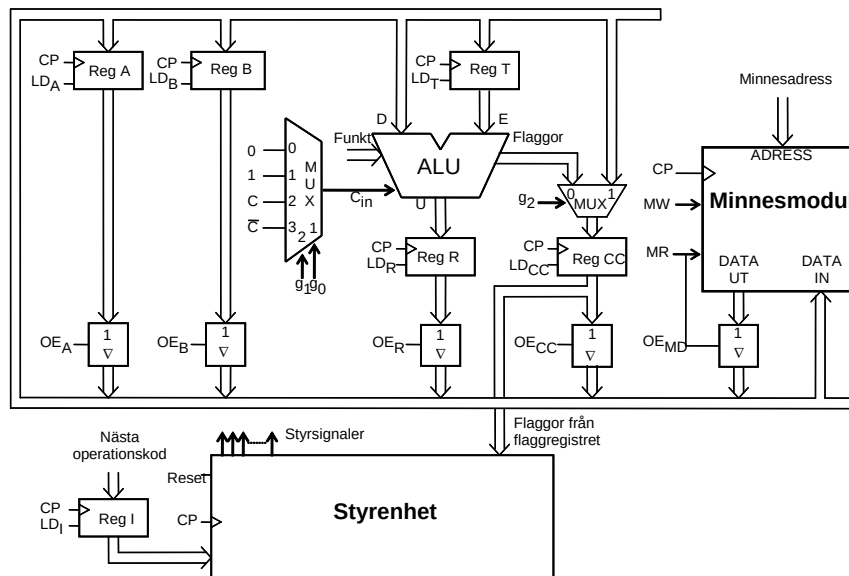


## FLX Datorn enligt von Neumann (≈ kap 10)

**Observera att FLIS-processorns instruktionsuppsättning och maskinspråk används i alla programexempel i detta häfte!**

Den databehandlande enheten med minne, från kapitel 7, återges nedan i figur F.1. System liknande det i figur F.1 var kända strax efter andra världskrigets slut. Man kunde vid denna tid konstruera processorer vars styrenheter var utformade för att lösa vissa speciella problem, som tex beräkningar av projektilbanor. Det fanns även processorer med modifierbara ("programmerbara") styrenheter. "Programmeringen" gick till så att man ställde in styrenhetens beteende genom att koppla om ledningar och ställa in switchar.



Figur F.1 Databehandlande enhet med minne.

### F.1 von Neumanns idé

Den amerikanske matematikern John von Neumann och hans medarbetare bidrog på detta stadium med en genial och mycket enkel idé:

*"Man kan koda de olika operationerna som binära tal på samma sätt som data och lagra både operationer och data i minnet." = "Det lagrade programmets princip."*

Data kan behandlas genom att en följd av instruktioner och data växelvis hämtas från minnet. Instruktionerna utför de önskade operationerna på data och skriver vid behov tillbaka data i minnet.

En instruktion är alltså kodad som ett binärt tal i vilket en del av bitarna representerar koden för en operation medan övriga bitar representerar en eller flera operander (data). En typisk instruktion består alltså av två delar enligt figuren till höger.

|               |             |
|---------------|-------------|
| Operationskod | Operand(er) |
|---------------|-------------|

Operanden är antingen *data* (datavärdet) eller någon form av *adress till data*. Vissa instruktioner saknar dock operand och därmed också operanddel.

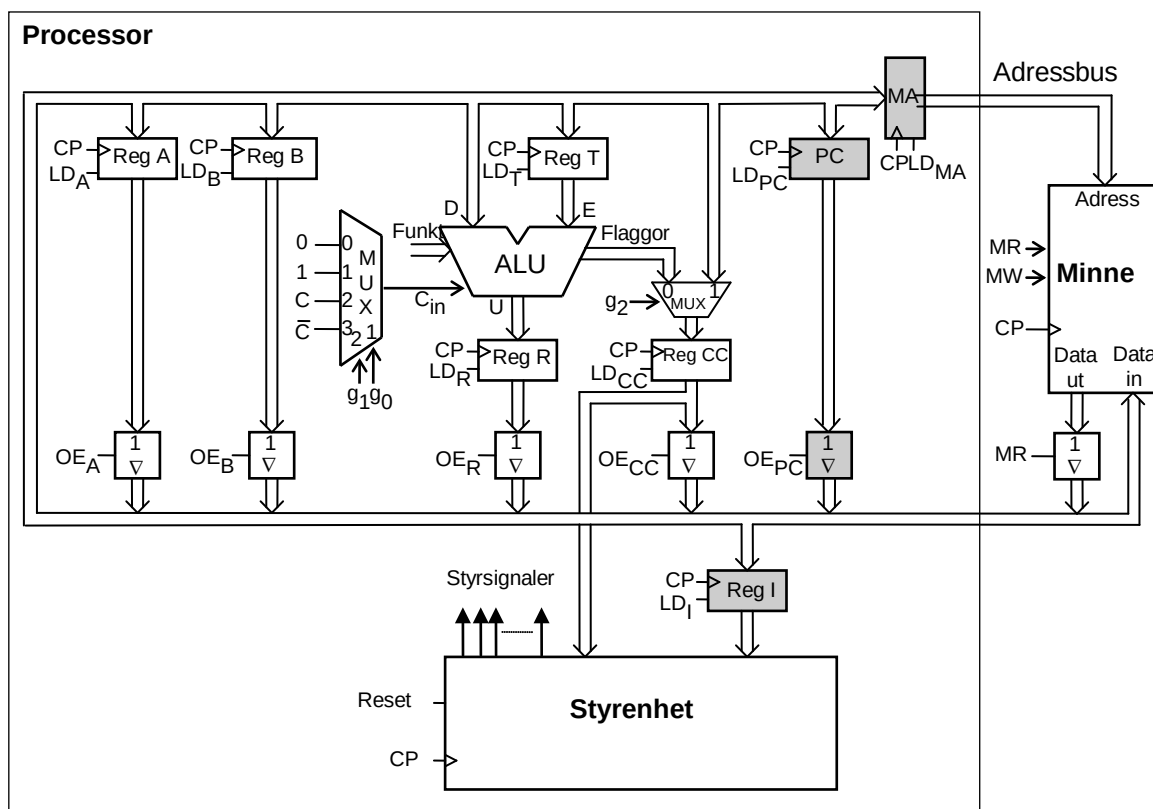
## F.2 von Neumannprocessorn

En komplett instruktionssekvens som utför en specifik databehandlingsuppgift kallas ett program. När ett program skall placeras i minnet är det praktiskt om instruktionerna hamnar på konsekutiva (på varandra följande) minnesadresser. Då programmet körs (exekveras) kommer därför processorn att hämta instruktionerna på konsekutiva adresser i minnet.

Det måste finnas en mekanism i processorn som skiljer mellan instruktioner och data eftersom man inte kan se någon skillnad på dem i minnet. Minnesadressen till den instruktion i programmet som står i tur att utföras lagras därför i ett nyinförd register i processorn. Registret kallas programräknaren (eng. program counter, PC) eftersom innehållet normalt ökas med ett för varje ny läsning av en instruktion i minnet. Se figur F.2.

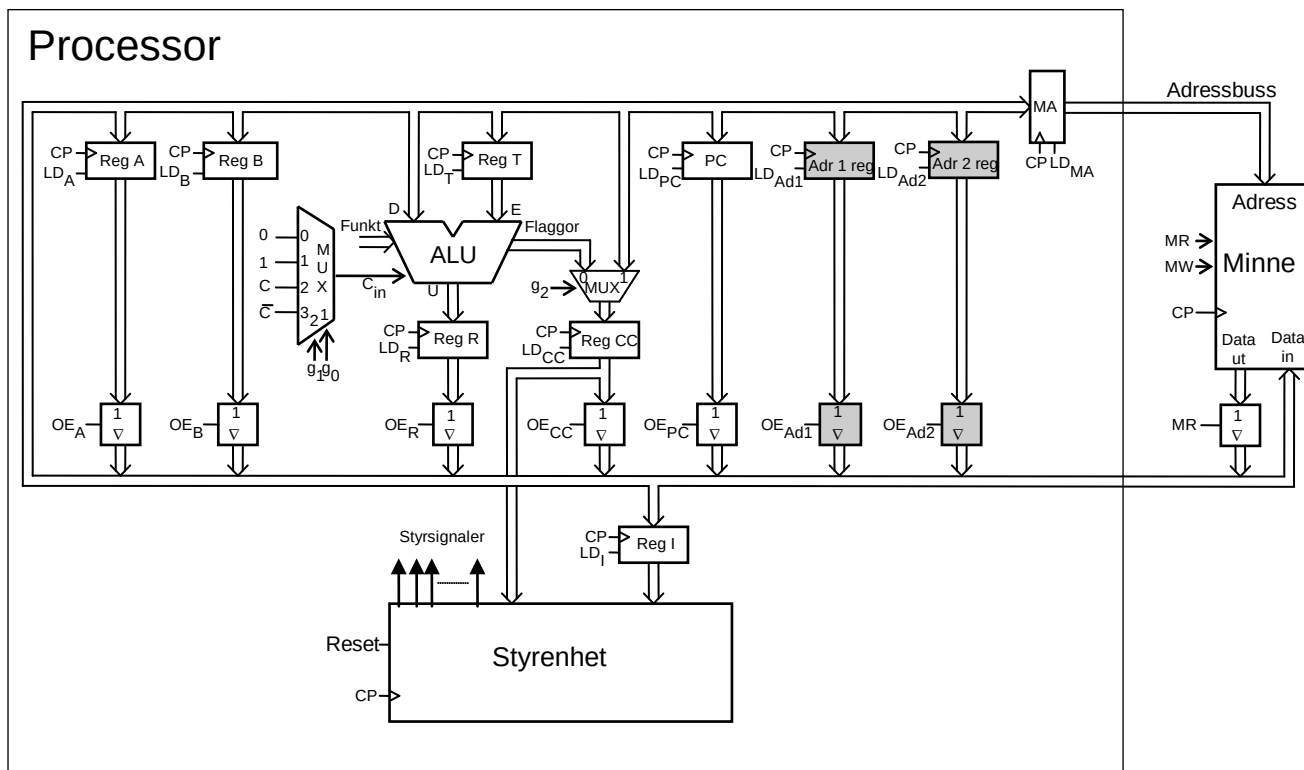
Då processorn skall läsa en instruktion i minnet måste först adressen som finns i PC placeras på minnets adressgång. Av detta skäl har ett minnesadressregister MA (eng **memory address register**) kopplats in mellan databussen och minnets adressgång.

Instruktionsregistrets (I) ingång har dessutom anslutits till databussen så att instruktionens operationskod kan laddas i I-registret då processorn läser instruktionen i minnet.



Figur F.2 Kombinationen von Neumannprocessor-minne.

Utöver läsning av programinstruktioner måste processorn kunna läsa och skriva data i minnet. Därför är det lämpligt att den också innehåller ett eller flera register för adresser till data. Figur F.3 nedan visar en von Neumannprocessor och minne, där von Neumannprocessorn från figur F.2 är kompletterad med två register för adresser till data.



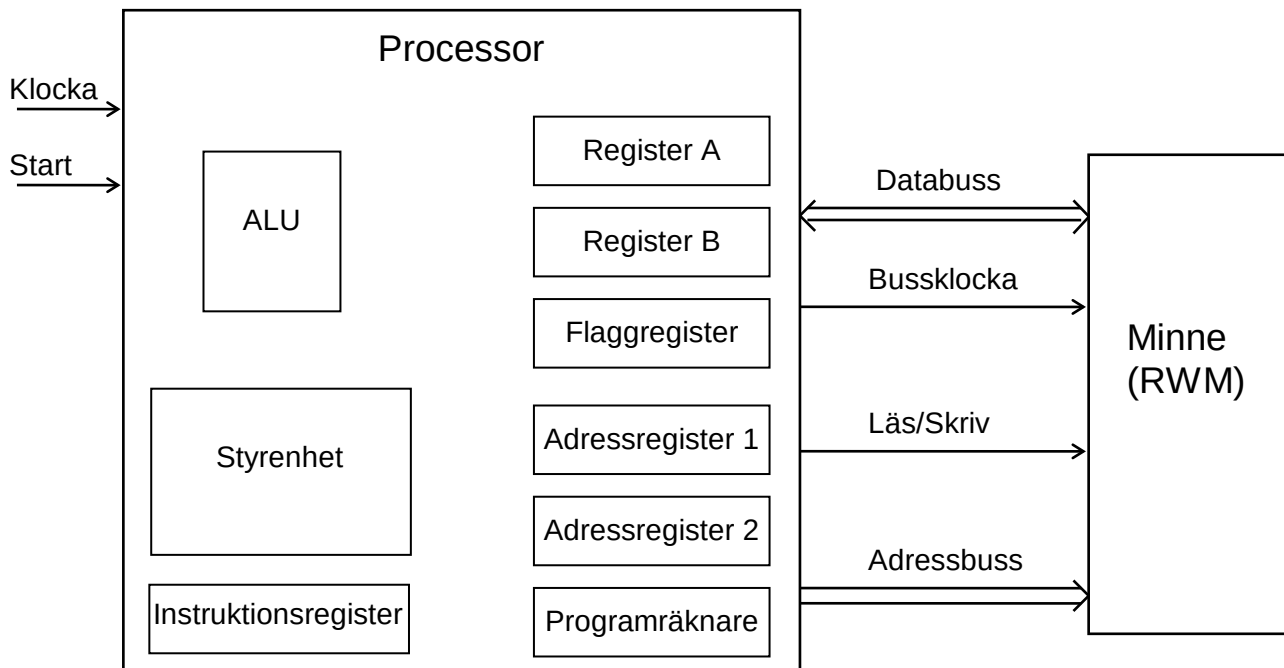
Figur F.3 Kombination av en utökad von Neumannprocessor och minne.

Kombinationen processor-minne i figurerna F.2 och F.3 kallas ofta von Neumann dator. Med ordet dator menar man egentligen ett system som består av processor, primärminne och in-/utenheter (I/O-enheter). Eftersom in-/utenheter saknas i figurerna visar de alltså inte kompletta datorer.

I vissa tillämpningar är det vanligt att man placerar program och konstanter i en minnestyp där man endast kan läsa data men inte skriva ny data. Denna minnestyp kallas läsminne (eng. read only memory, ROM). Variabler måste dock placeras i läs- och skrivbart minne (eng. read write memory, RWM), ofta oegentligt kallat "RAM", random access memory.

### F.3 Programmeringsmodell för von Neumanndator

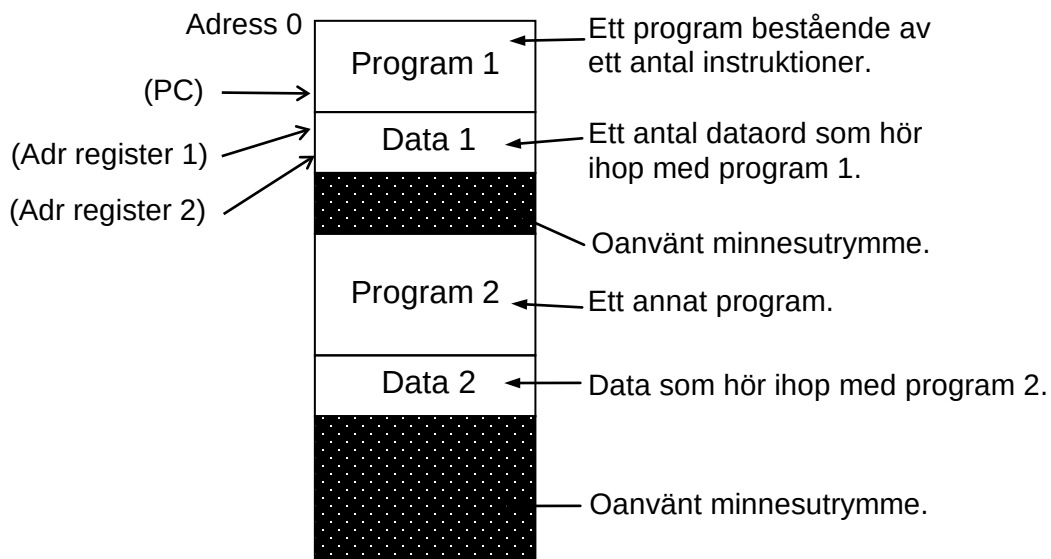
Bilden av von Neumanndatorn i figur F.3 innehåller en stor mängd detaljer som är onödiga att känna till för en person som endast skriver program för datorn. Man har därför infört en beskrivningsnivå som bara innehåller de detaljer som är av intresse för programmeraren. Figur F.4 nedan visar hur datorn i figur F.3 är uppbyggd sett med en programmerares ögon, **programmeringsmodellen**.



