,-R




***Post auto increment*** **[R+]**
RTN: M(R); R+1→R
Assemblersyntax: ,R+




***Post auto decrement*** **[R-]**
RTN: M(R); R-1→R
Assemblersyntax: ,R-



*Exempel 9.47 Funktion memcpy*

Konstruera en subrutin, `memcpy`, som kopierar ett minnesblock av storleken `size`, med början från adressen `start` till ett nytt minnesblock med början på adressen `dest`. Alla parametrar är en byte stora och anropskonventionen ges av följande:

```
memcpy(  start, dest, size )
```

Vi bestämmer först adresser för parametrarna:

```
returPC     =  0,SP      (används ej)
start       =  1,SP
dest        =  2,SP
size        =  3,SP.
```

Vi upplåter register X till `start` och register Y till `dest` för att kunna använda *autoincrement* adressering, parametern `size` når vi direkt på stacken där den finns redan vid anropet.

```
memcpy:  LDX   1,SP         ; startadress till X
         LDY   2,SP         ; destinationsadress till Y
memcpy_loop:
         TST   3,SP         ; size = 0 ?
         BEQ   memcpy_exit  ; om så, avsluta
         LDA   1,X+         ; kopiera en byte
         STA   1,Y+
         DEC   3,SP         ; size = size-1
         BRA   memcpy_loop  ; nästa
memcpy_exit:
         RTS
```

*Exempel 9.48 Användning av autoincrement och autodecrement*

Två vektorer, `origin` och `mirror`, är lika långa (`size` bytes), skriv en subrutin `inverse` som tar dessa vektorer som parametrar och kopierar `origin` till `mirror` i omvänd ordning. Alla parametrar är en byte stora och anropskonventionen ges av följande:

```
inverse(  origin, mirror, size )
```

Vi bestämmer först adresser för parametrarna:

```
returPC     =  0,SP      (används ej)
origin      =  1,SP
mirror      =  2,SP
size        =  3,SP.
```

Vi upplåter register X till `origin` och register Y till `size` för att kunna använda *autoincrement* adressering, parametern `size` når vi direkt på stacken där den finns redan vid anropet.

```
inverse: LDX   1,SP         ; startadress till X
         LDA   2,SP         ; Bestäm slutadress hos parameter 'mirror'
         ADDA  3,SP
         PSHA
         PULY               ; destinationsadress till Y
inverse_loop:
         TST   3,SP         ; size = 0 ?
         BEQ   inverse_exit ; om så, avsluta
         LDA   1,X+         ; kopiera en byte
         STA   1,Y-
         DEC   3,SP         ; size = size-1
         BRA   inverse_loop ; nästa
inverse_exit:
         RTS
```





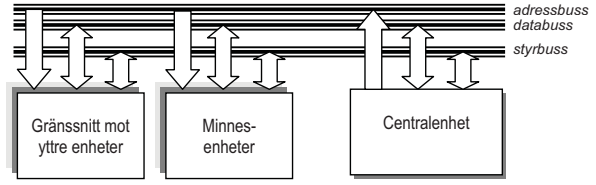
# 10 Datorsystemet

Detta kapitel ägnas åt metoder och principer som används för att bygga upp ett komplett datorsystem bestående av centralenhet, minne och in- ut- /matningsenheter.

- Bussystem, intern kommunikation i datorsystemet
- Adressavkodning, hur primärminne och I/O-enheter kan anslutas
- Olika principer för intern kommunikation (busskommunikation)
- Undantagshantering

## 10.1 Bussystem

Datorn är uppbyggd av enheter som kommunicerar med varandra via elektriska ledare. En grupp av sådana ledare kallas för *buss*. Begreppet *buss* anger alltså ett antal olika elektriska signalförare som funktionellt kan grupperas tillsammans. Flera olika bussar används för att koppla samman *centralenhet*, *minne* och olika typer av *gränssnitt*. Beteckningen *gränssnitt* är här en gemensam beteckning på enheter för in- och/eller ut-/matning från/till elektroniska system som kopplats till datorn. Olika typer av gränssnitt är ofta utförda som integrerade kretsar (eller sammanbyggda med centralenhet och minne) och de kallas också i bland för *periferikretsar*. Informationsutbyte som sker i datorsystemet görs normalt via tre bussar: *databuss*, *adressbuss* och *styrbuss*.

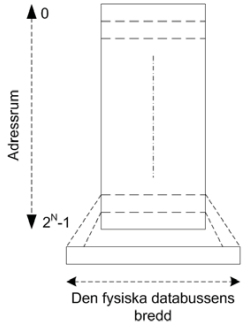



FIGUR 10.1 BUSSYSTEM

Centralenheten initierar kommunikationen genom att välja ett gränssnitt mot någon yttre enhet eller en minnesenhet via adressbussen. Varje enskild enhet har således ett unikt adressintervall inom vilket endast denna enhet aktiveras för att överföra data till eller från centralenheten via databussen. Signaler på styrbussen anger i vilken riktning data överförs. Se Figur 10.1, observera hur vi indikerar att adressbussens innehåll bestäms enbart av centralenheten. Detta är inte alltid fallet men för diskussioner i detta kapitel förutsätter vi att endast centralenheten kan generera adresser på adressbussen. Då det gäller databussen är situationen en annan. Såväl centralenheten som minnesenheter och yttre enheter tillåts placera data på databussen. Detta avgörs dock av signaler på styrbussen vilka i sin tur genereras av centralenheten. Av denna anledning har vi ritat databussen dubbelriktad i figuren. Slutligen innehåller styrbussen signaler som kan ha genererats antingen av centralenheten eller av någon yttre enhet och vi ritar av denna anledning även styrbussen som dubbelriktad.

### 10.1.1 Adressbussen

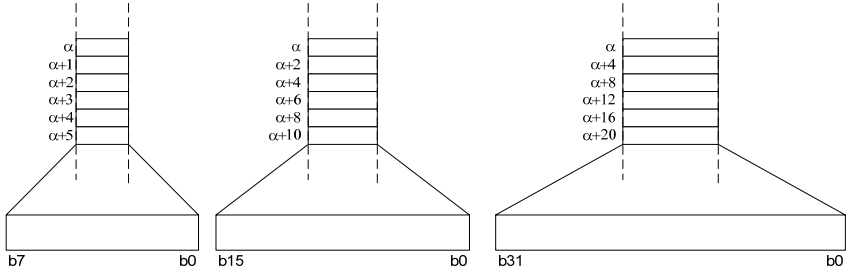
Adressbussens bredd (storlek) bestäms av centralenhetens konstruktion. Antalet adressledningar varierar alltså mellan olika processorer men, som exempel, är 16 bitar vanligt för små processorer, dessa har då oftast *ordbredden* 8 bitar, dvs. 8 bitars databuss. Processorer i mellanklassen har betydligt större adressrum, exempelvis *ARM* (*Advanced Risc Machine*), vanligt förekommande i så kallade "smarta" mobiltelefoner. Här används 32 bitar för såväl adressbuss som databuss. För större processorer, som exempelvis *Intel Core*, är såväl adressbuss som databuss 64 bitar.