Figur F.4 Programmeringsmodell för enkel von Neumanndator.

I figur F.4 finns ett register med flaggbitar. 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. Villkorliga instruktioner utförs endast om det aktuella villkoret, t ex carry-flaggan = 1, är uppfyllt.

För att förenkla användandet av datorn och få program med överskådlig uppbyggnad försöker man strukturera användningen av minnet, ofta i en hierarkisk struktur. Figur F.5 visar exempel på en enkel sk minnesdisposition.



Figur F.5 Minnesdisposition för von Neumann-dator.

Programräknaren PC innehåller adressen till den instruktion som står i tur att utföras. Detta kan tolkas så att PC *pekar på* ett visst ställe i minnet och betecknas som i figur F.5. Parentesen runt PC, "(PC)", anger att man menar *innehållet i* PC. Eftersom PC pekar på ett ställe i Program 1 kan man dra slutsatsen att processorn håller på att köra Program 1.

På samma sätt pekar "Adressregister 1" på en viss position i minnet där ett dataord för program 1 finns lagrat, dvs "Adressregister 1" innehåller adressen till denna minnesposition..

#### F.4 Ordlength, adressbredd- och databussens bredd

Man brukar ange en processors **ordlength** som det antal databitar ALU:n tar emot på ena dataingången eller lämnar på datautgången. Vanliga ordlength är idag (2004) 8, 16, 32 och 64 bitar.

**Databussens bredd** (antal bitar) bestämmer hur många bitar man kan hämta från minnet vid varje läsning eller lagra i minnet vid varje skrivning.

I de fall instruktioner eller dataord består av fler bitar än databussens bredd måste flera läsningar göras för varje instruktion och flera läsningar eller skrivningar göras för varje dataord.

Omvänt gäller att databussen kan vara bredare än processorns ordlength. I så fall kan processorn nå flera dataord vid varje läsning eller skrivning i minnet. Motsvarande gäller vid läsning av instruktioner i minnet. Det lästa minnesordet kan i detta fall innehålla två eller flera instruktioner.

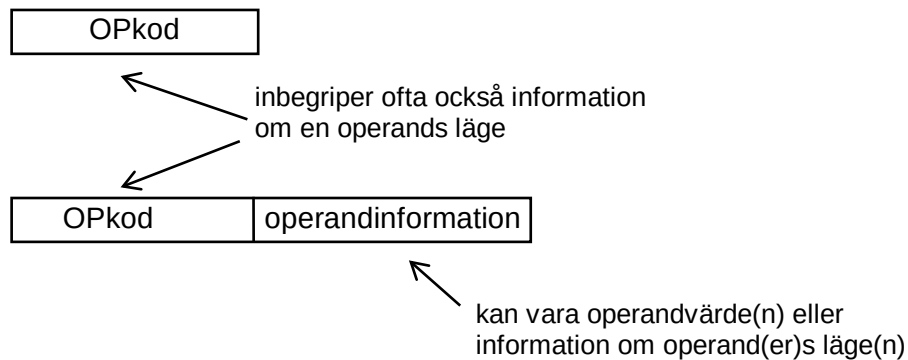
Vanliga bredder på databussen är idag 8, 16, 32 och 64 bitar.

**Adressbussens bredd** brukar överensstämma med programräknarens bredd (antal bitar) och bestämmer hur många olika adresser som kan användas av processorn. Mängden av samtliga adresser en processor kan adressera kallas **processorns adressrum**.

En adressbuss med 16 bitar ger t. ex.  $2^{16} = 65536$  olika adresskombinationer. Processorn kan då adressera  $2^{16} = 65536 = 64k$  olika minnesord. (Vi använder konventionen att  $1k = 2^{10} = 1024$ .) Vanliga bredder på adressbussen är idag 16, 24 och 32 bitar.

## F.5 Instruktionsformat

En instruktion innehåller den information (kodat med nollor och ettor) som behövs för att processorn skall kunna exekvera dess operation. En instruktion skall innehålla information om både operation och operand(er). Den del av instruktionen som specificerar operationen kallas operationskod, **OPkod**. Vanligen inkluderar denna kod också viss information om en operand. När så behövs ges ytterligare **operandinformation i en tilläggsdel**. Operations- och operandinformationen följer ett s k **instruktionsformat**, se figur F.6.



Figur F.6 Instruktionernas längd och innehåll följer ett sk instruktionsformat.

Sättet att koda operations- och operandinformation varierar starkt mellan processorer från olika tillverkare och ibland även mellan processorer från samma tillverkare.

## F.6 Instruktionsuppsättning

En processors användbarhet bestäms av de instruktioner som den är konstruerad att exekvera. Mängden instruktioner kallas för processorns **instruktionsuppsättning** (eng. **instruction set**). Varje processortillverkare ger en detaljerad beskrivning av instruktioner och de operationer de utför i en **instruktionslista** (eng. **instruction manual**). Trots att det finns stora skillnader mellan processorer från olika tillverkare så finns det också stora likheter dem emellan vad operationer och instruktionstyper beträffar.

Varje instruktion kännetecknas av

1. **en OPkod** = ett binärt ord som i instruktionslistan ges i hexadecimal form
2. **en längd** = ett antal bytes
3. **en exekveringstid** = ett antal klockcykler
4. **en operation med åtföljande flaggpåverkan** = det instruktionen utför
5. **en adresseringsmod**, som anger hur operandläget identifieras
6. **en mnemonisk beteckning** = en av tillverkaren rekommenderad symbolisk beteckning som har anknytning till operation och adresseringsmod

Instruktionerna grupperas efter operation i kategorier. Engelska namn anges inom parentes.

- |                                             |                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. <b>Dataöverföring</b><br>(Data transfer) | Flyttar data mellan processor och minne eller mellan interna register i processorn<br><br>Flyttning (kopiering) av data från minnet till ett register i processorn kallas att <b>ladda</b> från minnet. (eng <b>LOAD</b> ).<br><br>Flyttning (kopiering) av data från ett register i processorn till minnet kallas att <b>lagra</b> i minnet (eng. <b>STORE</b> ). |
| 2. <b>Aritmetik</b><br>(Arithmetic)         | Utför aritmetik, normalt med 2-komplementrepresentation<br>Byter tecken på tal. Ökar eller minskar tal med 1.                                                                                                                                                                                                                                                      |
| 3. <b>Logik</b><br>(Logic)                  | Utför logikoperationer som bitvis AND, OR eller XOR                                                                                                                                                                                                                                                                                                                |
| 4. <b>Test</b><br>(Test)                    | Testar och jämför dataord.                                                                                                                                                                                                                                                                                                                                         |
| 5. <b>Hopp</b><br>(Jump, Branch)            | Utför ovillkorliga och villkorliga hopp i program.                                                                                                                                                                                                                                                                                                                 |
| 6. <b>Andra</b>                             | Utför speciella processorberoende funktioner.                                                                                                                                                                                                                                                                                                                      |

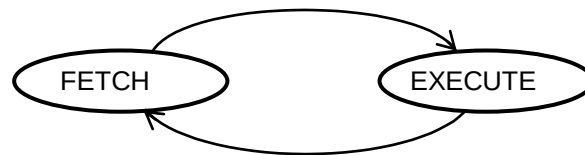
## F.7 Instruktioners exekvering

En instruktion utförs i två faser.

I den första fasan, som kallas **hämtfasen** (eng. **FETCH**), hämtas (läses) operationskoden för instruktionen i minnet och lagras i instruktionsregistret I.

Hämtfasen börjar med att innehållet i programräknaren skickas ut på adressbussen och en läsning görs i minnet. Det aktuella minnesordet (operationskoden) läggs då ut på databussen av minnet, läses av processorn och placeras (laddas) i instruktionsregistret I. Under tiden ökas innehållet i programräknaren med ett eftersom man normalt kommer att använda nästa adress i minnet nästa gång programräknaren används. Därmed är hämtfasen (FETCH) slut och processorn övergår till utförandefasen.

I den andra fasan, som kallas **utförandefasen** (eng. **EXECUTE**), känner styrenheten av innehållet i instruktionsregistret I, dvs operationskoden, och alstrar en styrsignalsekvens som beror på vilken instruktion som skall utföras. När utförandefasen är slut startas nästa instruktions hämtfas enligt tillståndsgrafen i figur F.7 nedan.



Figur F.7 Tillståndsgraf för instruktionernas två faser.



## F.8 FLEX-processorn

För att illustrera hur en processor är uppbyggd och arbetar har en enkel von Neumann-processor byggts vid institutionen. Den kallas FLEX för att den är flexibel i den mening att den vid behov kan omkonfigureras på flera olika sätt. FLEX-processorn är en vidareutveckling av den utökade von Neumann-processorn i figur F.3.

Programräknaren, PC, har ersatts av en laddningsbar räknare med räknevillkoret **IncPC** för ökning med 1 (inkrementering).

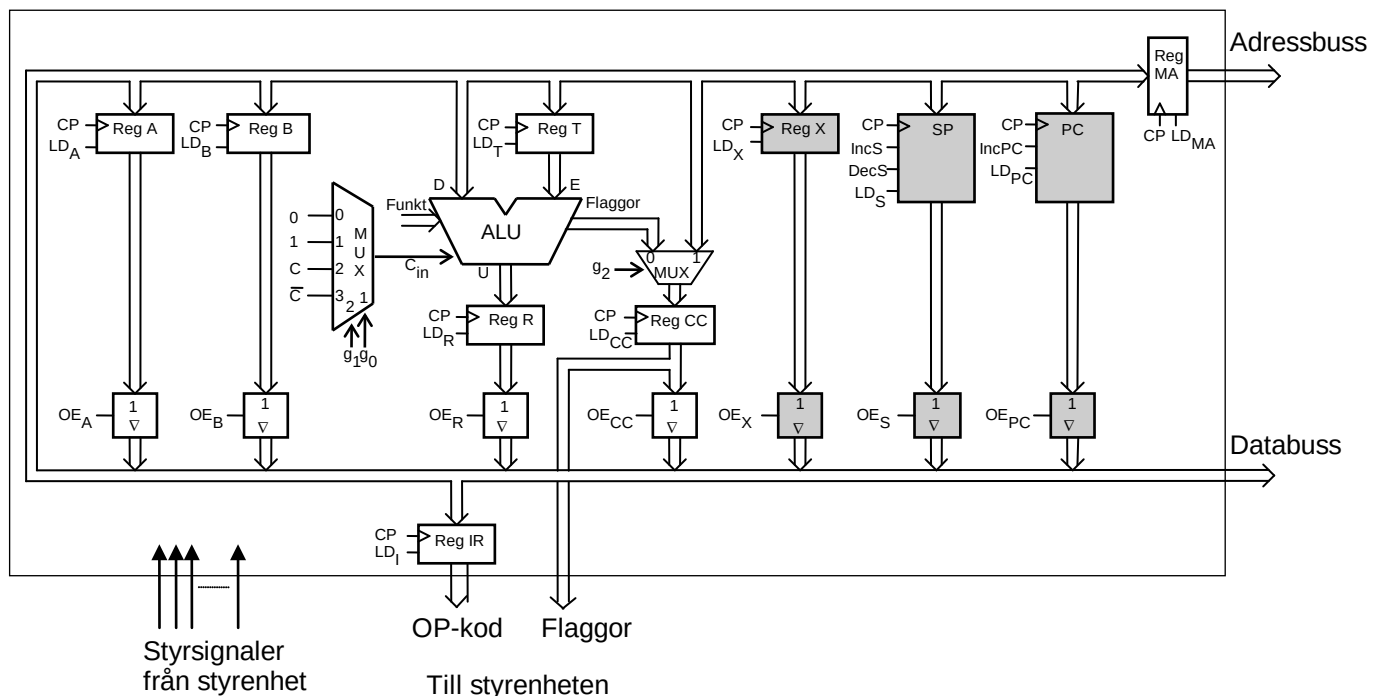
Det ena adressregistret har ersatts av en laddningsbar räknare med namnet **SP**. Denna räknare har räknevillkoret **IncSP** för ökning med 1 (inkrementering) och **DecSP** för minskning med 1 (dekrementering).

**SP** står här för **stackpekare**. Vad som menas med det förklaras senare.

Det andra adressregistret har fått namnet X.

**Datavägen** för FLEX-processorn ges i figur F.8.

FLEX arbetar genomgående med åttabitors ordlängd, vilket innebär att den har en åttabitors ALU, åttabitors register samt åttabitors adress- och databuss. Med åttabitors adressbuss begränsas primärminnets storlek till  $2^8 = 256$  adresser.



Figur F.8 FLEX-processorns dataväg.

Ett blockschema för **styrenheten** i FLEX-processorn ges i figur F.9. Styrenheten genererar de styrsignaler som behövs i datavägen. Den genererar också styrsignalerna för läsning och skrivning i primärminnet.

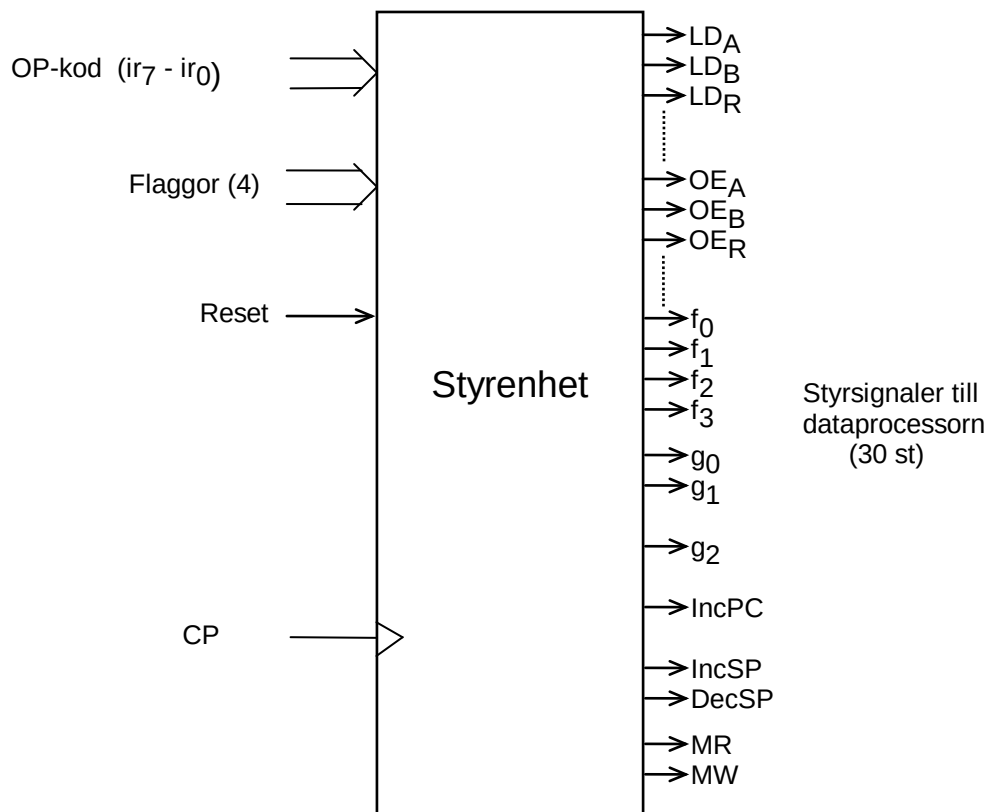
I kapitel 7 såg vi hur den databehandlade enheten kunde styras klockcykel för klockcykel genom att styrsignalerna först gavs önskade värden varpå en klockpuls CP (positiv flank) verkställer laddning av ett eller flera utvalda register. Klockpulssignalen CP bestämmer således arbetstakten.

Styrenheten utgöres av ett synkront sekvensnät med CP som klocksignal och processorns samtliga styrsignaler som utsignaler.

Styrenheten för FLEX-processorn skall kunna generera samtliga styrsignaler som behövs för att instruktionerna i dess instruktionslista skall utföras korrekt. Eftersom detta bl a medför läsningar och skrivningar i minnet måste även dessa styrsignaler genereras. Styrenheten måste alltså generera ett stort antal olika styrsignalsekvenser av varierande längd och komplexitet.

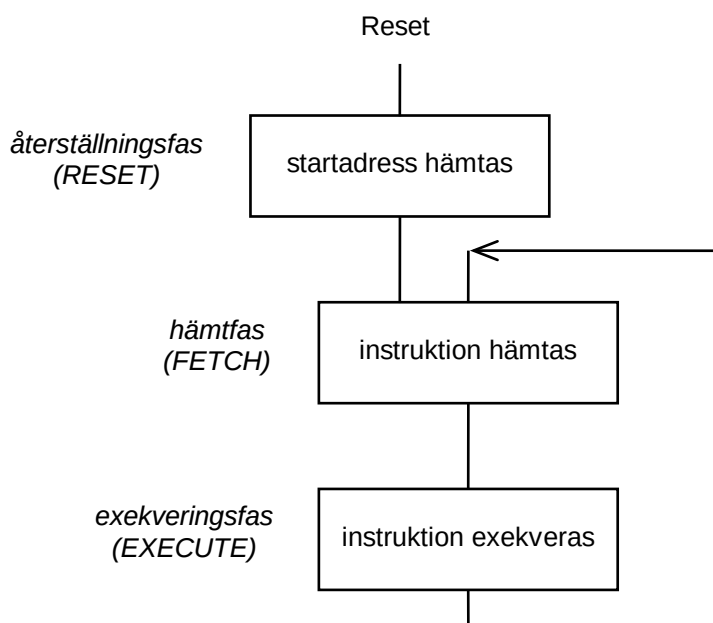
Som insignaler används dels innehållet i instruktionsregistret (OP-koden), dels flaggornas värden. OP-koden bestämmer utförandefasen och flaggornas värden kan användas för ett villkorligt vägval i exekveringen.

För att processorn skall kunna startas på ett väldefinierat sätt behövs även en startsignal Reset.



Figur F.9 FLEX-processorns styrenhet

Styrenhetens arbetssätt beskrivs bäst utifrån den tidigare omtalade instruktionscykeln. Den måste dock kompletteras med en fas för start/återstart, såsom visas i figur F.10. Den senare kallas vanligen för **återställningsfas** för att programräknaren **återställs** (eng **Reset**) till ett värde som definierar begynnelseadressen i det program som skall köras vid start/återstart.



Figur F.10 Instruktionscykeln kompletterad med s k återställningsfas.

Vi skall här inte gå in på hur styrenheten är uppbyggd men konstaterar att den är ett synkront sekvensnät och kan beskrivas med en tillståndsgraf såsom tidigare beskrivits i kapitel 5. Tillståndsgrafen visar hur växling sker mellan interna tillstånd, klockcykel för klockcykel, när instruktioner exekveras, dvs när instruktionscyklerna genomlöps.

Enligt figur F.10 sker uppstart när en startsignal (Reset) anländer till styrenheten. En återställningsfas (RESET) (ett antal tillstånd) kommer då att genomlöpas. Därefter övergår processorn till hämtfasen (FETCH). Under normalt arbete växlar den sedan ständigt mellan hämtfasen och exekveringsfasen (EXECUTE) enligt figur F.10.

För att ge en bild av hur processorn verkligen arbetar skall återställningsfasen och hämtfasen beskrivas.

### Återställningsfasen (RESET)

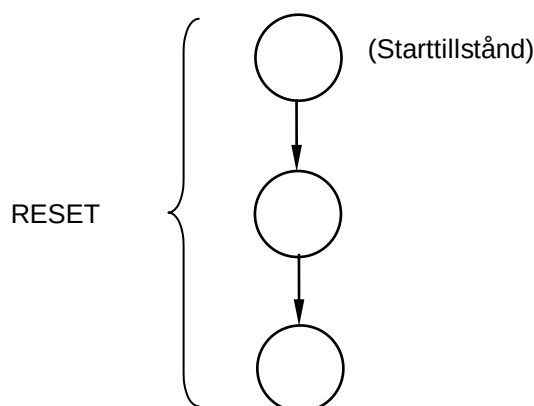
Återställning förbereds (initieras) genom att FLEX-processorns insignal Reset aktiveras. Den därpå följande klockpulsen startar återställningen som därefter sker under ett antal klockcykler. Processorn genomlöper en sekvens av tillstånd, den s k återställningssekvensen (RESET-sekvensen).

FLEX-processorns RESET-sekvens (vi kallar den ofta så) börjar med att processorn läser minnesinnehållet på den högsta adressen ( $FF_{16}$ ) i adressrummet. Det lästa minnesinnehållet är den adress (8 bitar) där processorn skall hämta den första instruktionen i programmet som skall köras. Det lästa dataordet, som alltså är startadressen för ett program, placeras därför i programräknaren (PC) och processorn kan övergå till hämtfasen. Det som sker i processorn under återställningsfasen beskrivs i tabell F.1.

Tabell F.1

| Klockcykel<br>(State nr) | RTN-<br>beskrivning     | Styr signaler                           | Kommentar                                                                                                                                                                                          |
|--------------------------|-------------------------|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0                        | $FF_{16} \rightarrow R$ | ALU-funktion = $F_{16}$ ,<br>$LD_R=1$ . | ALU-funktionen väljs så att talet $FF_{16}$ finns på ALU:ns utgång. Laddingången på R-registret ettställs så att utvärdet från ALU'n ( $FF_{16}$ ) laddas i R-registret vid nästa klockpuls.       |
| 1                        | $R \rightarrow MA$      | $OE_R=1$<br>$LD_{MA}=1$ .               | Talet $FF_{16}$ i R-registret kopplas ut på bussen. Talet $FF_{16}$ på bussen laddas i minnesadress-registret vid nästa klockpuls.                                                                 |
| 2                        | $M \rightarrow PC$      | $MR=1$ ,<br>$LD_{PC}=1$ .               | Minnesinnehållet på adressen $FF_{16}$ läses genom att minnet aktiveras för läsning. Det dataord som läses placeras i PC vid nästa klockpuls. Nästa klockcykel skall vara den första i fetchfasen. |

RESET-sekvensen kan alltså utföras med tre tillstånd enligt tillståndsgrafen i figur F.11. De tre ringarna representerar var sitt (unika) tillstånd, precis som i tillståndsgraferna som beskrevs i kapitel 5. Övergång mellan ringarna verkställs av klockpulser till styrenheten. I varje ring skall vissa styr signaler aktiveras (ettställas), vilka framgår av kolumnen styr signaler i tabell F.1.



Figur F.11

## Hämtfasen

I FLEX-processorn har alla instruktioner samma krav på hämtfasen, dvs arbetet som utförs under hämtfasen (FETCH) är detsamma för varje instruktion.

En instruktion utförs genom att OPkoden först hämtas från minnet med en läsoperation och placeras i instruktionsregistret IR under hämtfasen. Därefter inleds exekveringsfasen vars arbete styrs av den aktuella OPkoden i IR-registret.

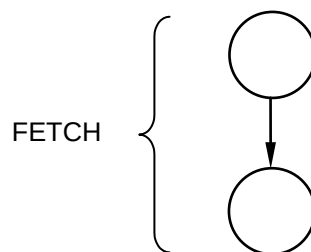
Under hämtfasen genomlöps en tillståndsssekvens som inleds med en klockcykel där innehållet i programräknaren PC laddas i minnesadressregistret MA för att användas som adress vid en läsning i minnet. I samma klockcykel ökas dessutom PC-värdet med ett.

Det lästa dataordet, som är en OPkod för en instruktion, laddas av nästa klockpuls i instruktionsregistret IR och exekveringsfasen kan därefter inledas i den därpå följande klockcykeln. Skeendet under hämtfasen beskrivs i tabell F.2.

Tabell F.2

| Klockcykel<br>(State nr) | RTN-<br>beskrivning   | Styr signaler                                             | Kommentar                                                                                                                                       |
|--------------------------|-----------------------|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 0                        | PC→MA,<br><br>PC+1→PC | OE <sub>PC</sub> =1, LD <sub>MA</sub> =1,<br><br>IncPC=1. | Adressen för nästa instruktions<br>operationskod kopieras från PC till<br>minnesadressregistret MA.<br>Adressen som finns i PC ökas med<br>ett. |
| 1                        | M→IR                  | MR=1,<br>LD <sub>I</sub> =1.                              | Läs operationskoden från minnet.<br>Placera den i instruktionsregistret IR.<br>Nästa klockcykel skall vara den första<br>i executefasen.        |

Hämtfasen kan alltså utföras med två tillstånd enligt figur F.12. När styrenheten "befinner sig" i den nedre av de två ringarna kommer den att övergå från FETCH till EXECUTE då nästa klockpuls kommer.

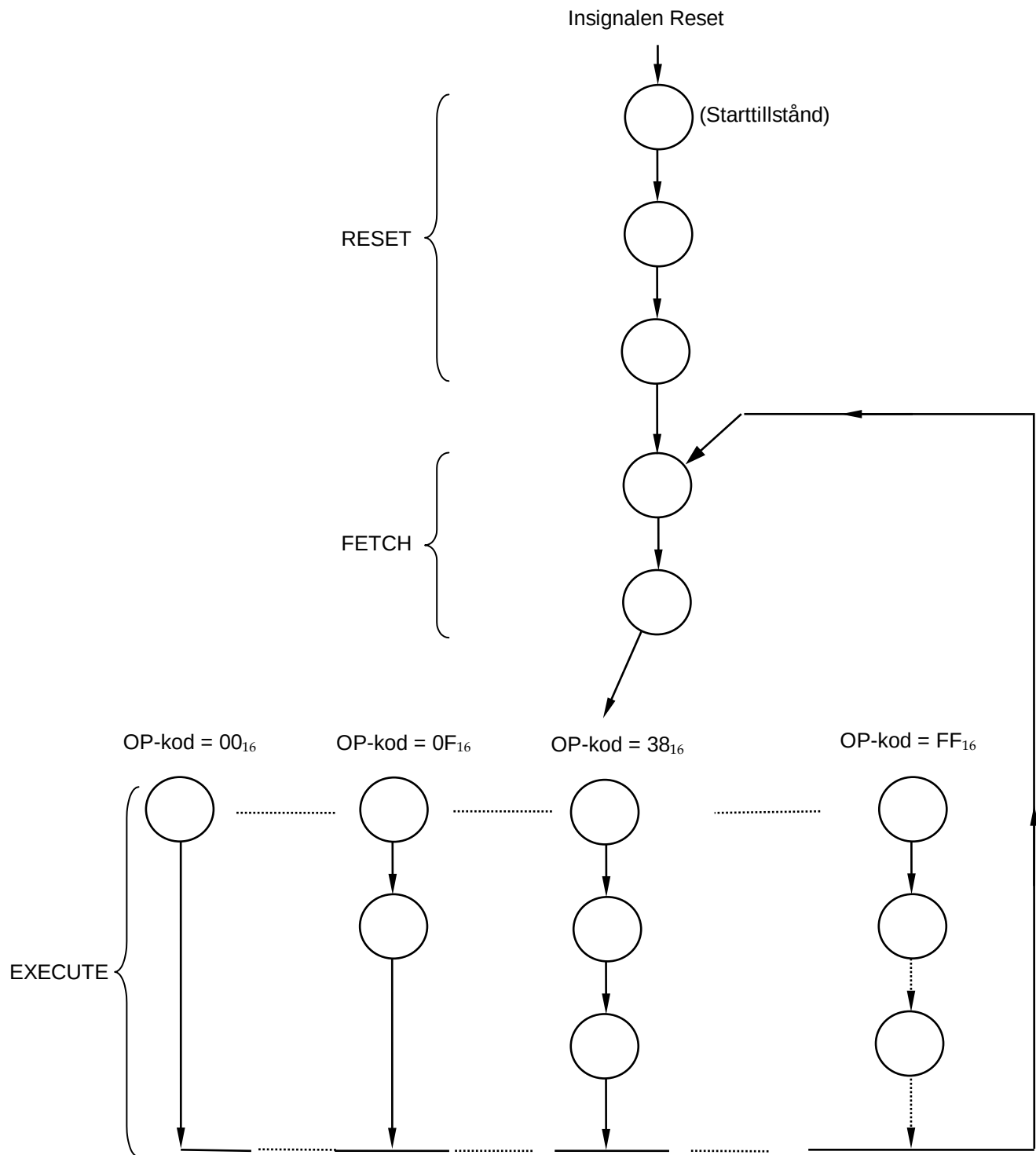


Figur F.12

## Tillståndsgraf för styrenheten

Under EXECUTE bestäms styrenhetens beteende av OP-koden i instruktionsregistret IR och eventuellt av flaggornas värden. FLEX-processorns OP-kod består av åtta bitar och kan därför ha  $2^8 = 256$  olika värden. Den representerar alltså 256 möjliga EXECUTE-faser.

Övergången från FETCH-fasen till någon av EXECUTE-faserna illustreras i tillståndsgrafen i figur F.13. Där visas RESET-sekvensen, FETCH-sekvensen och EXECUTE-sekvenserna för OP-koderna 00<sub>16</sub> t o m FF<sub>16</sub>.



Figur F.13 Tillståndsgraf för styrenheten till FLEX-processorn.

I tillståndsgrafen skall egentligen också finnas övergångar (pilar) från varje ring till den översta ringen, dvs starttillståndet. Denna typ av övergång skall ske när styrenhetens insignal Reset = 1.

Processorns tillståndsgraf kommer att studeras mer i detalj senare.

## F.9 Instruktionsuppsättning för FLIS-processorn

**Observera att vi här använder FLIS-processorn istället för FLEX-processorn!**

En processors instruktionsuppsättning är sammansatt av instruktioner som behövs för 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 som kan uttrycka konstruktionerna **sekvens**, **selektion** och **iteration** vilka behövs i algoritmer.

Instruktionsuppsättningen är ofta ganska primitiv. I enkla processorer saknas ex vis ofta instruktioner för operationer som multiplikation och division. Lite mer komplexa operationer implementeras då istället som följder av processorns primitiva operationer. Alla **operationer** som erbjuds framgår av instruktionsuppsättningen som visas i processorns instruktionslista. Merparten av instruktionerna tillhör någon av följande kategorier

- \* flyttning av data
- \* bearbetning av data, aritmetik och logik
- \* tester
- \* hopp

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

Alla kategorierna är på något sätt beroende av adresser. De tre första kategorierna hanterar operander och måste på något sätt specificera lägen för 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 hopp-adress kan specificeras via instruktioner. Specificeringen kan ske på ett flertal olika sätt. Processorn sägs ha olika **adresseringsmoder** (eng **addressing modes**). Dessa framgår också av instruktionslistan. I texten definieras och diskuteras adresseringsmoderna när det är lämpligt.

I detta avsnitt beskrivs både operationer och adresseringsmoder som är typiska för en enkel processors instruktionsuppsättning, samt hur de kan användas. Operationerna beskrivs med RTN och namnges med gängse engelska benämningar. Detta görs huvudsakligen utifrån ovanstående kategoriindelning.

De instruktioner (maskininstruktionerna) som processorn hämtar från minnet (primärminnet) består av enbart nollor och ettor. För att vi människor lättare skall kunna läsa och förstå maskinprogram har man infört sk **assemblerspråk** (eng **assembly language**), där varje instruktion representeras av en bokstavs-förkortning av dess funktion, en sk **mnemonisk beteckning**.

Processortillverkaren Motorola (numera Freescale) har definierat assemblerspråk för sina kommersiella mikroprocessorer, som kommer att användas i en senare kurs. Vi har därför använt samma skrivsätt för FLIS-processorns instruktioner.

Då tal skall anges i assemblerinstruktionerna kan man välja att använda det hexadecimala, binära eller decimala talsystemet. Valet av talsystem framgår av ett prefix till talsiffrorna enligt uppställningen nedan:

**\$**hexadecimala talsiffror = hexadecimalt  
**%**binära talsiffror = binärt  
 decimala talsiffror = decimalt (inget prefix)

### Instruktioner för flyttning av data

Såvida inget annat anges är flyttning av data detsamma som kopiering av data. I instruktionsuppsättningen finns instruktioner för att flytta data **mellan interna register**

$r_{src} \rightarrow r_{dst}$  transfer  
 $r_1 \leftrightarrow r_2$  exchange (inbördes byte)

och **mellan interna register och register i minnet (M, primärminnet)**

$r_{src} \rightarrow M$  store  
 $M \rightarrow r_{dst}$  load

Med en "transfer"-instruktion flyttas ett ord ifrån ett av processorns register till ett annat. Det register varifrån data hämtas kallas allmänt för **källregister**,  $r_{src}$  (eng. **source register**) och det register data flyttas till kallas **destinationsregister**,  $r_{dst}$  (eng. **destination register**). De register som instruktionen skall använda måste specificeras i maskininstruktionen.