FIGUR 10.2 ADRESSRUM

Adressbussens totala omslutning kallas processorns *adressrum*, se Figur 10.2, och kan vara utformad för att adressera *bytes*, dvs. varje enskild adress pekar ut en unik *byte* (8 bitar) i primärminnet. Hos vissa processorer har adressering i stället utformats enbart för jämna ordbredder, dvs. för adressering av 16-bitars data, detta kallas då ofta *word-adressering*. Fördelen med detta är att 16 bitar då kan läsas/skrivas vid varje minnesoperation och prestanda blir alltså högre. För större processorer förbättras prestanda ytterligare genom att hela ord (32 eller 64 bitar) alltid läses från, eller skrivs till minnet.

Konsekvenser av att endast 32 bitar behandlas i en arbetscykel, innebär exempelvis att läsning eller skrivning endast kan ske på adresser som slutar med en med en multipel 4 i ett *byte*-adresserat adressrum, se Figur 10.3. Processorer med dessa egenskaper introducerar då restriktioner på hur data kan lagras i minnet, dvs 16-bitars data måste lagras på en jämn adress, etc. Sådana restriktioner kallas *alignment conditions*, dvs. villkor för hur data kan lagras i minnet. Detta får självfallet konsekvenser då vi vill hantera datatyper som är mindre än adresserbar ordbredd.




FIGUR 10.3 OLIKA SÄTT ATT ORGANISERA MINNET, SOM 8,16 ELLER 32 BITARS ENTITETER

### 10.1.2 Databussen

Medan syftet med adressbussen är att ”peka ut” den del av adressrummet som centralenheten ska utväxla information med är databussens uppgift att överföra informationen mellan dessa delar. Databussens bredd (ordbredden) bestämmer därför hur mycket information som kan överföras under en arbetscykel. Ju mer information som kan överföras desto högre blir förstås prestanda, givet att arbetscykelns tid inte ändras.

Eftersom databussen används för att överföra information såväl till som från centralenheten är bussen också dubbelriktad. Bussens innehåll bestäms därför av att antingen centralenheten, en yttre enhet eller någon minnesenhet *driver* bussen. *Bussarbitreringen*, dvs. avgörandet av vilken enhet som för tillfället får driva databussen, bestäms av centralenheten som signalerar detta via speciella kontrollsignaler som vi samlat i *styrbussen*.



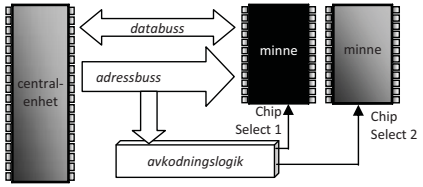


### 10.1.3 Styrbussen

Styrbussen omfattar, precis som namnet antyder, ett antal kontrollsignaler avsedda att dirigera informationsöverföringen. I denna grupp ingår samtliga styrsignaler som krävs för kommunikationen mellan enheterna anslutna till bussystemet. Ett exempel på en styrsignal från centralenheten är skriv- och lässignaler (*MR och MW*) där centralenheten anger att den ska läsa (hämta data från) minne eller gränssnitt respektive skriva till (överföra data till) minne eller gränssnitt. Uppenbarligen kan inte dessa signaler vara aktiva samtidigt. De kan dock vara passiva samtidigt och indikerar då att centralenheten är upptagen med internt arbete som inte kräver kommunikation med någon yttre enhet eller minnet.

## 10.2 Adressavkodning

Med ”adressavkodning” menar man den den teknik som används för att avkoda adressbussen och bilda den speciella signal, ofta kallad ”chip-select” (CS) som används för att aktivera en speciell enhet så som minne eller gränssnitt. Vi har sett att centralenheten kommunicerar med såväl minne som gränssnitt via systemets bussar. Låt oss nu titta närmre på hur detta går till, hur man kan vara säker på att rätt minneskrets eller rätt gränssnitt adresseras av centralenheten. Oftast består datorn av flera minnestyper organiserade i block, och vanligtvis ingår också flera typer av gränssnitt. Vi ska nu inledningsvis, som ett enkelt exempel, redogöra för hur ett komplett datorsystem, med centralenhet, två minneskretsar och två gränssnitt kan byggas upp. Vi gör detta genom att konstruera ett block bestående av grindar. Detta logikblock kallas *adressavkodningslogik* (Figur 10.4).

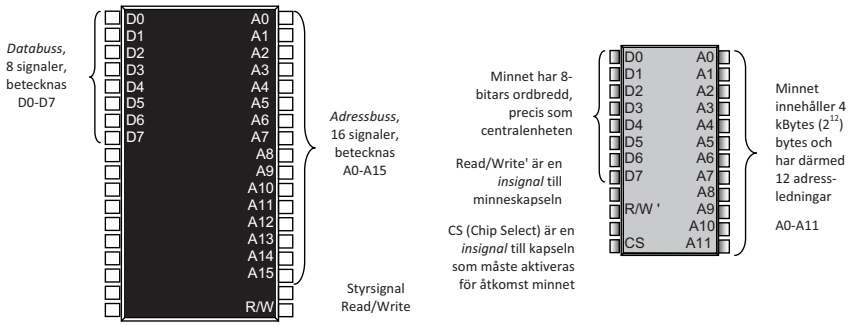



FIGUR 10.4 ADRESSAVKODNINGSLOGIK

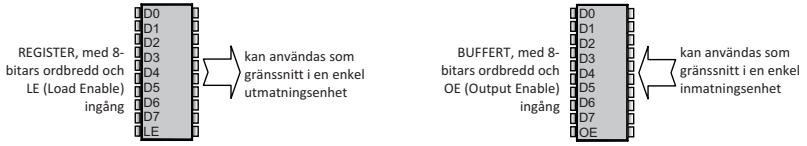
För exemplet antar vi att vi har tillgång till följande komponenter.

- Centralenhet, 64kByte adressrum (16 bitars adressbuss), 8-bitars ordbredd (databuss),
- Minneskapsel RWM 4 kbyte,
- Minneskapsel ROM 4 kbyte
- Register, 8 bitar som inport, Register, 8 bitar som utport.
- Standardkretsar med diverse grindar.

Med de givna förutsättningarna kan vi ge förenklade bilder av de ingående komponenterna. I dessa bilder utelämnar vi signaler och komponenter som inte behövs för adressavkodningen.




FIGUR 10.5 FÖRENKLADE FIGURER AV CENTRALENHET OCH MINNE




FIGUR 10.6 FÖRENKLADE FIGURER AV REGISTER OCH BUFFERT

Vi börjar med att bestämma var, i centralenhetens adressrum, vi vill placera minnen och register. För exemplet väljer vi följande minnesdisposition (adresserna anges i hexadecimal form):

| Kapsel | Startadress | Slutadress |
|---|---|---|
| Minneskapsel RWM 4 kbyte | 0000 | 0FFF |
| Minneskapsel ROM 4 kbyte | C000 | CFFF |
| Buffert, 8 bitar som inport | A000 | A000 |
| Register, 8 bitar som utport | 8000 | 8000 |

TABELL 10.1 MINNESDISPOSITION

Av minnesdispositionen framgår att:

- RWM-minnet ska aktiveras om centralenheten genererar någon av adresserna 0-0FFF.
- ROM-minnet ska aktiveras om centralenheten genererar någon av adresserna C000-CFFF.
- Inporten ska aktiveras om centralenheten genererar adress A000.
- Utporten ska aktiveras om centralenheten genererar adress 8000.

Följande tabell visar de värden adressbussens signaler får anta för respektive kapsel.

| Kapsel | | Adressbuss | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| RWM | $0FFF | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ROM | $CFFF | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $C000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| inport | $A000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | $A000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| utport | $8000 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | $8000 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Vi måste nu, konstruera *logik* som kan jämföra adressbussens värde med den del av adressrummet vi tilldelat respektive kapsel och skapa en signal CS (*chip select*) för varje kapsel. För att göra detta använder vi grindar av typen NOT och AND. Av tabellen framgår att vi kan göra en unik CS-signal för





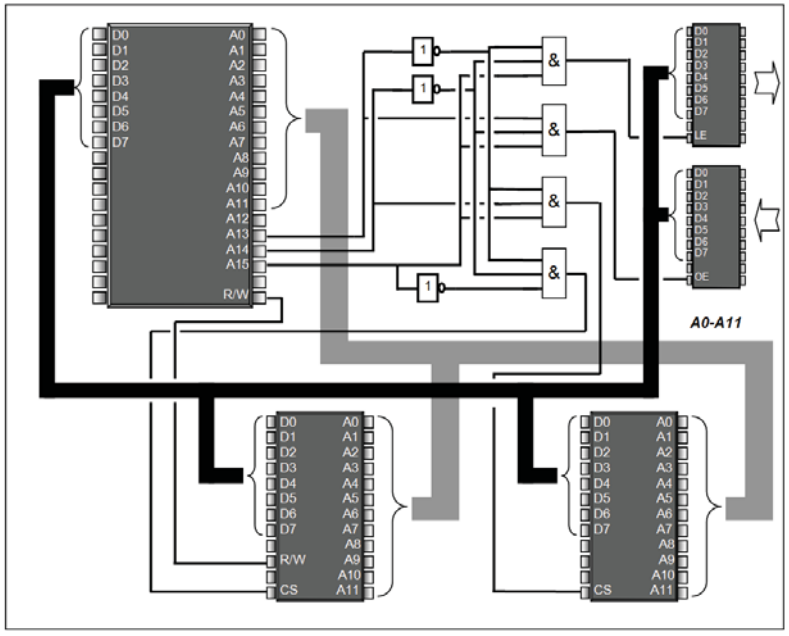
varje kapsel genom att bara använda adressledningarna A15, A14 och A13. Vi ser detta tydligare i följande uppställning:

| Kapsel | | Adressbuss | | |
|---|---|---|---|---|
| | | A15 | A14 | A13 |
| RWM | $0000-$0FFF | 0 | 0 | 0 |
| ROM | $C000-$CFFF | 1 | 1 | 0 |
| inport | $A000 | 1 | 0 | 1 |
| utport | $8000 | 1 | 0 | 0 |

För exempelvis RWM-kapseln ska dessa tre adressignaler vara 0, för ROM-kapseln ska vi ha A15=1, A14=1 och A13=0, osv. Genom att bara låta dessa signaler ingå i adressavkodningen får vi minsta möjliga kretslogik. Eftersom vi lämnar adressledningar från A12 och nedåt som ”don’t care” kommer CS-signalen att vara aktiv för ett större adressintervall än det som från början avsågs för respektive kapsel. Man säger att samma modul avbildas på flera olika adressintervall och kallar detta *ofullständig adressavkodning.* Följande tabell visar nu vårt slutliga val av adressavkodning

| Kapsel | Adressbuss | | | |
|---|---|---|---|---|
| | A15 | A14 | A13 | Adressintervall |
| RWM | 0 | 0 | 0 | 0000-1FFF |
| ROM | 1 | 1 | 0 | C000-DFFF |
| inport | 1 | 0 | 1 | A000-BFFF |
| utport | 1 | 0 | 0 | 8000-9FFF |

Figur 10.7 nedan visar kopplingarna för adressavkodninglogiken. Dataledningarnas anslutningar (D0 till D0, D1 till D1 osv.) är markerade med en svart buss. På motsvarande sätt har adressledningar A0-A11 markerats av en grå buss. Utöver dessa adressledningar, databussen och CS-signalerna ansluts även R/W mellan centralenheten och RWM.



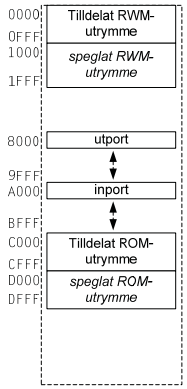

FIGUR 10.7 KOPPLINGAR FÖR ADRESSAVKODNINGSLOGIKEN

### 10.2.1 Ofullständig adressavkodning

Ofullständig adressavkodning innebär att vi utnyttjar det minsta antal adressbitar som krävs, i vårt aktuella system, för att avkoda adressbussen och skapa en unik CS-signal. Innebörden är att det krävs minimalt med logik (grindar) för att implementera systemet. Samtidigt kan det bli svårt, ibland omöjligt, att utvidga systemet. En ofullständig adressavkodning innebär en moduls adress (adresser) replikeras, i adressrymden.

Se figuren till höger, den beskriver resultatet av adressavkodningen i Figur 10.7. Vi ser att ROM-minneskapseln som har tilldelats adressintervallet C000-CFFF dessutom replikeras i adressintervallet D000-DFFF. På liknande sätt kommer RWM-kapseln att aktiveras för adresser i intervallen 0-FFF och 1000-1FFF. Detta beror på att vi utelämnat A13 i avkodningslogiken.

Då det gäller inporten, vars tilldelade adress är A000, kommer den att aktiveras av avkodningslogiken för *varje* adress i intervallet A000-BFFF, dvs. A000, A001, A002, A003 osv. t.o.m. BFFF. Slutligen aktiveras, på motsvarande sätt, utporten för varje adress i adressintervallet 8000 t.o.m. 9FFF.




10.8 ADRESSRUMMET VID OFULLSTÄNDIG ADRESSAVKODNING

*Exempel 10.1*

Vi har ett system med 16 bitars adressbuss och 8 bitars databuss. Till centralenheten vill vi ansluta en ROM-modul (32 kbyte), en RWM-modul (16 kbyte) och en I/O-port (en inport och en utport). I/O-porten skall placeras på adress $4000, ROM-modulen placeras med start på adress $8000, RWM-modulen placeras med start på adress 0000 i adressrummet.

Konstruera adressavkodningslogiken, dvs. ange booleska uttryck för ”chip select” (CS)- signalerna. Använd ofullständig adressavkodning. Alla ”chip select” signaler ($CS_{ROM}$, $CS_{RWM}$, $CS_{IN}$ och $CS_{UT}$) är aktiva låga.

*Lösning:*

Vi har ett enkelt sätt att bestämma det antal adressbitar som krävs utgående från storleken av ett tilldelat adressutrymme, kom bara ihåg att 1 kbyte = $2^{10}$ och skriv:

32 kbyte = 32● $2^{10}$ = $2^5$ ● $2^{10}$ = $2^{15}$, dvs. 15 adressledningar kopplas direkt till kapseln.
16 kbyte = 16● $2^{10}$ = $2^4$ ● $2^{10}$ = $2^{14}$, dvs. 14 adressledningar kopplas direkt till kapseln.