### Exempel F.1

Av instruktionslistan framgår att FLISP har flera "transfer"- och "exchange"-instruktioner.

Instruktionen TFR X,Y kopierar värdet i register X och placerar det i register Y. Instruktionen utgörs av en byte, som är Opkoden.

|                   |                   |
|-------------------|-------------------|
| maskininstruktion | operation         |
| 1A (OPkod)        | $X \rightarrow Y$ |

Med "store"- och "load"-instruktionerna lagras resp. hämtas ett operandvärde på en adress i minnet. Dessa instruktioner finns i olika varianter beroende på hur operandadressen specificeras.

En variant är att explicit (direkt) ge operandadressen i instruktionen. I maskinspråket är i dessa fall den verkliga (absoluta) adressen till destinationen resp. källan placerad i instruktionen, efter OPkoden. Genom en sådan instruktion tvingas programmet att alltid använda den givna adressen. Man kan säga att programmet använder sig av ett variabelvärde som finns på en bestämd adress. Adresseringsmoden kallas ibland **absolut** (eng. **absolute**) och ibland **direkt** (eng. **direct**).



**Exempel F.2**

En FLISP-instruktion med absolut (absolute) adressering är STA \$AB som i maskinspråket har utseendet

|                    |                            |
|--------------------|----------------------------|
| maskininstruktion  | operation                  |
| E1 (OPkod)         |                            |
| AB (operandadress) | $A \rightarrow M(AB_{16})$ |

Här är det OPkoden  $E1_{16}$  som anger att A-registret är källregister och att den följande byten,  $AB_{16}$ , utgör adressen till operanden. Det interna registret som instruktionen använder sig av specificeras således i OPkoden.

\$-tecknet används för att ange att adressen är på hexadecimal form.

I en variant av "load"-instruktionen kan operandvärdet ingå i instruktionen. Det adresseras då av programräknaren och hämtas omedelbart in till processorn i samband med att instruktionen hämtas. Adresseringsmoden kallas **omedelbar** (eng **immediate**). Denna instruktion används när operandvärdet är en konstant. Konstanter används ju ofta vid beräkning av uttryck.

**Exempel F.3**

Hos FLISP anger #-tecknet omedelbar (immediate) adressering, dvs att operanden följer omedelbart i instruktionen. Instruktionen LDA #\$35 har i maskinspråket utseendet

|                   |                         |
|-------------------|-------------------------|
| maskininstruktion | operation               |
| F0 (OPkod)        |                         |
| 35 (operand)      | $35_{16} \rightarrow A$ |

Här är det OPkoden  $F0_{16}$  som anger att A-registret är destinationsregister och att den efterföljande byten  $35_{16}$  är en operand.

För FLISP finns det ingen "store"-instruktion med omedelbar adressering.

En processor har vanligen också "store"- och "load"-instruktioner med andra adresseringsmoder, dvs där operand specificeras på andra sätt. Dessa beskrivs senare.

### Bearbetning av data, aritmetik och logik

I detta delavsnitt beskrivs de instruktioner som utför aritmetik- resp. logikoperationer. De delas in efter om operationen är unär eller binär. Tillsammans med instruktioner för dataflyttning används de i hög grad för beräkning av uttryck. De kan använda sig av operander i form av konstanter eller variabelvärden. Vid användning av konstanter kan adresseringen vara **omedelbar** (eng. **immediate**). Vid användning av variabelvärden är adressering ordnad enligt någon annan adresseringsmod.

#### Unära operationer

Unära operationer arbetar endast på en operand. Operationen består i en bearbetning (manipulation) av operanden. Hos varje processor finns ett antal operationer av denna natur. Läget för operanden kan antingen vara ackumulator A eller en adress i minnet. Läget indikeras tillsammans med den mnemoniska beteckningen, här generellt betecknad MNE, på följande sätt

|                      |            |      |             |
|----------------------|------------|------|-------------|
| $f(A) \rightarrow r$ | operation, | MNEA |             |
| $f(M) \rightarrow M$ | "          | MNE  | operandinfo |

där A är beteckningen på ackumulator A och M är beteckningen på en adress i minnet.

I det första fallet, där operationen arbetar på innehållet i ett register, specificeras registret **inne i** instruktionens OPkod. På engelska sägs då specificeringen vara "inherent" i OPkoden och följaktligen kallas denna adresseringsmod "**inherent**".

I det senare fallet kan adresseringsmoden vara omedelbar, direkt (absolut) eller någon av de minnesrelaterande moder som beskrivs senare i kapitlet. Adressen bildas med ledning av den operandinfo som följer med instruktionen. Man kan i dessa fall få intrycket att processorn kan bearbeta ord utanför processorn, direkt i minnet. Detta är förstås ej möjligt utan operanden hämtas in till processorn, bearbetas och återplaceras i minnet med en och samma instruktion. Här följer en beskrivning av de vanligaste unära operationerna.

#### Inkrementering och dekrementering av operand

|                     |            |      |             |
|---------------------|------------|------|-------------|
| $A+1 \rightarrow A$ | increment, | INCA |             |
| $M+1 \rightarrow M$ |            | INC  | operandinfo |
| $A-1 \rightarrow A$ | decrement  | DECA |             |
| $M-1 \rightarrow M$ |            | DEC  | operandinfo |

Ofta används innehållet i ackumulator A eller på en adress som ett "räknarvärde", som skall användas på något sätt under ett programs gång. Ett sådant värde kan med dessa instruktioner ökas med 1, inkrementeras, eller minskas med 1, dekrementeras, när så önskas.

**Exempel F.4**

|                                                                    |                   |           |
|--------------------------------------------------------------------|-------------------|-----------|
| I FLISP inkrementeras innehållet i ackumulator A med instruktionen |                   |           |
|                                                                    | maskininstruktion | operation |
| INCA                                                               | 07 (OPkod)        | A+1 → A   |

**Exempel F.5**

|                                                                             |                     |                                               |
|-----------------------------------------------------------------------------|---------------------|-----------------------------------------------|
| Innehållet på adress 12 <sub>16</sub> inkrementeras med FLISP-instruktionen |                     |                                               |
|                                                                             | maskininstruktion   | operation                                     |
| INC \$12                                                                    | 37 (OPkod)          |                                               |
|                                                                             | 12 (operand adress) | M(12 <sub>16</sub> )+1 → M(12 <sub>16</sub> ) |

*Nollställning av operand*

|       |        |                 |
|-------|--------|-----------------|
| 0 → A | clear, | CLRA            |
| 0 → M |        | CLR operandinfo |

*Invertering av operands bitar*

|                         |             |                 |
|-------------------------|-------------|-----------------|
| $\bar{A} \rightarrow A$ | complement, | COMA            |
| $\bar{M} \rightarrow M$ |             | COM operandinfo |

Den senare operationen används dels som en ren logikoperation för att bitvis invertera (komplementera) en operands bitar, dels tillsammans med "clear"-operationen för att ettställa alla bitarna i en operand.

**Exempel F.6**

|                                                                                                                |                   |                         |
|----------------------------------------------------------------------------------------------------------------|-------------------|-------------------------|
| Ackumulator A:s bitar inverteras med FLISP-instruktionen                                                       |                   |                         |
|                                                                                                                | maskininstruktion | operation               |
| COMA                                                                                                           | 0A (OPkod)        | $\bar{A} \rightarrow A$ |
| vilket innebär att om innehållet i A är 01000011 så ändrar instruktionen ackumulatorns innehåll till 10111100. |                   |                         |

**Exempel F.7**

|                                                                                  |      |  |
|----------------------------------------------------------------------------------|------|--|
| Hos FLISP kan alla bitarna i A-ackumulatorn ettställas med instruktionssekvensen |      |  |
|                                                                                  | CLRA |  |
|                                                                                  | COMA |  |
|                                                                                  | .    |  |

Negering (tvåkomplementering) av en operand

$$0 - A = 0 + \bar{A} + 1 \rightarrow A \quad \text{negate,} \quad \text{NEGA}$$

$$0 - M = 0 + \bar{M} + 1 \quad \text{NEG operandinfo}$$

Operationen används i aritmetik med tal på tvåkomplementform för att byta tecken på en operand.

### Exempel F.8

Talet på adress  $45_{16}$  tvåkomplementeras med instruktionen

|          | maskininstruktion                | operation                               |
|----------|----------------------------------|-----------------------------------------|
| NEG \$45 | 36 (OPkod)<br>45 (operandadress) | $0 - M(45_{16}) \rightarrow M(45_{16})$ |

Om  $M(45_{16}) = 01110100$  så ändrar instruktionen detta till  $10001100$ .

### Exempel F.9

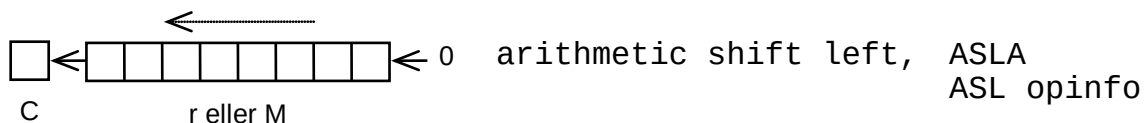
Konstanten  $(-32)_{10}$  på 2-komplementrepresentation placeras på adress  $D1_{16}$  med instruktionssekvensen

```

      .
      LDA  #32
      NEGA
      STA  $D1
      .

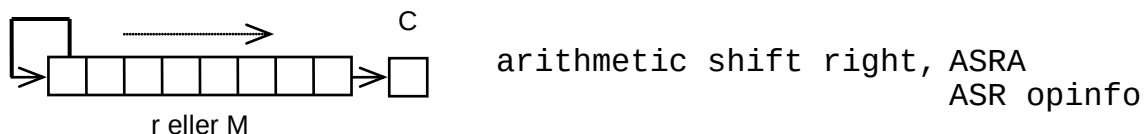
```

Aritmetiskt skift åt vänster



Denna instruktion skiftar operandens bitar ett steg åt vänster på det sätt som visas i operationsfiguren. Den används i aritmetik med tal på tvåkomplementform för att multiplicera operanden med 2. Detta illustreras i exemplet på nästa sida.

Aritmetiskt skift åt höger



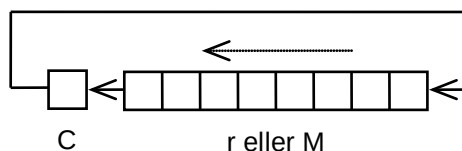
Denna instruktion skiftar operandens bitar ett steg åt höger på det sätt som visas i operationsfiguren. Den används i aritmetik med tal på tvåkomplementform för att dividera operanden med 2.

**Exempel F.10**

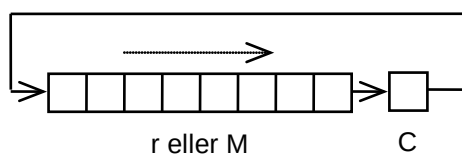
Utgå ifrån talet  $W = (+25)_{10} = (00011001)_2$  och iakttag hur  $W$  och  $-W$  ändras när de multipliceras med 2. Binärpunkten antas vara placerad till höger om bit 0.

|              | dec | bin                                   |
|--------------|-----|---------------------------------------|
| positivt tal |     |                                       |
| $W =$        | 25  | 00011001                              |
| $2W =$       | 50  | 000110010                             |
|              |     | hamnar i C-flaggan ← skiftas in i LSB |
| negativt tal |     |                                       |
| $-W =$       | -25 | 11100111                              |
| $-2W =$      | -50 | 111001110                             |
|              |     | hamnar i C-flaggan ← skiftas in i LSB |

Man löper alltid en risk att tappa signifikanta siffror vid dubbling.

*Skift vid rotation*

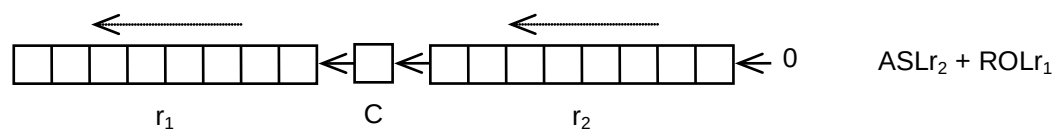
rotate left, ROLA  
ROL opinfo



rotate right, RORA  
ROR opinfo

Vid rotationsskift åt vänster skiftas operandens bitar ett steg åt vänster. 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. Genom 9 likadana rotationsinstruktioner kan således bitmönstret i C-flaggan + operanden roteras (cirkuleras) ett varv.

Rotationsinstruktionerna kan med fördel användas tillsammans med andra skiftinstruktioner för att ta hand om de bitar som dessa skiftar ut ur en operand. Man kan på så vis göra en aritmetisk operandutvidgning från 8 till 16 bitar enligt den idé som ges i figur F.14. (Minst ett register antas finnas i minnet.) De kan också användas för att separera en operand i flera delar och placera dem på olika ställen.



Figur F.14 Operandutvidgning åt vänster.

*Binära operationer*

Eftersom binära operationer arbetar på två operander borde tre lägen (två källor och en destination) vara förknippade med dessa. Så är emellertid ej fallet hos de vanliga processorerna, ty destinationen för resultatet är vanligen densamma som källan för en av operanderna. Hos FLISP är detta läge normalt ackumulatort, A, men kan också vara flaggregistret. De binära operationerna betecknas med RTN och mnemonik på följande sätt

$r * M$  operation, MNEr operandinfo

där som förut r är beteckningen på det interna registret och M är beteckningen på adressen i minnet. Den andra operanden finns således alltid i processorns minne. Dess adresseringsmod kan vara omedelbar, direkt (absolut) eller någon av de som beskrivs senare.

*Aritmetikinstruktioner*

$A + M \rightarrow A$  add, ADDA operandinfo

Dessa additionsinstruktioner utför additionen  $P = D + E$  på följande sätt

$$\begin{array}{r} \text{C}_8 \text{C}_7 \text{C}_6 \text{C}_5 \text{C}_4 \text{C}_3 \text{C}_2 \text{C}_1 \text{C}_0 \\ \text{d}_7 \text{d}_6 \text{d}_5 \text{d}_4 \text{d}_3 \text{d}_2 \text{d}_1 \text{d}_0 \\ + \text{e}_7 \text{e}_6 \text{e}_5 \text{e}_4 \text{e}_3 \text{e}_2 \text{e}_1 \text{e}_0 \\ \hline \text{carry} = \text{c}_8 \quad \text{p}_7 \text{p}_6 \text{p}_5 \text{p}_4 \text{p}_3 \text{p}_2 \text{p}_1 \text{p}_0 \end{array}$$

vilket lämpar sig väl för addition av två 8 bitars tal med eller utan tecken. För tal med tecken skall 2-komplementrepresentation användas.

$A - M \rightarrow A$  subtract, SUBA operandinfo

Det är känt att subtraktion utförs genom addition. Subtraktionsinstruktioner utför subtraktionen  $P = D - E$  på följande sätt

$$\begin{array}{r} \text{C}_8 \text{C}_7 \text{C}_6 \text{C}_5 \text{C}_4 \text{C}_3 \text{C}_2 \text{C}_1 \text{1} \\ \text{d}_7 \text{d}_6 \text{d}_5 \text{d}_4 \text{d}_3 \text{d}_2 \text{d}_1 \text{d}_0 \text{svarar mot} \\ + \text{e}_7 \text{'e}_6 \text{'e}_5 \text{'e}_4 \text{'e}_3 \text{'e}_2 \text{'e}_1 \text{'e}_0 \text{' } \\ \hline \text{p}_7 \text{p}_6 \text{p}_5 \text{p}_4 \text{p}_3 \text{p}_2 \text{p}_1 \text{p}_0 \end{array} \quad \begin{array}{r} \text{b}_8 \text{b}_7 \text{b}_6 \text{b}_5 \text{b}_4 \text{b}_3 \text{b}_2 \text{b}_1 \text{b}_0 \\ \text{d}_7 \text{d}_6 \text{d}_5 \text{d}_4 \text{d}_3 \text{d}_2 \text{d}_1 \text{d}_0 \\ - \text{e}_7 \text{e}_6 \text{e}_5 \text{e}_4 \text{e}_3 \text{e}_2 \text{e}_1 \text{e}_0 \\ \hline \text{p}_7 \text{p}_6 \text{p}_5 \text{p}_4 \text{p}_3 \text{p}_2 \text{p}_1 \text{p}_0 \end{array}$$

Det skall vid subtraktioner observeras att det är **lånesiffran** (eng. **borrow**)  $b_8$  som placeras i C-flaggan. Mellan  $b_8$  och  $c_8$  i uppställningen gäller att

$$b_8 = c_8'$$

Tänk igenom varför!

**Exempel F.11**

|                                                                                                 |      |                    |                                |
|-------------------------------------------------------------------------------------------------|------|--------------------|--------------------------------|
| Med FLISP adderas talet på adress 20 <sub>16</sub> till talet i ackumulator A med instruktionen |      |                    |                                |
|                                                                                                 |      | maskininstruktion  | operation                      |
| ADDA                                                                                            | \$20 | A6 (OPkod)         |                                |
|                                                                                                 |      | 20 (operandadress) | $A + M(20_{16}) \rightarrow A$ |

**Exempel F.12**

|                                                                                                  |      |                    |                                |
|--------------------------------------------------------------------------------------------------|------|--------------------|--------------------------------|
| Talet på adress 24 <sub>16</sub> subtraheras ifrån talet i ackumulator A med FLISP-instruktionen |      |                    |                                |
|                                                                                                  |      | maskininstruktion  | operation                      |
| SUBA                                                                                             | \$24 | A4 (OPkod)         |                                |
|                                                                                                  |      | 24 (operandadress) | $A - M(24_{16}) \rightarrow A$ |

I många fall måste operander uttryckas med fler än 8 bitar. Vid användning av 8 bitars processorer uttrycks de då med 16, 24 eller någon annan multipel av 8 bitar. Man säger att operanden har **dubbelprecision**, **trippelprecision** eller **multipelprecision**. Vid aritmetik för sådana operander måste då bearbetningen delas upp i en följd av 8 bitars operationer. För att klara av detta har processorn de speciella additions- och subtraktionsinstruktionerna

$A+M+C \rightarrow A$       add with carry,      ADCA    operandinfo

$A-M-C \rightarrow A$       subtract with borrow,      SBCA    operandinfo

Dessa används på följande sätt:

Additionen,  $P = D + E$ , där D och E är 16 bitars ord med eller utan tecken, delas då upp i två delar enligt

$$\begin{array}{r}
 \begin{array}{cccccccc}
 C_{16} & C_{15} & C_{14} & C_{13} & C_{12} & C_{11} & C_{10} & C_9 & C_8 \\
 & d_{15} & d_{14} & d_{13} & d_{12} & d_{11} & d_{10} & d_9 & d_8 \\
 + & e_{15} & e_{14} & e_{13} & e_{12} & e_{11} & e_{10} & e_9 & e_8 \\
 \hline
 & p_{15} & p_{14} & p_{13} & p_{12} & p_{11} & p_{10} & p_9 & p_8
 \end{array}
 &
 \begin{array}{r}
 \begin{array}{cccccccc}
 C_8 & C_7 & C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 \\
 & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\
 + & e_7 & e_6 & e_5 & e_4 & e_3 & e_2 & e_1 & e_0 \\
 \hline
 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
 \end{array}
 \end{array}
 \end{array}$$

I denna adderas först operandernas minst signifikanta bytes,

$$PL = DL + EL$$

och därefter deras mest signifikanta bytes tillsammans med den minnessiffran,  $c_8$ , som den första additionen producerat

$$PH = DH + EH + c_8$$

Denna senare addition kan således utföras med en "add with carry"-instruktion, eftersom  $c_8$  efter den första additionen hamnar i C-flaggan.

På liknande sätt delas subtraktionen,  $P = D - E$ , upp i två delar. Även om det är känt att den utförs som addition av tvåkomplementet av  $E$ , så illustreras här den subtraktion som additionen utför

$$\begin{array}{r} b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8 \\ d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8 \\ - e_{15}e_{14}e_{13}e_{12}e_{11}e_{10}e_9e_8 \\ \hline p_{15}p_{14}p_{13}p_{12}p_{11}p_{10}p_9p_8 \end{array} \qquad \begin{array}{r} b_7b_6b_5b_4b_3b_2b_1b_0 \\ d_7d_6d_5d_4d_3d_2d_1d_0 \\ - e_7e_6e_5e_4e_3e_2e_1e_0 \\ \hline p_7p_6p_5p_4p_3p_2p_1p_0 \end{array}$$

där operandernas minst signifikanta bytes behandlas först

$$PL = DL - EL$$

och deras mest signifikanta bytes därefter

$$PH = DH - EH - b_8$$

Av detta inses att den senare subtraktionen kan utföras med en "subtract with borrow"-instruktion, eftersom lånesiffran efter den första subtraktionen  $b_8$  finns i C-flaggan.

### Exempel F.13

Med FLISP kan 16-bitars talet på adresserna  $11_{16}$  (MSB) och  $12_{16}$  (LSB) subtraheras från 16-bitars talet på adresserna  $13_{16}$  (MSB) och  $14_{16}$  (LSB) med instruktionssekvensen

|      |      |                                |
|------|------|--------------------------------|
| LDA  | \$14 | Hämta låg byte                 |
| SUBA | \$12 | Här hamnar lånesiffran i C     |
| STA  | \$14 | Spara låg byte. C påverkas ej  |
| LDA  | \$13 | Hämta hög byte. - " -          |
| SBCA | \$11 | Här finns lånesiffran kvar i C |
| STA  | \$13 | Spara hög byte                 |

### Logikinstruktioner

Processorn har instruktioner för att bitvis bilda AND, OR och EXCLUSIVE-OR mellan operandernas bitar. Hur detta görs skall beskrivs i följande uppställning

$$\begin{array}{rcl} & d_7d_6d_5d_4d_3d_2d_1d_0 & \text{operand 1} \\ \text{operation} & *e_7e_6e_5e_4e_3e_2e_1e_0 & \text{operand 2} \\ & \hline & p_7p_6p_5p_4p_3p_2p_1p_0 & \text{producerat resultat} \end{array}$$

där  $*$  svarar mot operationen. 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

$$p_i = d_i * e_i$$

Logikinstruktionerna spelar en stor roll i mikrodatortekniken, ty man kan med dem förändra enstaka bitar i en operand. Försättningsvis skall logikinstruktionernas huvudsakliga användning beskrivas.

$$A \wedge M \rightarrow A \quad \text{and,} \quad \text{ANDA} \quad \text{operandinfo}$$



Med OCH-operationen kan man selektivt nollställa enstaka bitar i en operand såsom framgår av uppställningen

$$\begin{array}{r}
 d_7 \ d_6 \ d_5 \ d_4 \ d_3 \ d_2 \ d_1 \ d_0 \quad \text{operand} \\
 \wedge \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \quad \text{mask = nollställande bitmönster} \\
 \hline
 0 \ 0 \ d_5 \ d_4 \ d_3 \ d_2 \ 0 \ 0 \quad \text{resultat}
 \end{array}$$

Man säger att operationen maskerar de bitar som nollställs och frigör de övriga för fortsatt användning.

$$A \vee M \rightarrow A \quad \text{or,} \quad \text{ORA operandinfo}$$

På samma sätt kan man med ELLER-operationen selektivt ettställa enstaka bitar i en operand och på så sätt maskera deras gamla värden

$$\begin{array}{r}
 d_7 \ d_6 \ d_5 \ d_4 \ d_3 \ d_2 \ d_1 \ d_0 \quad \text{operand} \\
 \vee \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \quad \text{mask = ettställande bitmönster} \\
 \hline
 d_7 \ d_6 \ d_5 \ d_4 \ 1 \ 1 \ 1 \ 1 \quad \text{resultat}
 \end{array}$$

#### Exempel F.14

Maskering med AND- och OR-instruktioner illustreras med en FLISP-instruktionssekvens, som undersöker bit 7 (teckenbiten) på adress 31<sub>16</sub>. Om den är 1 så ettställs bit 4 på adress 32<sub>16</sub> och om den är noll så nollställs bit 4 på adress 32<sub>16</sub>.

| Adr |      |            |                                         |
|-----|------|------------|-----------------------------------------|
| 11  | LDA  | \$31       | Hämta innehållet på adress \$31         |
| 13  | ANDA | ##10000000 | Här maskas bit7 fram i ackumulator A    |
| 15  | BMI  | \$1D       | Om bit7=1 fortsätter man på adress \$1D |
| 17  | LDA  | ##11101111 | Man ser här att bit4 = 0, övriga = 1    |
| 19  | ANDA | \$32       | Här nollställs bit4 i ackumulator A     |
| 1B  | JMP  | \$21       | Här fortsätter man på adress \$21       |
| 1D  | LDA  | ##00010000 | Man ser här att bit4 = 1, övriga = 0    |
| 1F  | ORA  | \$32       | Här ettställs bit4 i ackumulator A      |
| 21  | STA  | \$32       | Spara ackumulator A på adress \$32      |
| 23  | .    | .          | .                                       |
| .   | .    | .          | .                                       |
| .   | .    | .          | .                                       |

$$A \oplus M \rightarrow A \quad \text{exclusive or,} \quad \text{EORA operandinfo}$$

Med denna operation kan man selektivt invertera enstaka bitar i en operand

$$\begin{array}{r}
 d_7 \ d_6 \ d_5 \ d_4 \ d_3 \ d_2 \ d_1 \ d_0 \quad \text{operand} \\
 \oplus \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \quad \text{bitmönster för selektiv invertering} \\
 \hline
 d_7' d_6' d_5' d_4' \ d_3 \ d_2 \ d_1 \ d_0 \quad \text{resultat}
 \end{array}$$

**Exempel F.15**

Följande instruktionssekvens för FLISP inverterar bit 2 av adress F1<sub>16</sub>

```

      .
      LDA    $F1
      EORA   #$04    Här är bit2 = 1
      STA    $F1
      .

```

**Tester**

Selektions- och iterationskonstruktioner är ofta baserade på villkor. På processornivå bildas dessa typiskt genom undersökning av en operand eller som resultatet av ett beräknat uttryck. För att undersöka en operand har processorn en eller flera testinstruktioner. Dessa operationer har vanligtvis två operander, en som skall undersökas och en annan som den testas mot för att undersökningen skall ge resultat. Tre typer av testinstruktioner är vanliga. Dessa kännetecknas av att de endast påverkar flaggregistret, dvs ger flaggpåverkan.

*Jämförelse med operand*

A-M  $\Rightarrow$  flaggpåverkan      compare,      CMPA    operandinfo

är en operation som använder subtraktion för att avgöra om en operand är mindre än, lika med eller större än en annan operand. Svaret finns i flaggregistret. Skillnaden bevaras ej.

**Exempel F.16**

Om ackumulator A innehåller det uppmätta temperaturvärdet 12<sub>16</sub> (18 °C) så utför instruktionen CMPB #20 operationen

```

      00010010
      - 00010100
      -----
      11111110   producerat resultat

```

med flaggvärdestilldelningen

C=1    V=0    Z=0    N=1

Den jämför således det uppmätta värdet med ett känt referensvärde 20 °C. Detta kan göras för att t ex starta uppvärmning när N-flaggan blir "1".

*Jämförelse med noll*

A-0  $\Rightarrow$  flaggpåverkan      test,      TSTA  
 M-0  $\Rightarrow$  flaggpåverkan      TST    operandinfo

är en operation som subtraherar noll ifrån operanden för att avgöra om den är noll, positiv eller negativ. Svaret finns i flaggregistret.

**Exempel F.17**

Om adress 98<sub>16</sub> innehåller värdet 45<sub>16</sub> så utför instruktionen TST \$98 operationen

$$\begin{array}{r} 01000101 \\ - 00000000 \\ \hline 01000101 \end{array} \quad \text{producerat resultat}$$

och ger Z- och N-flaggorna värdena

$$Z:=0 \quad N:=0$$

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 F.3. Observera att när tal utan tecken jämförs med 0 så behövs bara relationerna = och  $\neq$ .

Tabell F.3

| relation | tal utan tecken          | tal med tecken                                 |
|----------|--------------------------|------------------------------------------------|
| >        | $C' \cdot Z'$            | $(N \oplus V)' \cdot Z'$                       |
| $\geq$   | $C'$                     | $(N \oplus V)'$                                |
| =        | $Z$                      | $Z$                                            |
| $\neq$   | $Z'$                     | $Z'$                                           |
| $\leq$   | $(C' \cdot Z')' = C + Z$ | $((N \oplus V)' \cdot Z')' = (N \oplus V) + Z$ |
| <        | $C$                      | $(N \oplus V)$                                 |

Observera att flaggvärdena antas vara resultat av en subtraktion och att lånebiten  $b_8$  finns 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  $N \oplus V$ . Om overflow inträffar vet man ju att teckenbiten N får fel värde. Eftersom V samtidigt får värdet 1 inverterar man N genom att bilda  $N \oplus V$  ( $N \oplus 1 = N'$ ). Om overflow inte inträffar ger  $N \oplus V$  värdet N eftersom  $N \oplus 0 = N$ .

Tänk igenom hur uttrycken verkligen representerar respektive relation!

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

## Test av operandbitar

$A \wedge M \Rightarrow$  flaggpåverkan      bit test,      BITA operandinfo

är en operation som utför OCH-operationen 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 är således ett enklare och bättre sätt att undersöka individuella bitar än att göra det med skiftoperationer och C-flaggan.

## Exempel F.18

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

|          | d <sub>7</sub> | d <sub>6</sub> | d <sub>5</sub> | d <sub>4</sub> | d <sub>3</sub> | d <sub>2</sub> | d <sub>1</sub> | d <sub>0</sub> |                               |
|----------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-------------------------------|
| $\wedge$ | 0              | 0              | 0              | 0              | 1              | 0              | 0              | 0              | operand i A                   |
|          | 0              | 0              | 0              | 0              | x <sub>3</sub> | 0              | 0              | 0              | mask                          |
|          |                |                |                |                |                |                |                |                | resultat hamnar <u>ej</u> i A |

N-flaggan får värdet

$$N = 0$$

Z- flaggan får värdet

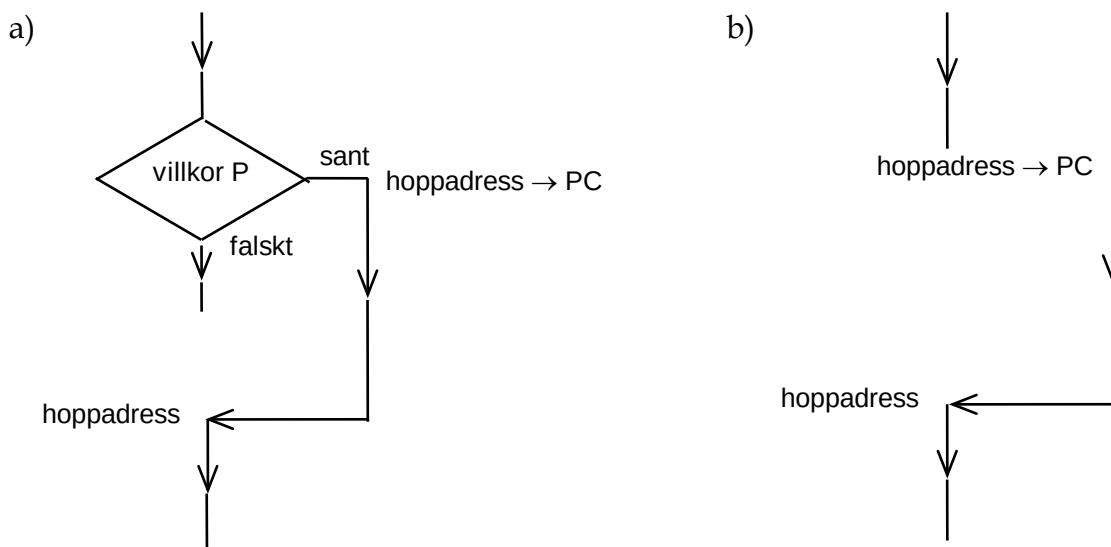
$$Z = 1 \text{ om } x_3 = 0$$

eller

$$Z = 0 \text{ om } x_3 = 1$$

## Hopp i program

Denna kategori av instruktioner är också oerhört viktig för selektions- och iterationskonstruktioner. I båda dessa konstruktionstyper måste **den raka exekveringsföljden** i en sekvens kunna brytas. I ett maskinprogram görs detta med **hopp**. Hopp sker genom att programräknaren inte ökas till nästa adress i en sekvens utan laddas med en ny adress. Ett hopp kan vara **villkorligt** (eng. **conditional**) eller **ovillkorligt** (eng. **unconditional**), se figur F.15.