Vi skapar en tabell med adressbussens ledningar (16 st.) och de ingående modulerna, vi bestämmer adressutrymmen för respektive modul enligt följande (startadress + storlek = slutadress+1):

ROM: (32 kbyte = $8000), slutadress = $8000+$8000-1 = $10000-1 = **$FFFF**
RWM: (16 kbyte =$4000), slutadress = 0+$4000-1 = **$3FFF**
I/O-port: ( 1 Byte ), slutadress = $4000+1-1 = **$4000**

Modulernas start respektive slutadresser förs in i tabellen enligt följande:

| Modul | | Adressbuss | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| ROM | $FFFF | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $8000 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RWM | $3FFF | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IO-port | $4000 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | $4000 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

En ofullständig adressavkodning innebär att vi vill använda så få bitar som möjligt för att därigenom minimera adressavkodningslogiken. Vi börjar därför med att inspektera den mest signifikanta adressbiten (A15) och därefter successivt inspektera adressbitar med lägre ordning (A14, A13, A12 osv.). Då vi ser att adressbitarna bara förekommer för en enda modul har vi också tillräckligt många bitar för adressavkodningen.





Vi börjar med ROM-modulen och ser att detta är den enda kapsel som kräver att A15 =1. För både RWM och I/O-port ska denna adressbit vara 0, det räcker därför med att använda enbart denna bit i adressavkodningen för $CS_{ROM}$.
CS- signalerna ska vara aktivt låga, varför vi får: $\overline{CS_{ROM}} = \overline{A15}$.

I tabellen ser vi att vi måste ta till både A15 och A14 för att skilja RWM-modulen från I/O-modulen. För RWM-modulen gäller att både A15 och A14 måste vara 0 och vi får därför: $\overline{CS_{RWM}} = \overline{\overline{A15} \cdot \overline{A14}}$.

| Modul | | Adressbuss | | |
|---|---|---|---|---|
| | | A15 | A14 | A13.... |
| ROM | $FFFF | 1 | 1 | 1 |
| | $8000 | 1 | 0 | 0 |
| RWM | $3FFF | 0 | 0 | 1 |
| | 0 | 0 | 0 | 0 |
| IO-port | $4000 | 0 | 1 | 0 |
| | $4000 | 0 | 1 | 0 |

För IO-porten gäller att A15 ska vara 0 och A14 ska vara 1.

Samtidigt observerar vi att porten är dubbelriktad, vi använder därför $R/\overline{W}$-signalen som en extra adressledning och aktiverar $CS_{IN}$ vid en läsning från adress $4000, respektive $CS_{UT}$ vid en skrivning till adress $4000. För IO-porten får vi därför: $\overline{CS_{IN}} = \overline{\overline{A15} \cdot A14 \cdot R/\overline{W}}$ och $\overline{CS_{UT}} = \overline{\overline{A15} \cdot A14 \cdot \overline{R/\overline{W}}}$.

Vi sammanfattar resultaten:

$$\overline{CS_{ROM}} = \overline{A15}$$
$$\overline{CS_{RWM}} = \overline{\overline{A15} \cdot \overline{A14}}$$
$$\overline{CS_{IN}} = \overline{\overline{A15} \cdot A14 \cdot R/\overline{W}}$$
$$\overline{CS_{UT}} = \overline{\overline{A15} \cdot A14 \cdot \overline{R/\overline{W}}}$$

### 10.2.2 Fullständig adressavkodning

Vid fullständig adressavkodning används alla adressbitar som krävs för att aktivera en modul enbart inom dess avsedda adressområde. Här tillåter man alltså ingen replikering av adressutrymmet för minneskapslar eller I/O-portar. Det följer direkt att adressavkodningen kommer att kräva fler (större) grindar för att generera CS-signalerna.

*Exempel 10.2*

En fullständig adressavkodning innebär att samtliga bitar som inte direkt krävs för att adressera modulen internt används i avkodningslogiken. Om vi valt att i stället konstruera fullständig avkodningslogik i Exempel 10.1 hade lösningen sett ut på följande sätt:

| Modul | | Adressbuss | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| ROM | $FFFF | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $8000 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RWM | $3FFF | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IO-port | $4000 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | $4000 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Avkodningslogiken blir:

$$\overline{CS_{ROM}} = \overline{A15}$$

$$\overline{CS_{RWM}} = \overline{\overline{A15} \cdot \overline{A14}}$$

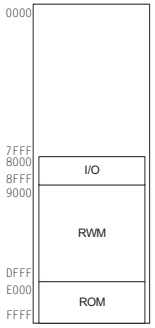
$$\overline{CS_{IN}} = \overline{\overline{A15} \cdot A14 \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \overline{A7} \cdot \overline{A6} \cdot \overline{A5} \cdot \overline{A4} \cdot \overline{A3} \cdot \overline{A2} \cdot \overline{A1} \cdot \overline{A0} \cdot R/\overline{W}}$$

$$\overline{CS_{UT}} = \overline{\overline{A15} \cdot A14 \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \overline{A7} \cdot \overline{A6} \cdot \overline{A5} \cdot \overline{A4} \cdot \overline{A3} \cdot \overline{A2} \cdot \overline{A1} \cdot \overline{A0} \cdot \overline{R/\overline{W}}}$$

Det framgår att minneskapslarna i själva verket redan är fullständigt avkodade medan avkodningslogiken för portarna blir betydligt mer omfattande.

Fullständig adressavkodning innebär att adressrummet utnyttjas maximalt. I vårt systemexempel (Exempel 10.1 och Exempel 10.2) kan vi exempelvis bygga ut med fler portar eller eventuellt någon mindre minneskrets om vi använt fullständig adressavkodning redan från början.

### Exempel 10.3




Vi har ett system med 16 bitars adressbuss och 8 bitars databuss. Konstruera fullständig adressavkodningslogik (ange booleska uttryck för CS-signalerna) så att adressrummet disponeras enligt figuren till vänster.

Alla ”chip select” signaler är aktiva låga.

*Lösning:*

Start och slutadresser framgår av figuren och vi kan ställa upp tabellen direkt:

| Modul | | Adressbuss | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| ROM | $FFFF | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $E000 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RWM | $DFFF | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $9000 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IO | $8FFF | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $8000 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

De booleska uttrycken för ”chip select” signalerna blir:

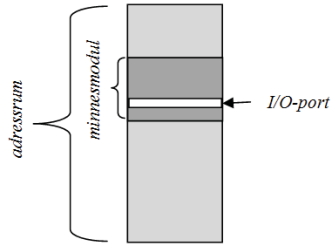
$$\overline{CS_{ROM}} = \overline{A15 \cdot A14 \cdot A13}$$

$$\overline{CS_{RWM}} = \overline{A15 \cdot \overline{A13} \cdot A12}$$

$$\overline{CS_{IO}} = \overline{A15 \cdot \overline{A14} \cdot \overline{A13} \cdot \overline{A12}}$$

### 10.2.3 Överlagrad minnesavbildning

I sammanhang där bara enstaka I/O-portar behövs och man i stället vill maximera det tillgängliga minnet kan man använda en överlagringsteknik där I/O-porten aktiveras i en del av adressrummet för någon minneskrets.




FIGUR 10.9 I/O-PORT I MINNESMODULS ADRESSRUM

Tekniken innebär att man först skapar en fullständig adressavkodning för I/O-porten och därefter använder dess invers som kompletterande villkor för minnesmodulens CS-signal.





### Exempel 10.4

Vi har ett system med 20 bitars adressbuss och 8 bitars databuss. Till centralenheten finns ansluten en 128 kbytes ROM-modul med start på adress $20000. En I/O-port, som upptar 32 bytes, ska överlagras på adress $22000.

Konstruera adressavkodningslogiken, dvs. ange booleska uttryck för ”chip select” (CS)- signalerna. Båda ”chip select” signalerna ($CS_{ROM}$ och $CS_{IO}$) är aktiva låga.

*Lösning:*

Bestäm start och slutadresser och ställ upp en tabell:

| Modul | | Adressbuss | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A19 | A18 | A17 | A16 | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| ROM | $3FFFF | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $20000 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IO-port | $2201F | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| | $22000 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Ur tabellen får vi direkt den fullständiga avkodningslogiken för I/O-porten:

$$\overline{CS_{IO}} = \overline{A19 \cdot \overline{A18} \cdot A17 \cdot \overline{A16} \cdot \overline{A15} \cdot \overline{A14} \cdot A13 \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \overline{A7} \cdot \overline{A6} \cdot \overline{A5}}$$

Avkodningslogiken för ROM-kapseln fås på samma sätt men vi grindar dessutom in inversen av $\overline{CS_{IO}}$.

$$\overline{CS_{ROM}} = \overline{A19 \cdot \overline{A18} \cdot A17 \cdot CS_{IO}}$$

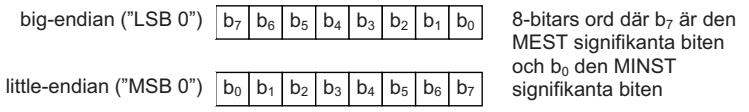
## 10.3 Buss-kommunikation

Datorns buss-kommunikation måste vara fastlagd i ett *protokoll*. Med protokoll menas helt enkelt regler för hur kommunikationen sker, dvs. hur information, *data*, utbytes mellan olika enheter i systemet. Regelverket inbegriper alltså överenskommelser om:

- ”Språket”, betydelsen av hög respektive låg logiknivå
- Organisation av bitarna i ett ord, ”endian bit order”
- Organisation av bytes i ett ord ”endian byte order”
- Tidsegenskaper, händelser korrekt ordnade i tid

I ett digitalt datorsystem är ”språket” enkelt, vi har logiknivån 1 och logiknivån 0. I bland betyder ”1” sant, annars betyder det falskt. Enda återstående alternativet är att ”1” betyder falskt och sålunda ”0” betyder sant. Fler kombinationer finns inte. För att ytterligare förenkla detta har vi infört uttrycket ”aktiv”. Med ”aktiv hög” menar vi att en signal ska betraktas som ”sann” om den är 1. Med uttrycket ”aktiv låg” är det precis tvärtom.

Då ettor och nollor sätts samman till kompletta ord som utväxlas via bussen måste man också ha fastlagt i vilken ordning bitarna lagras. Vi känner till exempel till formen där den mest signifikanta biten placeras i ordets första bit osv. med den minst signifikanta biten i ordets sista bit. Denna form kalla ”big-endian bit order”. Ordningen kan växlas, så att den minst signifikanta biten lagras i den första positionen och den mest signifikanta biten lagras i den sista positionen, Detta kallas då ”little-endian bit order”, se Figur 10.10.

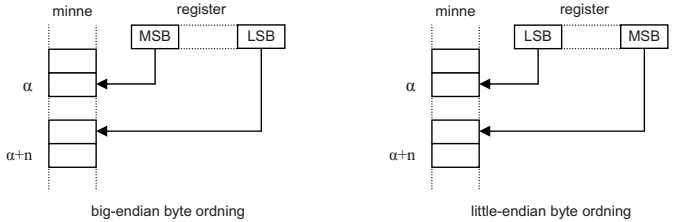



FIGUR 10.10 ”BIG/LITTLE –ENDIAN BIT ORDER”

### Exempel 10.5 Bit-ordning

**big-endian** bit ordning är det absolute vanligast förekommande, bland andra Freescale 68xx, 68xxx, ColdFire, Intel x86 och ARM. Det finns också processorer där bit-ordningen är programmerbar, exempelvis PowerPC.

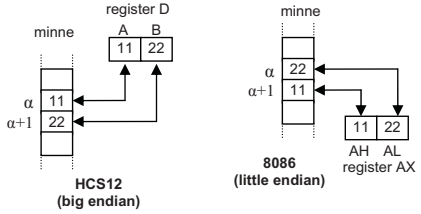
På motsvarande sätt kan delarna av ett ord bestående av flera bytes, organiseras med antingen den mest signifikanta byten först (*Most Significant Byte, MSB*), ”big endian”, eller den minst signifikanta byten först (*Least Significant Byte, LSB*), ”little endian”. Med ”först” menas då den lägsta minnesadressen. Se Figur 10.11




FIGUR 10.11 ORGANISATION VID ”BIG/LITTLE-ENDIAN BYTE ORDER”, *n*-BYTES ORD MED STARTADRESS α

### Exempel 10.6 Motorola HCS12/Intel 8086 byte ordning

Det hexadecimala ordet $(1122)_{16}$ uppträder i minnet (adress α) respektive register enligt följande figurer:



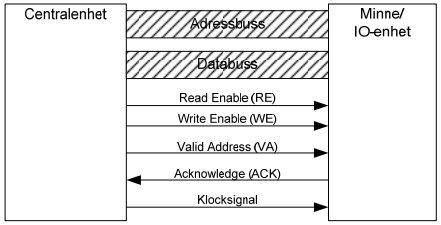

Den sista viktiga aspekten på ett buss-protokoll är dess tidsmässiga egenskaper, dvs. att data utväxlas i en arbetstakt som är gemensam för alla enheter på bussen. Här skiljer vi mellan tre principiellt olika metoder, asynkron-, synkron- och multiplex-buss. Vi behandlar dessa i de följande avsnitten.

### 10.3.1 Asynkron buss

Med en *asynkron* buss används så kallade *handskakningssignaler* för kommunikationen (dataöverföringen). Med detta menas att centralenheten signalerar till de övriga enheterna att en adress nu finns på bussen genom att aktivera en speciell signal, *Valid Address* (VA). Övriga enheter kan läsa av adressbussen och därefter måste den enhet som adresseras generera en annan signal, *Acknowledge* (ACK) tillbaks till centralenheten, för att på så sätt tala om för centralenheten att adressen är igenkänd. Denna signal tolkas av centralenheten som att dataöverföringen nu kan ske. Följaktligen inväntar centralenheten svar från den enhet den kommunicerar med (Figur 10.12). Signalerna VA och ACK ingår i styrbussen.

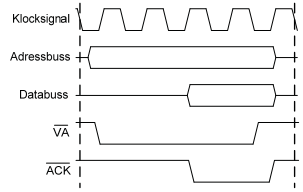







FIGUR 10.12 ASYNKRONT BUSSPROTOKOLL

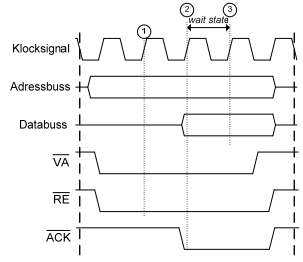
Figur 10.13 illustrerar principen för överföring via en asynkronbuss. Observera att de signaler som ingår i styrbussen är aktivt låga. Då periferienheten upptäckt den aktiva VA-signalen kommer den att avkoda adressbussen. Vid en läsning från minnet kommer periferienheten att placera data på databussen och därefter aktivera signalen ACK. När centralenheten upptäckt en aktiv ACK-signal läser den in data från databussen. Centralenheten indikerar sedan att den läst databussen genom att återställa VA. Slutligen avlägsnar centralenheten adressen på adressbussen. Periferienheten som ser att VA inte längre är aktiv avlägsnar ACK-signalen och data från databussen.