Figur F.15 Exekveringsföljd vid a) villkorliga resp. b) ovillkorliga hopp.

Av denna ser man att den raka exekveringsföljden alltid bryts vid ett ovillkorligt hopp, då programräknaren laddas med en hoppadress och exekveringen fortsätter med början på denna. Instruktioner för ovillkorliga hopp benämns vanligen **"jump"-instruktioner**.

Vid villkorliga hopp bryts den raka exekveringsföljden endast om ett specificerat **hoppvillkor P** är uppfyllt (sant) och då laddas programräknaren med en hoppadress. Om villkoret inte är sant (dvs falskt) så fortsätts den raka exekveringsföljden. Man kan säga att exekveringsföljden har en **förgrening** (eng. **branch**) via en villkorlig hoppinstruktion. Därför benämns instruktioner för villkorliga hopp **"branch"-instruktioner**

Hoppen behöver inte alltid gå framåt i programmet såsom visas i figur F.15 utan kan också gå bakåt.

Instruktioner för villkorliga hopp uttrycks ofta på följande sätt

|                   |                                                                   |
|-------------------|-------------------------------------------------------------------|
| <b>if P then</b>  | "branch" om P är sann                                             |
| PC := HOPPADDRESS | (med symbolen := menas att PC<br>tilldelas värdet efter symbolen) |

Om hoppvillkoret P är sant eller falskt kan avgöras med hjälp av flaggorna i flaggregistret, t ex m h a relationerna i tabell F.3. Hoppvillkoret används för att testa ett resultat som har erhållits i instruktionsexekveringen. De olika hoppvillkoren framgår av instruktionslistan, där man lägger märke till att de förekommer i par. Om det finns en "branch"-instruktion för ett hoppvillkor så finns också en annan "branch"-instruktion för det motsatta hoppvillkoret.

### Exempel F.19

|                                                                                                                                                                 |                                    |          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------|----------|
| FLIS-instruktionen BVS \$63 tyds på följande sätt:                                                                                                              |                                    |          |
| <b>if V=1 then</b>                                                                                                                                              | "branch if V is set"               | BVS \$63 |
| PC := 63 <sub>16</sub>                                                                                                                                          | (hoppa om "overflow"-flaggan är 1) |          |
| I instruktionslistan finns då också en "branch"-instruktion BVC för hopp om "overflow"-flaggan ej är 1, dvs 0. Den kan med BVC \$AB beskrivas på följande sätt: |                                    |          |
| <b>if V=0 then</b>                                                                                                                                              | "branch if V is cleared"           | BVC \$AB |
| PC := AB <sub>16</sub>                                                                                                                                          | (hoppa om "overflow"-flaggan är 0) |          |

### Exempel F.20

|                                                                                                                                                                                                                                                                                                                                                   |                           |          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------|----------|
| Antag för FLISP att en operand med tecken (med 2-komplementrepresentation) i ackumulator A jämförs med 0 genom instruktionen TSTA. För att basera ett villkorligt hopp på denna jämförelse finns i instruktionslistan "branch"-instruktionen BLE för hopp om operanden är mindre eller lika med noll. Den beskrivs med BLE \$56 på följande sätt: |                           |          |
| <b>if (N⊕V)+Z=1 then</b>                                                                                                                                                                                                                                                                                                                          | "branch if less or equal" | BLE \$56 |
| PC := 56 <sub>16</sub>                                                                                                                                                                                                                                                                                                                            | (hoppa om (N⊕V) + Z=1)    |          |



**Exempel F.23**

Om hoppet i ex F.22 istället görs med "jump"-instruktionen blir maskinprogrammet:

| adress<br>hex | innehåll<br>hex | kommentar                   |
|---------------|-----------------|-----------------------------|
| 00            | 33              | OPkod för JMP               |
| 01            | 09              | hoppadress                  |
| 02            | .               | början på nästa instruktion |
| .             | .               |                             |
| .             | .               |                             |
| 09            | .               | hoppadress                  |
| .             | .               |                             |

Genom att specificera hoppet med avståndet till hoppadressen i maskininstruktionen görs hoppet **positionsberoende**, (eng **position independent**) dvs hoppet sker till rätt adress inom programmet oberoende av var programmet placeras i minnet.

Om maskininstruktionen för ett hopp innehåller den explicita hoppadressen är hoppet positionsberoende. I vissa fall är det viktigt med positionsberoende hopp, ex vis när det är nödvändigt att hoppa till ett program som börjar på en bestämd adress, ett s k **minnesresident** program.

Positionsberoende hopp är väsentliga för att ett program skall kunna vara **relokerbart**, dvs kunna placeras var som helst i minnet.

**Mer om adressering av data**

Tidigare i detta avsnitt har adresseringsmodernerna

- inherent (eng inherent)
- omedelbar (eng immediate)
- absolut, direkt (eng absolute, direct)
- PC-relativ (eng PC-relative)

definierats i samband med att instruktionerna beskrevs.

Vid absolutadressering adresseras en operand i minnet genom att operandadressen, den s k **effektivadressen**, **EA** (eng **effective address**) explicit ingår i instruktionen. Absolutadressering är endast lämplig när data uppträder på fasta adresser.

Vi skall nu se på några vanliga alternativ till absolutadressering. Det är adresseringsmoder egenskaper 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 positionsberoende program också skall vara positionsberoende

### Indirekt adressering

Ett program kan från körning till körning, använda sig av operander med varierande lägen i M genom instruktioner med indirekt adressering. I dessa fall innehåller instruktionen ej operandens effektivadress utan anger istället det läge, minnesadress eller processorregister, i vilket effektivadressen finns placerad. Man skall därvid se effektivadressen som en parameter till programmet.

När effektivadressen finns lagrad i primärminnet på en adress som explicit finns med i instruktionen, säger man helt generellt att adresseringsmoden är **indirekt** (eng **indirect**). I instruktionen ges en minnesadress och på den finns adressen till operanden.

Den vanligaste formen för indirekt adressering är att låta ett processorregister innehålla effektivadressen och tjäna som **pekare** (eng **pointer**) till operanden. Processorn har då maskininstruktioner använder innehållet i detta register som effektivadress. Pekarregistret anges explicit i instruktionen. Denna adresseringsmod kallas **register-indirekt** (eng **register indirect**).

### Register-indirekt adressering med konstant "offset" (indexerad adressering)

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

- **basadress** (startadress)
- **operandavstånd** (eng. **offset**), som är operandens adressavstånd från basadressen
- **längd**, som är den sista operandens avstånd från basadressen.

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

Det finns tre vanliga varianter av indexerad adressering. De skiljer sig åt i sättet att specificera basadress och index och kan sammanfattas enligt

- a) basadress i instruktionen och index i processorregister
- b) basadress i processorregister och index i instruktionen
- c) basadress i ett processorregister och index i ett annat processorregister

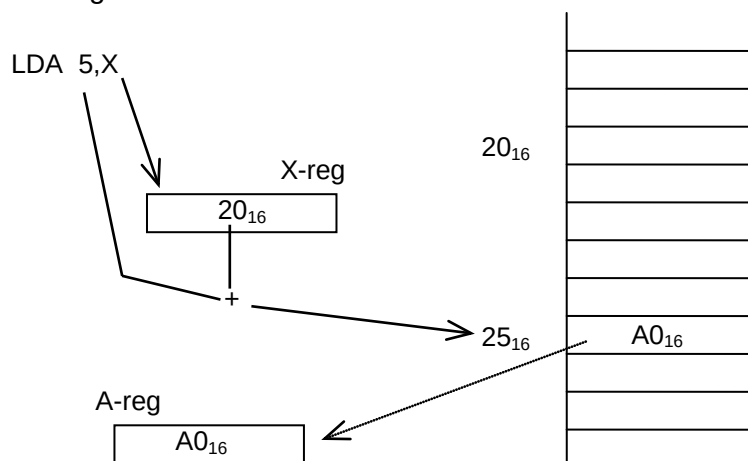
### Exempel F.24

Hos FLISP har de två första varianterna formen  $MNE\ n, r_{bas}$  där  $n$  är ett positivt eller negativt heltal, i området  $-128 \leq n \leq 127$  och  $r_{bas}$  något av registren X, Y eller SP. Adresseringsmoden anges av att kommatecknet föregås av  $n$ .



## Exempel F.25

Indexerad adressering enligt variant a) eller b) sker ex vis med instruktionen **LDA 5,X**. Den läser data på adressen  $X + 5$  och placerar det i ackumulator A, såsom visas i figur F.16.



Figur F.16 Principen för indexerad adressering enligt variant b).

Den har i maskinspråket utseendet

| maskininstruktion | operation                |
|-------------------|--------------------------|
| F3 (OPkod)        |                          |
| 05 (n)            | $M(X + n) \rightarrow A$ |

OPkoden  $F3_{16}$  anger:

- att det är en "load"-instruktion
- att A-registret är destinationsregister
- att X-registret används för basadress (eller index)
- att byten efter OPkoden innehåller index (eller basadressen)

Av figur F.16 ser man att X-registret här innehåller datastrukturens basadress och att index finns i instruktionen, dvs variant b).

Även en tolkning enligt variant a) är möjlig i detta fall. I så fall innehåller X-registret index som adderas till basadressen 5. Den effektiva adressen (adressen till data) blir i båda fallen densamma.

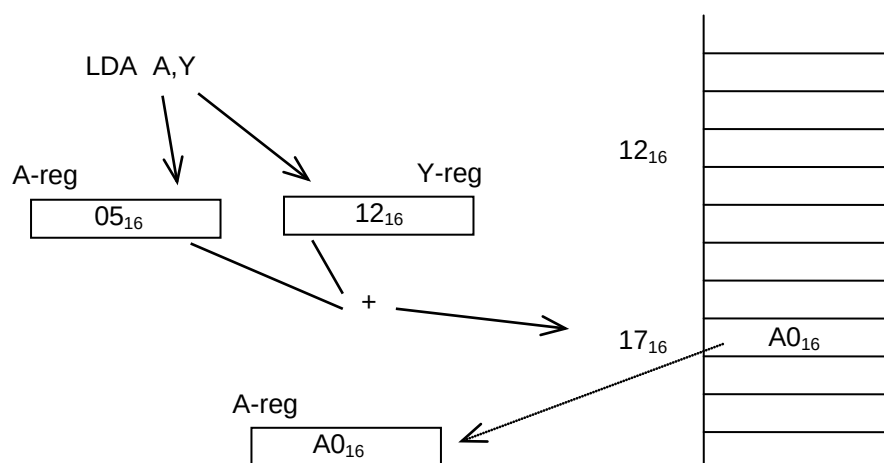
### Register-indirekt adressering med ackumulator-"offset"

#### Exempel F.26

Den tredje varianten har hos FLISP formen  $MNE\ A, r_{bas}$  där  $r_{bas}$  är X- eller Y-registret, som innehåller basadressen. Ackumulator A innehåller index, som är ett positivt eller negativt heltal i området  $[-128, +127]$ . Adresseringsmoden anges av att kommatecknet föregås av A.

#### Exempel F.27

Indexerad adressering enligt variant c) sker ex vis med FLISP-instruktionen  $LDA\ A, Y$ . Den läser data på adressen  $Y + A$  och placerar det i ackumulator A, såsom visas i figur F.17.



Figur F.17 Principen för indexerad adressering enligt variant c).

Den har i maskinspråket utseendet

| maskininstruktion | operation                |
|-------------------|--------------------------|
| FA (OPkod)        | $M(Y + A) \rightarrow A$ |

OPkoden  $FA_{16}$  anger:

- att det är en "load"-instruktion
- att A-registret är destinationsregister
- att Y-registret används för basadress (eller index)
- att A-registret används för index (eller basadress)

Av figur F.17 ser man att Y-registret här innehåller datastrukturens basadress och att index finns i A-registret. Även den omvända tolkningen är möjlig.

*Register-indirekt adressering med "autoinkrement" och "autodekrement"*

Register-indirekt adressering har en naturlig utvidgning som är lämplig att använda när samhörande operander lagrats på konsekutiva (på varandra följande) adresser och man i programmet vill flytta sig från operand till operand. Detta är ex vis vanligt när man arbetar med tabelliknande datastrukturer.

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 eller bakåt i datastrukturen.

Instruktioner med denna typ av automatisk modifiering av pekarregistrets innehåll har adresseringsmoden **register-indirekt med "autoinkrement" eller "autodekrement"**.

**Pre- resp. post-inkrementering** innebär att instruktionen ökar innehållet i pekarregistret med 1 före resp. efter det att adresserad operand använts för att förbereda användning av nästa element i strukturen.

**Pre- resp. post-dekrementering** innebär att instruktionen minskar innehållet i pekarregistret med 1 före resp. efter det att adresserad operand använts.

**Exempel F.28**

Hos FLISP finns register-indirekt adressering med pre- och post-inkrementering

MNE ,+X

MNE ,X+

och med pre- och post-dekrementering

MNE ,-X

MNE ,X-

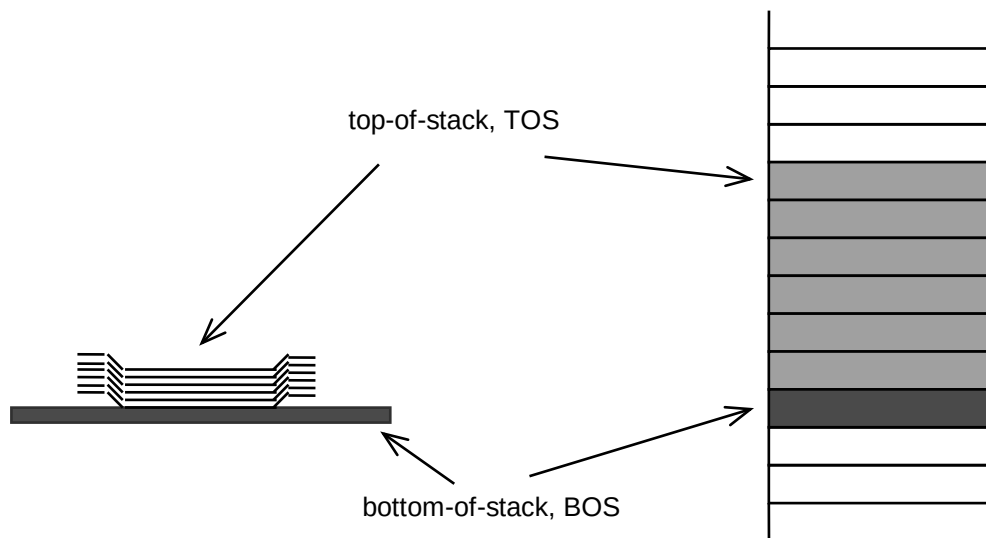
För laddning av A finns då instruktionerna

|          |                                             |         |                                             |
|----------|---------------------------------------------|---------|---------------------------------------------|
| LDA ,+X  | $X+1 \rightarrow X$<br>$M(X) \rightarrow A$ | LDA ,-X | $X-1 \rightarrow X$<br>$M(X) \rightarrow A$ |
| LDA 1,X+ | $M(X) \rightarrow A$<br>$X+1 \rightarrow X$ | LDA ,X- | $M(X) \rightarrow A$<br>$X-1 \rightarrow X$ |

### Stackbyggnad och användning

Med "store"- och "load"-instruktioner som har register-indirekt adressering med autoinkrement och autodekrement kan man skapa och använda en datastruktur som kallas **stack**.

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 F.18. Basläget kallas "**bottom-of-stack**", **BOS** och läget för det sist pålagda elementet kallas "**top-of-stack**", **TOS**.



Figur F.18 En stackstruktur.

Nya element tillförs ovanifrån, dvs läggs på toppen. Principen för att komma åt de binära orden, höet och tallrikarna är "sist in - först ut" (eng. last in - first out, LIFO). Processen att lägga nya element på stacken benämns "**push**" och processen att hämta element ifrån stacken benämns "**pull**" (ibland "**pop**").

Stacken är en **LIFO-struktur**, som främst används för att temporärt lagra data på ett enkelt sätt. Ett smidigt sätt att göra detta är via "store"- resp. "load"-instruktioner med register-indirekt adressering och med automatisk dekrementering resp. inkrementering. Ett pekarregister, kallat **stackpekare SP** (eng **stack pointer**) får hålla reda på TOS-läget enligt figur F.19. Stacken initieras när stackpekaren en gång laddades med BOS-adressen.



Figur F.19 Stackens tillväxt och adressering via stackpekaren.