FIGUR 10.13 SIGNALERING VID ÖVERFÖRING PÅ EN ASYNKRON BUSS

Protokollet fungerar på liknande sätt vid en skrivning till periferienheten. I detta fall innebär VA-signalen att såväl innehållet på adressbussen som på databussen är giltigt. Periferienheten avkodar, precis som tidigare, adressbussen, och läser därefter databussens innehåll. Slutligen aktiverar periferienheten ACK-signalen som signal till centralenheten att skrivcykeln kan avslutas.

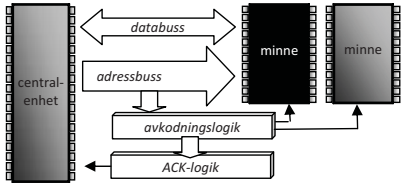
Vår principiella beskrivning i Figur 10.13 är visserligen korrekt men inte fullständig. I verkligheten sker alla tillståndsövergångar då systemet klockas, vanligtvis vid en positiv klockflank. Det innebär att signaländringar på den asynkrona bussen inte är momentant kända. Låt oss illustrera detta med en ny bild av hur en läscykel kan gå till (Figur 10.14).




FIGUR 10.14 LÄSCYKEL MED ASYNKRON BUSS

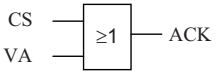
Vid ”1” börjar periferienheten avkoda adressen från adressbussen, detta tar en kort stund, varefter databussen drivs av periferienheten och ACK-signalen aktiveras. ACK-signalen upptäcks av centralenheten vid ”2” och databussen läses av vid nästa positiva klockflank ”3”. Antalet klockcykler från det att centralenheten upptäcker ACK, tills data klockas in från databussen kallas *väntecykler* (”*wait states*”).

Centralenhetens kommunikation med minnesenheter måste oftast utformas på speciellt sätt eftersom minneskretsar normalt inte är asynkrona, dvs. de genererar ingen ACK-signal. Man måste då införa extra logik som genererar ACK-signal till centralenheten när minnet adresseras.



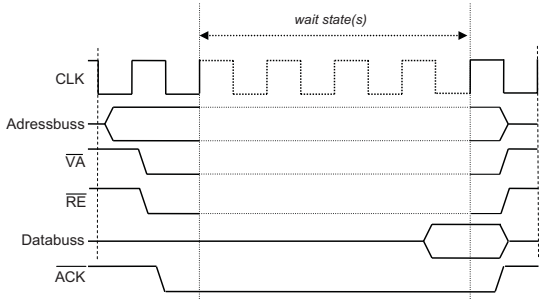

FIGUR 10.15 INKOPPLING AV MINNE SOM SAKNAR ACK-SIGNAL TILL ASYNKRON BUSS

ACK-logiken måste alltså utformas på så sätt att tidsegenskaperna hos det använda minnet respekteras. Detta kräver då att logiken blir speciell för någon familj av minneskretsar. Dessbättre är detta inget större problem i praktiken eftersom centralenheter som arbetar med asynkronbuss oftast kan programmeras för att sätta in väntecykler för olika regioner av adressrummet. ACK-logiken kan då utformas extremt enkel (Figur 10.16) medan tidsegenskaperna kan uppfylls genom en lämpligt vald initieringssekvens för centralenheten (Figur 10.17).




där CS genereras av adressavkodningslogiken för minneskapseln

FIGUR 10.16 FÖRENKLAD ACK-LOGIK FÖR MINNESKRETS




FIGUR 10.17 PROGRAMMERBART ANTAL VÄNTECYKLER

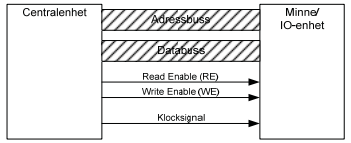
De flesta centralenheter med asynkron buss bevakar att VA-signalen inte är aktiv för länge, typiskt något hundratal klockcykler vilket är tillräckligt för att alla olika typer av kringenheter ska kunna hinna svara. Vid utebliven ACK-signal, dvs adressbussen anger en adress som inte finns representerad hos någon annan enhet kommer ett internfel ”bus error” att genereras.

### 10.3.2 Synkron buss

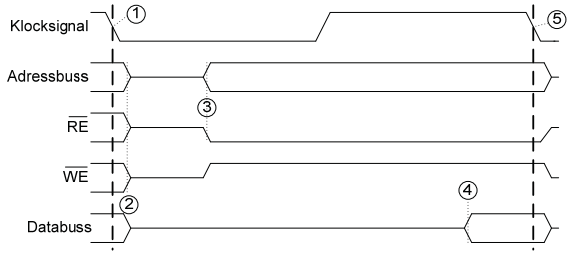
En buss sägs arbeta *synkront* om signalerna överförs vid en förutbestämd tidpunkt. Fördelen med synkron buss är att handskakningssignaler inte behövs längre. Nackdelen är att det kan vara svårt att blanda olika typer av minnen etc. om dessa har olika tidsegenskaper. Vid dataöverföring på en synkron buss *förutsätter* centralenheten att den andra enheten hinner med i den arbetstakt som används och det krävs inte längre någon ACK- signal.





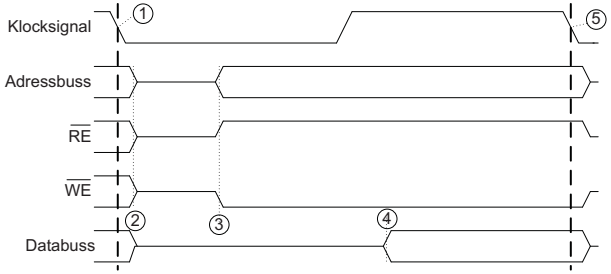



FIGUR 10.18 SIGNALER VID ÖVERFÖRING PÅ SYNKRON BUSS




FIGUR 10.19 LÄSCYKEL PÅ SYNKRON BUSS

1. Läscykeln inleds med en negativ flank hos klocksignalen.
2. Signaler som var giltiga under föregående cykel tas bort och bussarna är nu i odefinierade tillstånd.
3. Centralenheten sätter upp adress och styrsignaler för minnet.
4. Minnet har avkodat adressbussen och lägger ut data på databussen.
5. Läscykeln avslutas med en negativ flank hos klocksignalen. Data klockas in till centralenheten från databussen.

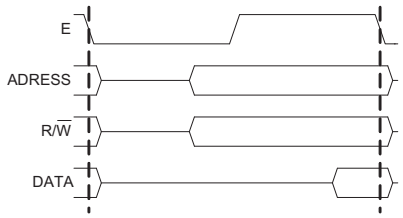



FIGUR 10.20 SKRIVCYKEL PÅ SYNKRON BUSS

1. Skrivcykeln inleds med en negativ flank hos klocksignalen.
2. Signaler som var giltiga under föregående cykel tas bort och bussarna är nu i odefinierade tillstånd.
3. Centralenheten sätter upp adress och styrsignaler för minnet.
4. Centralenheten lägger ut data på databussen, adressbussen avkodas och aktiverar minneskapsel.
5. Skrivcykeln avslutas med en negativ flank hos klocksignalen. Data klockas in till minnet från databussen.

*Exempel 10.8*

Vi har ett synkront system med 16 bitars adressbuss och 8 bitars databuss. Data klockas i systemet vid negativ flank hos signalen E.