I en processor finns i regel minst ett stackpekaregister, SP. Det finns då också speciella "push"- och "pull"-instruktioner som lagrar data från processorns interna register på stacken och hämtar data från stacken till interna register

$$SP-1 \rightarrow SP$$
$$r_{src} \rightarrow M(SP)$$
$$M(SP) \rightarrow r_{dst}$$
$$SP+1 \rightarrow SP$$

push  $r_{src}$

pull  $r_{dst}$

### Exempel F.29

FLISP har en stackpekare SP och "push"-instruktioner samt "pull"-instruktioner.

"PUSH"

PSHA

PSHX

PSHY

PSHC

"PULL"

PULA

PULX

PULY

PULC

### Exempel F.30

FLISP-instruktionen PSHA placerar en kopia av det ord som finns i ackumulator A på stacken, som definieras av stackpekare SP.

maskininstruktion

10 (OPkod)

operation

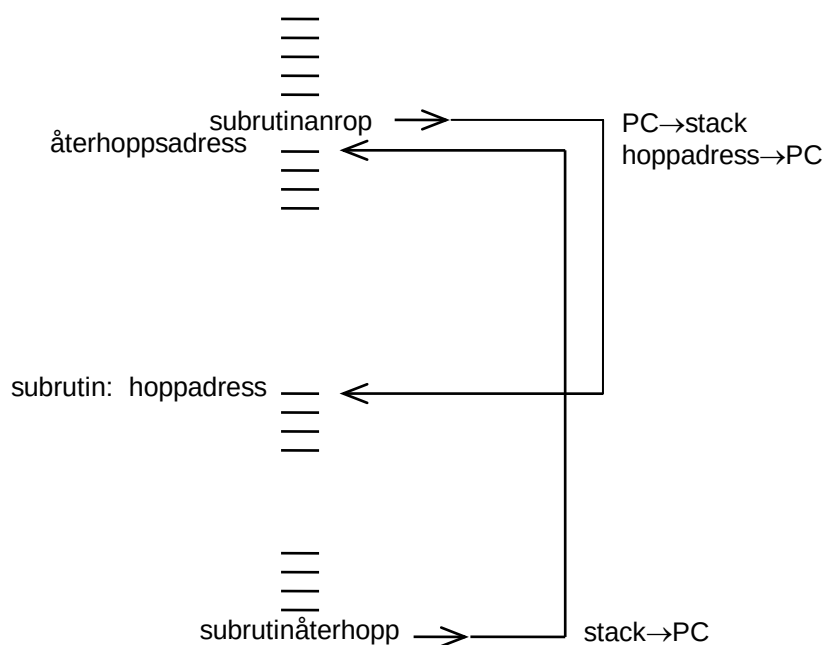
$$\text{SP-1} \rightarrow \text{SP}$$
$$A \rightarrow M(SP)$$

## Hopp till subrutin

Utöver de hoppinstruktioner som tidigare beskrivits har processorn instruktioner för att göra hopp till och från s k **subrutiner** (eng **subroutines**). Med dessa kan programmerare modularisera sina program och placera modulerna fritt i minnet. Hopp till subrutin är vanligen ovillkorliga och kallas subrutinanrop. Följande instruktioner är typiska

|                                  |                                            |
|----------------------------------|--------------------------------------------|
| återhoppadress → stack           |                                            |
| hoppadress → PC                  | jump to subroutine<br>branch to subroutine |
| återhoppadress (från stack) → PC | return from subroutine                     |

Vid de två första instruktionerna sker hopp till den hoppadress som operand-informationen specificerar. Innan de laddar programräknaren med hoppadressen, lagrar de först programräknarens innehåll på stacken (att stackpekaren justeras automatiskt är underförstått). På stacken lagras således adressen till den instruktion i det anropande programmet som ligger omedelbart efter hoppinstruktionen. Detta görs för att man senare skall kunna hoppa tillbaka till denna adress. Återhopp sker via en "pull"-instruktion, som har den mnemoniska beteckningen RTS. Den återladdar programräknaren med adress från stacken. Denna utför exakt samma sak som en "pull"-instruktion PULPC skulle göra. I realiteten utför den ett ovillkorligt hopp till subrutinens återhoppadress. Exekveringsföljden vid hopp till en subrutin och återhopp visas i figur F.20.



Figur F.20 Exekveringsföljd vid subrutinanrop.

Med instruktioner för anrop av och återhopp från subrutin kan programmeraren på ett smidigt sätt använda programavsnitt som placerats på annat ställe i minnet genom att anropa dem som subrutiner.

**Exempel F.31**

Hos FLISP finns följande instruktioner för subrutinanrop

JSR adr             $SP-1 \rightarrow SP; PC \rightarrow M(SP)$   
                      $adr \rightarrow PC$

BSR adr             $SP-1 \rightarrow SP; PC \rightarrow M(SP)$   
                      $adr = PC + \text{offset} \rightarrow PC$

och följande instruktion för återhopp

RTS                 $M(SP) \rightarrow PC; SP+1 \rightarrow SP$

BSR är som synes en "branch"-instruktion vilket innebär att hoppadressen ingår som ett avstånd (offset ) i maskininstruktionen. När instruktionen utförs så adderas avståndet till PC's värde och bildar hoppadressen.

## Övningsuppgifter

Övningsuppgifterna i kapitel F avser FLIS-processorn, vars instruktioner och motsvarande koder definieras i "INSTRUKTIONSLISTA FÖR FLISP".

**F.1** Ge i hexadecimal form maskininstruktionen för

- a) LDA \$3A
- b) LDA 58
- c) STA 235
- d) STA \$EB
- e) LDA #\$56
- f) LDA #%01010110
- g) LDA #86

**F.2** Ett antal på varandra följande minnesord har följande hexadecimala innehåll:

- a) F1, 20, E1, 21, 96, 0F, A9, 25, 06, 97, 10.
- b) F0, 80, A1, 50, FA, 94, 4A, 99, F0, 1B.
- c) 90, 40, F1, A0, 0B, F4, 54, 00.

Den första byten ( $F1_{16}$ ,  $F0_{16}$  resp  $90_{16}$ ) är en operationskod för FLIS-processorn. Översätt sekvensen till assemblerspråk, dvs disassemblera sekvensen.

**F.3** Ge en sekvens av LDA- och STA-instruktioner (dvs ett programavsnitt) som flyttar innehållet på adresserna  $1A_{16}$ - $1C_{16}$  till adresserna  $2A_{16}$ - $2C_{16}$ .

**F.4** Ge en sekvens av LDA- och STA-instruktioner som placerar telefonnumret 4631678900 i NBCD-form på adresserna  $21_{16}$ - $25_{16}$ .

**F.5** Manuell översättning från assemblerspråk till maskinkod kallas ofta "**handassemblering**".

- a) Handassemblera följande instruktionssekvens. Första instruktionen skall placeras på adress  $80_{16}$ . Hur många minnespositioner upptar instruktionssekvensen?

```
LDA    $10
ANDA   #$F
ORA     $11
EORA   #$44
ANDA   #$EE
COMA
STA     $10
```

- b) Antag att talet  $10110110_2$  finns på adress  $10_{16}$ . Ange i hexadecimal form det tal som kommer att finnas i denna minnesposition efter exekvering av instruktionssekvensen ovan, om adress  $11_{16}$  innehåller  $01_{16}$  före exekveringen.

**F.6** a) Skriv en instruktionssekvens som inverterar bitarna  $b_0$  och  $b_7$  i A-registret och ettställer bitarna  $b_2$  och  $b_5$  samt nollställer bit  $b_4$ .

- b) Handassemblera instruktionssekvensen. Första instruktionen skall placeras på adress  $20_{16}$ .

**F.7** I minnet finns data enligt figuren till höger.

Skriv en instruktionssekvens som ökar innehållet på adressen  $80_{16}$  med 5 och minskar innehållet på adressen  $81_{16}$  med 7.

Vidare skall instruktionssekvensen addera innehållen på minnesadresserna  $82_{16}$  och  $83_{16}$  samt placera summan på adressen  $84_{16}$ .

Översätt instruktionssekvensen till maskinkod (hexadecimal form) och placera den i minnet med början på adressen  $20_{16}$ .

| Adr       |           |
|-----------|-----------|
| $80_{16}$ | $56_{16}$ |
| $81_{16}$ | $23_{16}$ |
| $82_{16}$ | $05_{16}$ |
| $83_{16}$ | $16_{16}$ |
| $84_{16}$ | $B7_{16}$ |



- F.8** Två åttabitors tal finns lagrade i minnet på adresserna  $50_{16}$  och  $51_{16}$ . Skriv en instruktionssekvens som adderar talen och placerar summan på adressen  $52_{16}$  i minnet. Summan förutsätts rymmas i åtta bitar ( $<256$ ). Översätt instruktionssekvensen till maskinkod. Startadressen skall vara  $10_{16}$ .
- F.9** Skriv en instruktionssekvens som adderar 9 till talet som finns på adressen  $60_{16}$  i minnet och placerar summan (åtta bitar) på adressen  $54_{16}$ . Översätt instruktionssekvensen till maskinkod. Startadress skall vara  $18_{16}$ .
- F.10** På adress  $65_{16}$  finns ett tal med inbyggt tecken (tvåkomplementrepresentation).
- a) Skriv en instruktionssekvens som multiplicerar talet med 4.  
Vad hamnar på adress  $65_{16}$  om innehållet där från början är
- b)  $18_{10}$   
c)  $-23_{10}$   
d)  $38_{10}$
- F.11** a) "Jump"-instruktionen JMP  $\$50$  är placerad med början på adress  $37_{16}$ .  
Vad är dess maskinkod?
- b) "Branch"-instruktionen BRA  $\$50$  är placerad med början på adress  $37_{16}$ .  
Vad är dess maskinkod?
- F.12** a) "Jump"-instruktionen JMP  $\$05$  är placerad med början på adress  $20_{16}$ .  
Vad är dess maskinkod?
- b) "Branch"-instruktionen BRA  $\$05$  är placerad med början på adress  $20_{16}$ .  
Vad är dess maskinkod?
- F.13** Instruktionssekvensen nedan multiplicerar innehållen (talen x och y) på minnesadresserna  $5C_{16}$  (x) och  $5D_{16}$  (y) genom att addera det ena talet till register A så många gånger som det andra talet anger. Varje gång det uppstår en minnessiffra ut från denna addition ökas den mest signifikanta byten av produkten med ett. Produkten (16 bitar) lagras på adresserna  $5E_{16}$  och  $5F_{16}$  med mest signifikant del på  $5E_{16}$ . Minnesadressen  $5B_{16}$  är ledig och kan användas.
- Översätt instruktionssekvensen till maskinkod. Startadress skall vara  $30_{16}$ .  
Analysera instruktionssekvensen genom att rita en maskinberoende flödesplan som beskriver multiplikationsförloppet.

| Läge    | Operation | Operand | Kommentar                                   |
|---------|-----------|---------|---------------------------------------------|
| START   | CLR       | $\$5E$  | Nollställ variabeln för produkten.          |
|         | CLR       | $\$5F$  | "-"                                         |
|         | LDA       | $\$5C$  | Hämta talet x till reg A.                   |
|         | TSTA      |         | Undersök talet x.                           |
|         | BEQ       | SLUT    | Talet x = 0. Hoppa ut.                      |
|         | STA       | $\$5B$  | Använd talet x som räknare i minnet.        |
|         | TST       | $\$5D$  | Undersök talet y.                           |
|         | BEQ       | SLUT    | Talet y = 0. Hoppa ut.                      |
| ADDLOOP | CLRA      |         | Nollställ reg A.                            |
|         | ADDA      | $\$5D$  | Addera talet y till reg A.                  |
|         | BCC       | NOCARRY | Kontrollera om summan i reg A > 255.        |
| NOCARRY | INC       | $\$5E$  | Ja, öka innehållet i produktens höga byte.  |
|         | DEC       | $\$5B$  | Minska talet x med ett.                     |
|         | BNE       | ADDLOOP | Upprepa tills talet x = 0.                  |
| SLUT    | STA       | $\$5F$  | Nu har vi adderat talet y till reg A x ggr. |
|         | .         |         | Skriv därför produktens låga byte i minnet. |
|         | .         |         |                                             |
|         | .         |         |                                             |

**F.14** I följande instruktionssekvens genereras en puls i position 0 på utport FB<sub>16</sub>.

| Läge   | Operation | Operand    | Kommentar |
|--------|-----------|------------|-----------|
| START  | CLR       | \$FB       |           |
|        | LDA       | #25        |           |
|        | STA       | \$10       |           |
| SCOUNT | DEC       | \$10       |           |
|        | BNE       | SCOUNT     |           |
|        | LDA       | ##00000001 |           |
|        | STA       | \$FB       |           |
|        | LDAB      | #100       |           |
|        | STA       | \$10       |           |
| PCOUNT | DEC       | \$10       |           |
|        | BNE       | PCOUNT     |           |
|        | COMA      |            |           |
|        | STA       | \$FB       |           |

- Rita en maskinberoende flödesplan för sekvensen.
- Antag att FLIS-processorn har klockfrekvensen 1 MHz och att sekvensen startas vid tidpunkten  $t = 0$ . När i tiden genereras pulsen? Hur lång är den?
- Kommentera instruktionerna på lämpligt sätt. Kommentarer ska helst vara maskinberoende.
- Översätt instruktionssekvensen till maskinkod. Antag att START skall läggas på adressen 20<sub>16</sub>. Måste den maskinkod du nu producerat alltid ligga på dessa adresser?

**F.15** Vad blir den *effektiva adressen* i följande instruktioner om innehållet i X-registret = 20<sub>16</sub>. För instruktioner som arbetar med data är *effektiva adressen* adressen till data. Översätt instruktionerna till maskinkod.

- LDA 0,X
- LDA 16,X
- LDA -16,X
- LDA ,X+
- LDA ,-X

**F.16** Skriv en instruktionssekvens som omvandlar ett 4-bitars binärt tal till Graykod. Det binära talet skall kontinuerligt läsas från bit b<sub>0</sub>-b<sub>3</sub> på inport FB<sub>16</sub>. Bit b<sub>4</sub>-b<sub>7</sub>:s värden är ej kända och kan variera mellan läsningarna. Graykoden skall matas ut på bit b<sub>0</sub>-b<sub>3</sub> på utport FB<sub>16</sub> med bit b<sub>4</sub>-b<sub>7</sub> nollställda. Graykoden för siffrorna 0<sub>16</sub>-F<sub>16</sub> finns lagrade i bit b<sub>0</sub>-b<sub>3</sub> i adress 30<sub>16</sub> - 3F<sub>16</sub>, med bit b<sub>4</sub>-b<sub>7</sub> nollställda.

**F.17** I den första fjärdedelen av minnet innehåller vid ett visst tillfälle varje position värdet av adressen.

Ex:

| adress<br>(hex) | innehåll<br>(hex) | Ange innehållen i registren A och X efter varje instruktion i instruktionssekvensen nedan. |
|-----------------|-------------------|--------------------------------------------------------------------------------------------|
| 00              | 00                | LDX #\$12                                                                                  |
| .               | .                 | LDA \$3E                                                                                   |
| .               | .                 | LDA 0,X                                                                                    |
| 1C              | 1C                | LDX 18,X                                                                                   |
| .               | .                 | LDA \$E0,X                                                                                 |
| .               | .                 | LDX 4,X                                                                                    |
| 32              | 32                | LDA ,X+                                                                                    |
| .               | .                 | LDA 0,X                                                                                    |
| .               | .                 | LDA ,-X                                                                                    |
| 3F              | 3F                |                                                                                            |

- F.18** Skriv en subrutin MASK som nollställer bit 7 i varje byte inom adressområdet  $40_{16}$ - $60_{16}$ .
- F.19** En tabell med 5 olika 8-bitars tal utan tecken lagras med början på adress  $18_{16}$  i minnet. Skriv en instruktionssekvens som letar upp det största talet.
- Talets värde och läge skall anges i minnespositionerna VALUE resp PLACE, vilka anses fördefinierade.
- F.20** Ge en sekvens av instruktioner som definierar en stack med början på adress  $F0_{16}$  och placerar operanderna  $06_{16}$ ,  $35_{16}$  och  $28_{16}$  på stacken. Vad är stackpekarens innehåll när sekvensen genomlöpts av processorn?
- F.21** Utgå ifrån att stacken redan är definierad (vilket är den vanligaste situationen vid skrivning av programavsnitt) och ge en sekvens av instruktioner som använder stacken för att flytta innehållet i ackumulator A till register X.
- F.22** Antag att vi har ett huvudprogram och två subrutiner enligt nedan:

| Huvudprogram |      |                | Subrutin Tvåkomp |      |                | Subrutin Plusett |      |       |
|--------------|------|----------------|------------------|------|----------------|------------------|------|-------|
| Adr          | Mem. | Opr            | Adr              | Mem  | Opr            | Adr              | Mem  | Opr   |
| 40           | LDA  | \$10           | 50               | PSHA |                | 60               | PSHA |       |
| 42           | JSR  | \$50 (Tvåkomp) | 51               | NEGA |                | 61               | ADDA | #\$01 |
| 44           | JMP  | \$44           | 52               | STA  | \$11           | 63               | STA  | \$12  |
|              |      |                | 54               | JSR  | \$60 (Plusett) | 65               | PULA |       |
|              |      |                | 56               | PULA |                | 66               | RTS  |       |
|              |      |                | 57               | RTS  |                |                  |      |       |

Antag vidare att huvudprogrammet körs.