Till centralenheten ska anslutas:

- 8 kbyte ROM med start på adress $E000
- 16 kbyte RWM med start på adress $8000
- 512 byte I/O, med start på adress $E000

Konstruera fullständig adressavkodningslogik, dvs. ange booleska uttryck för ”chip select” (CS)- signalerna. Alla ”chip select” signalerna ($CS_{ROM}$, $CS_{RWM}$ och $CS_{IO}$) är aktiva låga.

*Lösning:*

Eftersom avkodningen ska vara fullständig bestämmer vi först antalet adressledningar som krävs för de respektive modulerna:

ROM: 8 kbyte = 8● $2^{10} = 2^3$ ● $2^{10} = 2^{13}$, dvs. 13 adressledningar kopplas direkt till ROM-kapseln.
RWM: 16 kbyte = 16● $2^{10} = 2^4$ ● $2^{10} = 2^{14}$, dvs. 14 adressledningar kopplas direkt till RWM-kapseln.
I/O: 512 byte = $2^9$, dvs. 9 adressledningar används för en generell ”I/O-select” signal.

Nu för vi in start- och slutadresser i en tabell:

| Modul | | Adressbuss | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| ROM | $FFFF | 1 | 1 | 1 | Kopplas direkt till ROM-kapsel | | | | | | | | | | | | |
| | $E000 | 1 | 1 | 1 | | | | | | | | | | | | | |
| RWM | $BFFF | 1 | 0 | Kopplas direkt till RWM-kapsel | | | | | | | | | | | | | |
| | $8000 | 1 | 0 | | | | | | | | | | | | | | |
| I/O | $E1FF | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Används för att bilda ytterligare CS-signaler för enskilda I/O-enheter | | | | | | | | |
| | $E000 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | | | | | | | | | |

Av tidsdiagrammet över bussens karakteristik kan vi utläsa att adressbuss och R/W-signal är giltig åtminstone den tid som E-signalen är hög. Vi använder därför denna som en *Valid Address*-signal, dvs VA=E (Anm: VA är alltså aktiv hög).

Vi börjar med I/O-blockets generella CS-signal:

$$\overline{CS_{IO}} = \overline{A15 \cdot A14 \cdot A13 \cdot \overline{A12} \cdot \overline{A12} \cdot \overline{A10} \cdot \overline{A9} \cdot VA}$$

därefter får vi enkelt CS-signalen för ROM-modulen:

$$\overline{CS_{ROM}} = \overline{A15 \cdot A14 \cdot A13 \cdot \overline{CS_{IO}}}$$

slutligen RWM-modulen:

$$\overline{CS_{RWM}} = \overline{A15 \cdot \overline{A14} \cdot VA}$$

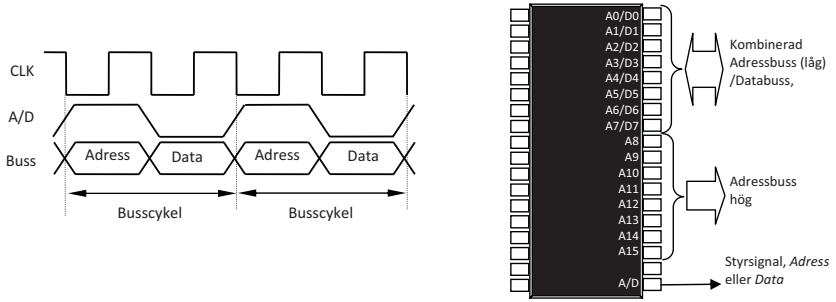




### 10.3.3 Multiplexad buss

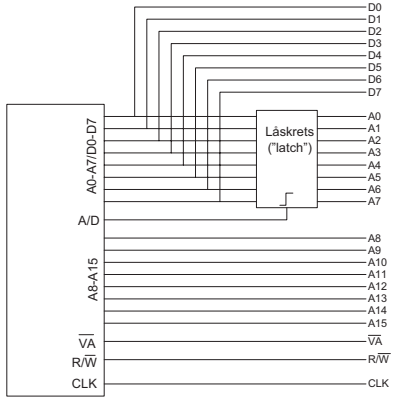
Vanligtvis är adressbussar och databussar fysiskt separerade. För att reducera antalet pinnar (signaler) i en processor’s kapsel kan man använda *multiplexad* buss. Med detta menas att bussen används för att växelvis överföra adresser respektive data.

I Figur 10.21 illustreras ett multiplexat system med 16 bitars adressbuss och 8 bitars databuss och med den *fysiska* busbredden 16 bitar. (Användes bland annat i den första 8 bitars mikroprocessorn Intel 8080). Som figuren visar finns en signal A/D som anger huruvida bussen används för att överföra adress eller data. Hela adressen överförs under den första busscykeln. Under den andra busscykeln används bara de 8 minst signifikanta bitarna för data.




FIGUR 10.21 ÖVERFÖRING MED EN MULTIPLEXAD BUSS, 16-BITAR ADRESS, 8-BITAR DATA

I Figur 10.22 visas hur en multiplexad processor kopplas samman med periferienheter (minne). Observera speciellt låskretsen, då A/D signalen växlar från låg till hög nivå, ”låses” innehållet på kretsens ingångar, dessa utgör då adressbussens värde för A0-A7. I nästa klockcykel, går A/D-signalen låg och indikerar att data nu finns (vid skrivning) eller ska läggas ut av minnet (vid läsning).




FIGUR 10.22 GRÄNSSNITT MELLAN MULTIPLEXAD PROCESSOR OCH MINNE

## 10.4 Undantagshantering

Med ”undantag” (exception) menar vi speciella händelser som föranleder avbrott i sekventiellt utförande av instruktioner. Sådana händelser kan exempelvis vara någon form av interna fel som uppstår under instruktionsexekvering och som gör att instruktionen inte kan fullföljas men processorn kan också fås att reagera på signaler från omgivningen. Med *extern styrning* menas sådana signaler som kan aktiveras utifrån och påverka processorn närsomhelst, oberoende av programexekveringen.

Vi säger att processorn alltid befinner sig i något av tillstånden *normal* eller *exception*.

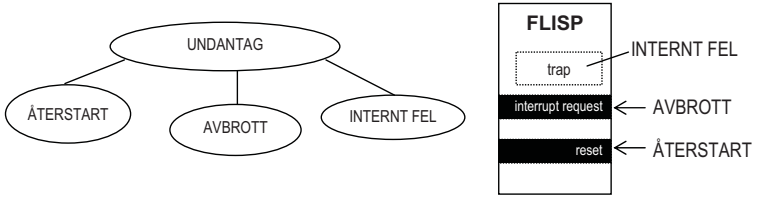
- *Normal*, processorn hämtar och utför instruktioner, dvs. normal exekvering.
- *Exception*, något “undantag” har inträffat som gör att processorn inte kan (eller ska) fortsätta normal exekvering.

Vi använder begreppet ”undantagshantering” (*exception handling*) för processorns sätt att utföra de moment som krävs för att växla mellan de olika tillstånden. Undantagshanteringen är mycket likartad för olika untantagstyper och skiljer sig endast i detaljerna. Dessa händelser kan delas in i tre olika grupper, se även Figur 10.23:

*Återstart ("reset")*, händelser som alltid föranleder återstart av processorn. I FLISP signaleras detta via en speciell ingång hos processorn.

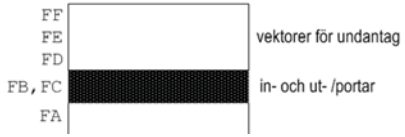
*Avbrott ("interrupt")*, externa händelser, som dock inte innebär att processorn ska återstartas. Även dessa händelser signaleras via en speciell ingång, IRQ ("*interrupt request*").

*Internt fel ("traps")*, händelser som uppträder vid utförandet av en instruktion och som gör att instruktionen inte kan fullföljas.




FIGUR 10.23 UNDANTAGSTYPER I FLISP

Varje undantagstyp har sin egen *undantagsvektor*, där startadressen till undantagets hanteringsrutin ska finnas. Hos Flisp har dessa vektorer placerats längst upp i adressrummet, se Figur 10.24, på adresser $FD_{16}$, $FE_{16}$ och $FF_{16}$.



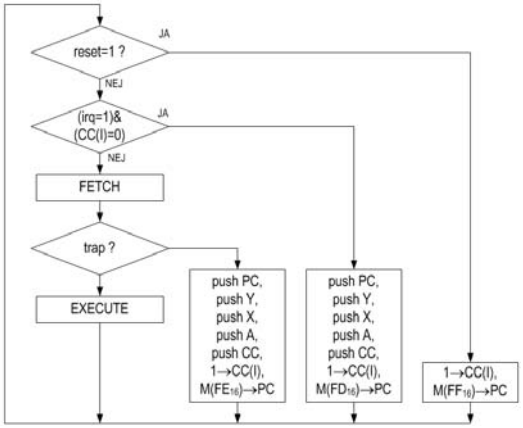

| | |
|---|---|
| $FF_{16}$ | ”reset”, återstart |
| $FE_{16}$ | ”trap”, undantag |
| $FD_{16}$ | ”interrupt”, avbrott |

FIGUR 10.24 UNDANTAGSVEKTORER HOS FLISP



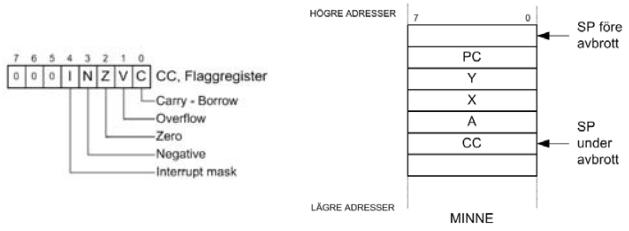


Övergången mellan de olika tillstånden *normal*/*exception* sker vid olika tillfällen beroende på undantagets art. I Figur 10.25 visas hur de olika kontrollerna sker, figuren visar också likheter och skillnader mellan de olika undantagshanteringarna. I figuren ser vi att undantagshantering för *reset* och *trap* är ovillkorliga medan undantagshantering för *irq* bara utförs om den så kallade avbrottsmasken, I-flaggan i CC-registret, är 0. Om I-flaggan är 1 säger vi att avbrottet "maskerats" och därför inte ska betjänas.




FIGUR 10.25 TILLSTÅNDSÖVERGÅNG NORMAL TILL EXCEPTION

Då undantagshantering för avbrott eller internt fel, påbörjas sparas först samtliga registerinnehåll på stacken. Bit I i CC sätts därefter till 1 för att förhindra ytterligare ett omedelbart avbrott. Slutligen laddas PC med adressen till hanteringsrutinen från vektorn för undantagshanteringen. I Figur 10.26 visas Flis-processorns flaggregister med I-flaggan (bit 4). Figuren visar också hur registerinnehållen placerats på stacken inför utförandet av avbrottsrutinen. Vid avbrott innehåller PC adressen till den instruktion som skulle ha utförts om inget avbrott signalerats. Vid internt fel innehåller PC adressen till instruktionen omedelbart efter den instruktion som orsakat det interna felet.




FIGUR 10.26 AVBROTTSMASK (I-FLAGGA) OCH STACKENS UTSEENDE VID AVBROTT/INTERNT FEL

Vi återgår till Figur 10.25, vid återstart ser vi hur Flisp först ettställer bit I i CC, därefter läser innehållet på adress $FF_{16}$ och placerar detta i PC. Vi känner igen återstartsförloppet från kapitlet om styrenheten med tillägget att I-flaggan ges ett definierat begynnelsevärde. Det är nödvändigt att återställningfasen också maskerar avbrott på detta sätt, annars skulle en aktiv avbrottsignal kunna signalera avbrottsbegäran innan systemet föreberetts för avbrottshantering. Då flera undantag uppträder samtidigt avgör *undantagsarbitreringen* vad som ska göras. Detta illustreras också av flödesschemat i figuren genom ordningen som kontrollerna utförs.

### 10.4.1 Kodning av hanteringsrutiner

Då en hanteringsrutin för avbrott eller trap avslutas vill man ofta återställa alla registerinnehåll från stacken, för detta finns instruktionen RTI

*Exempel 10.10*

Här visas hur man skapar de nödvändiga förutsättningarna för undantagshantering i form av en kort startsekvens följd av anrop till ett huvudprogram.

```
            ORG   $20
start:
            LDSP          #$20
            ANDCC         #%11101111    ; nollställ I-flagga
restart:
            JSR           main
            BRA           restart
```

Här placeras programkoden för huvudprogrammet.

```
main:
```

Vi tillhandahåller även hanteringsrutiner för avbrott och interna fel, ingenting utförs av hanteringsrutinerna i detta exempel.

```
irqhandler:
            RTI

traphandler:
            RTI
```

Undantagsvektorer

```
            ORG   $FD
            FCB   irqhandler      ; hanteringsrutin för avbrott
            FCB   traphandler     ; hanteringsrutin för interna fel
            FCB   start           ; återstart
```





### 10.4.2 Interna fel

Interna fel uppstår då instruktionsexekvering inte kan fullföljas. Några exempel på interna fel, i det generella fallet är:

- *Division med 0*, kan inträffa i processorer som har maskininstruktioner för division och eftersom operationen inte är definierad kan den normalt inte heller fullföljas.
- Om processorn avkodar en *otillåten operationskod* kan inte denna utföras som en instruktion, detta klassificeras då som ett internt fel. I bland kallas det också för *trap*, eller *software interrupt*, eftersom man kan få processorn att utföra programmerad undantagshantering genom att placera en otillåten operationskod i programmet.
- *Adressfel*, processorer har oftast mycket stora adressrum som inte fullständigt bestyckats med minne och periferikretsar. Om processorn gör ett försök att hämta en instruktion eller data från en sådan adress kan detta upptäckas och utlösa adressfel.
- *Bussfel*, kan inträffa hos processorer som kräver att vissa datatyper lagras på visst sätt i minnet. Exempelvis kan man kräva att när en 32-bitars datatyp lagras i ett byte-organiserat minne får detta bara ske på adresser som är jämnt delbara med 4 (*alignment condition*).

Hos FLISP kan interna fel uppstå endast om en otillåten operationskod lästs in till instruktionsregistret. Följande exempel visar hur detta kan användas för att *emulera* en instruktion som inte finns i instruktionsuppsättningen.

*Exempel 10.11 Emulering av FLISP-instruktion*

I detta exempel visas hur en odefinierad operationskod (03) kan användas för att implementera en instruktion (`EXG A,X`) som normalt inte finns i instruktionsuppsättningen.

```
            ORG   $20
start:
            LDSP  #$20
            LDA   #$55
            LDX   #$88
            FCB   3           ; illegal opkod
stop        BRA   stop


traphandler:
            STA   2,SP
            STX   1,SP
            RTI

            ORG   $FE
            FCB   traphandler           ; hanteringsrutin för interna fel
            FCB   start           ; återstart
```