Fyll i minnesadress, instruktion, registervärden och stackens innehåll så att de motsvarar situationen efter det att motsvarande instruktion har exekverats. Markera även med en pil det minnesord som stackpekaren (SP) pekar på. Utgå från situationen efter första instruktionen, vilken finns given nedan.

| Adr            |    | Instruktion 1 |       | Adr       |    | Instruktion 2 |  |
|----------------|----|---------------|-------|-----------|----|---------------|--|
| 40             |    | LDA \$10      |       | 42        |    | JSR \$50      |  |
| Register:      |    | Stacken:      |       | Register: |    | Stacken:      |  |
| PC = $42_{16}$ | CA |               |       | PC =      | CA |               |  |
| A = $FF_{16}$  | CB |               |       | A =       | CB |               |  |
| SP = $D0_{16}$ | CC |               |       | SP =      | CC |               |  |
|                | CD |               |       |           | CD |               |  |
|                | CE |               |       |           | CE |               |  |
|                | CF |               |       |           | CF |               |  |
|                | D0 | 00            | ← (S) |           | C0 | 00            |  |

| Adr       |    | Instruktion 3 |  | Adr       |    | Instruktion 4 |  |
|-----------|----|---------------|--|-----------|----|---------------|--|
| Register: |    | Stacken:      |  | Register: |    | Stacken:      |  |
| PC =      | CA |               |  | PC =      | CA |               |  |
| A =       | CB |               |  | A =       | CB |               |  |
| SP =      | CC |               |  | SP =      | CC |               |  |
|           | CD |               |  |           | CD |               |  |
|           | CE |               |  |           | CE |               |  |
|           | CF |               |  |           | CF |               |  |
|           | D0 | 00            |  |           | D0 | 00            |  |

Adr Instruktion 5

| Register: |    | Stacken: |
|-----------|----|----------|
| PC =      | CA |          |
| A =       | CB |          |
| SP =      | CC |          |
|           | CD |          |
|           | CE |          |
|           | CF |          |
|           | D0 | 00       |

Adr Instruktion 6

| Register: |    | Stacken: |
|-----------|----|----------|
| PC =      | CA |          |
| A =       | CB |          |
| SP =      | CC |          |
|           | CD |          |
|           | CE |          |
|           | CF |          |
|           | D0 | 00       |

Adr Instruktion 7

| Register: |    | Stacken: |
|-----------|----|----------|
| PC =      | CA |          |
| A =       | CB |          |
| SP =      | CC |          |
|           | CD |          |
|           | CE |          |
|           | CF |          |
|           | D0 | 00       |

Adr Instruktion 8

| Register: |    | Stacken: |
|-----------|----|----------|
| PC =      | CA |          |
| A =       | CB |          |
| SP =      | CC |          |
|           | CD |          |
|           | CE |          |
|           | CF |          |
|           | D0 | 00       |

Adr Instruktion 9

| Register: |    | Stacken: |
|-----------|----|----------|
| PC =      | CA |          |
| A =       | CB |          |
| SP =      | CC |          |
|           | CD |          |
|           | CE |          |
|           | CF |          |
|           | D0 | 00       |

Adr Instruktion 10

| Register: |    | Stacken: |
|-----------|----|----------|
| PC =      | CA |          |
| A =       | CB |          |
| SP =      | CC |          |
|           | CD |          |
|           | CE |          |
|           | CF |          |
|           | D0 | 00       |

Adr Instruktion 11

| Register: | Stacken: |
|-----------|----------|
| PC =      | CA       |
| A =       | CB       |
| SP =      | CC       |
|           | CD       |
|           | CE       |
|           | CF       |
|           | D0       |
|           | 00       |

Adr Instruktion 12

| Register: | Stacken: |
|-----------|----------|
| PC =      | CA       |
| A =       | CB       |
| SP =      | CC       |
|           | CD       |
|           | CE       |
|           | CF       |
|           | D0       |
|           | 00       |

Adr Instruktion 13

| Register: | Stacken: |
|-----------|----------|
| PC =      | CA       |
| A =       | CB       |
| SP =      | CC       |
|           | CD       |
|           | CE       |
|           | CF       |
|           | D0       |
|           | 00       |

Adr Instruktion 14

| Register: | Stacken: |
|-----------|----------|
| PC =      | CA       |
| A =       | CB       |
| SP =      | CC       |
|           | CD       |
|           | CE       |
|           | CF       |
|           | D0       |
|           | 00       |

**F.23** Skriv ett programavsnitt som läser ett tal från inport  $FB_{16}$ . Om talet är  $00_{16}$  skall en subrutin FALL1 anropas. Talet skall då ligga i A-registret för att subrutinen skall kunna använda det för bearbetning. Om talet är  $18_{16}$  skall på motsvarande sätt en subrutin FALL2 anropas.

Om talet inte är lika med något dessa värden skall en subrutin FALL3 anropas.  
Rita också en flödesplan.

**F.24** Skriv en subrutin som adderar två 8-bitars naturligt binärkodade tal. De två talen skall finnas i Y- resp. A-registret vid anrop av subrutinen. Mest signifikanta byten skall returneras i Y-reg. och minst signifikanta byten i A-reg.

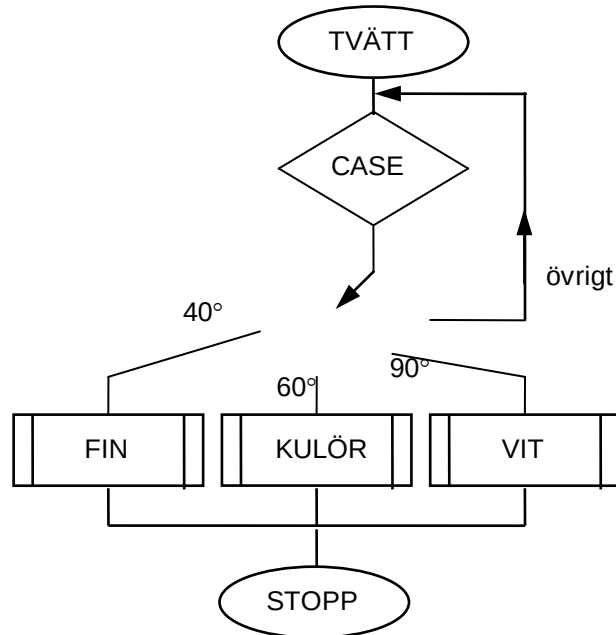
Av de interna registren får subrutinen endast påverka Y- och A-registren samt flaggregistret. Z-flaggan skall sättas till 1 om summan är lika med noll. De övriga flaggornas värden saknar betydelse.

Rita även en flödesplan.

**F.25** En enkel datorstyrd tvättmaskin har tre olika tvättprogram

- 40°C Fintvätt / 60°C Kulör tvätt / 90°C Vittvätt.

När tvättmaskinen startats agerar den enligt flödesplanen nedan.



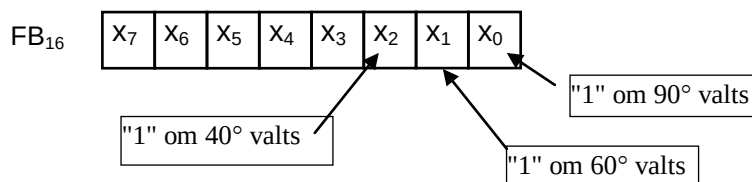
- a) Rita om flödesplanen så att fyrvalssituationen realiseras som en följd av tvåval (if-then-else)!
- b) Användaren kan trycka på tre olika knappar som svarar mot de olika tvättprogrammen. Den valda tvättemperaturen är åtkomlig för processorn via inporten  $FB_{16}$  enligt figuren nedan.

Observera att vissa värden på  $x_2x_1x_0$  representerar ogiltiga val (t.ex. om både  $x_1$  och  $x_0$  är ett). Dessa fall motsvaras av alternativet "övrigt" i flödesplanen.

De olika tvättprogrammen är subrutiner med början på de symboliska adresserna "Fin", "Kulör" respektive "Vit". Omsätt den nya flödesplanen i ett program skrivet med FLEX instruktioner.

Ledning:

Varje tvättemperatur motsvaras av ett unikt bitmönster i de tre minst signifikanta positionerna i data från inport  $FB_{16}$ , 60°C Kulör tvätt motsvaras t.ex. av  $x_2x_1x_0 = 010$ .



**F.26** Skriv en subrutin som skriver in talen 00,01,02,...,7F<sub>16</sub> i adresserna 00, 01, ..., 7F<sub>16</sub>.

**F.27** Vad är det för fel på följande subrutin?

```

SUBRTN  PSHA
        LDA    TAL_1
        ADDA   TAL_2
        NEGA
        RTS
  
```

**Svar till vissa av övningsuppgifterna**

**F.1** a) F1 3A                      b) F1 3A  
c) E1 EB                        d) E1 EB  
e) F0 56                        f) F0 56  
g) F0 56

**F.2** a) LDA \$20                      b) LDA #\$80                      c) LDX #\$40  
STA \$21                            LDY \$50                            LDA \$A0  
ADDA #\$0F                        LDA A,Y                            ASLA  
ANDA \$25                        SUBA #\$4A                        LDA A,X  
NEGA                              ANDA #\$F0                        JSR 0,X  
CMPA #\$10                        TFR Y,X

**F.3** LDA \$1A  
STA \$2A  
LDA \$1B  
STA \$2B  
LDA \$1C  
STA \$2C

**F.4** LDA #\$46  
STA \$21  
LDA #\$31  
STA \$22  
LDA #\$67  
STA \$23  
LDA #\$89  
STA \$24  
LDA #\$00  
STA \$25

**F.5** a) Adress: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C  
Innehåll: F1 10 99 0F AA 11 9B 44 99 EE 0A E1 10  
Instruktionssekvensen upptar 13 minnespositioner.

b) BD<sub>16</sub>

**F.6** a) EORA    #%10000001  
ORA        #%00100100  
ANDA        #%11101111

b) Adress: 20, 21, 22, 23, 24, 25  
Innehåll: 9B, 81, 9A, 24, 99, EF

**F.7** LDA \$80  
ADDA #5  
STA \$80  
LDA \$81  
SUBA #7  
STA \$81  
LDA \$82  
ADDA \$83  
STA \$84

Adress: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31  
Sekvens: F1 80 96 05 E1 80 F1 81 94 07 E1 81 F1 82 A6 83 E1 84

**F.8** LDA \$50  
 ADDA \$51  
 STA \$52

Adress: 10 11 12 13 14 15  
 Sekvens: F1 50 A6 51 E1 52

**F.9** LDA \$60  
 ADDA #9  
 STA \$54

Adress: 18 19 1A 1B 1C 1D  
 Sekvens: F1 60 96 09 E1 54

**F.10** a) ASL \$65      b)  $72_{10}$       c)  $-92_{10}$       d)  $-104_{10}$   
 ASL \$65

**F.11** a) 33, 50      b) 21, 17

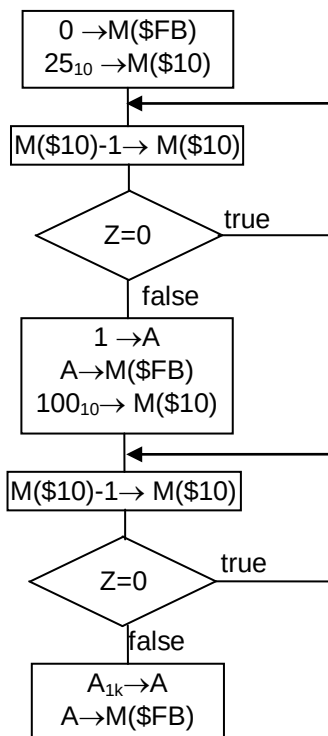
**F.12** a) 33, 05      b) 21, E3

**F.13** Instr.(hex)      Adress      Innehåll

|         |             |    |    |                                 |
|---------|-------------|----|----|---------------------------------|
| START   | CLR \$5E    | 30 | 35 |                                 |
|         |             | 31 | 5E |                                 |
|         | CLR \$5F    | 32 | 35 |                                 |
|         |             | 33 | 5F |                                 |
|         | LDA \$5C    | 34 | F1 |                                 |
|         |             | 35 | 5C |                                 |
|         | TSTA        | 36 | 09 |                                 |
|         | BEQ SLUT    | 37 | 24 |                                 |
|         |             | 38 | 15 | $(4C_{16} - 39_{16} = 13_{16})$ |
|         | STA \$5B    | 39 | E1 |                                 |
|         |             | 3A | 5B |                                 |
|         | TST \$5D    | 3B | 39 |                                 |
|         |             | 3C | 5D |                                 |
|         | BEQ SLUT    | 3D | 24 |                                 |
|         |             | 3E | 0D | $(4C_{16} - 3F_{16} = 0D_{16})$ |
|         | CLRA        | 3F | 05 |                                 |
| ADDLOOP | ADDA \$5D   | 40 | A6 | ADDLOOP = $40_{16}$             |
|         |             | 41 | 5D |                                 |
|         | BCC NOCARRY | 42 | 29 |                                 |
|         |             | 43 | 02 | $(46_{16} - 44_{16} = 02_{16})$ |
|         | INC \$5E    | 44 | 37 |                                 |
|         |             | 45 | 5E |                                 |
| NOCARRY | DEC \$5B    | 46 | 38 | NOCARRY = $46_{16}$             |
|         |             | 47 | 5E |                                 |
|         | BNE ADDLOOP | 48 | 25 |                                 |
|         |             | 49 | F6 | $(40_{16} - 4A_{16} = F6_{16})$ |
|         | STA \$5F    | 4A | E1 |                                 |
|         |             | 4B | 5F |                                 |
| SLUT    | ...         | 4C |    | $(SLUT = 4C_{16})$              |



F.14 a)



|        | Instruktion    | Adr. | Inneh. | ~ | Kommentar                                              |
|--------|----------------|------|--------|---|--------------------------------------------------------|
| START  | CLR \$FB       | 20   | 35     | 3 | Nollställ utport M(\$FB)                               |
|        | LDA #25        | 21   | FB     |   |                                                        |
|        |                | 22   | F0     | 2 | Sätt fördröjning                                       |
|        |                | 23   | 19     |   |                                                        |
|        | STA \$10       | 24   | E1     | 3 |                                                        |
|        |                | 25   | 10     |   |                                                        |
| SCOUNT | DEC \$10       | 26   | 38     | 4 | Vänta                                                  |
|        |                | 27   | 10     |   |                                                        |
|        | BNE SCOUNT     | 28   | 25     | 4 |                                                        |
|        |                | 29   | FC     |   | 26 <sub>16</sub> - 2A <sub>16</sub> = FC <sub>16</sub> |
|        | LDA #%00000001 | 2A   | F0     | 2 |                                                        |
|        |                | 2B   | 01     |   |                                                        |
|        | STA \$FB       | 2C   | E1     | 3 | Mata ut. Pulsstart.                                    |
|        |                | 2D   | FB     |   |                                                        |
|        | LDA #100       | 2E   | F0     | 2 | Sätt ny väntetid                                       |
|        |                | 2F   | 64     |   |                                                        |
|        | STA \$10       | 30   | E1     | 3 |                                                        |
|        |                | 31   | 10     |   |                                                        |
| PCOUNT | DEC \$10       | 32   | 38     | 4 | Vänta                                                  |
|        |                | 33   | 10     |   |                                                        |
|        | BNE PCOUNT     | 34   | 25     | 4 |                                                        |
|        |                | 35   | FC     |   | 32 <sub>16</sub> - 36 <sub>16</sub> = FC <sub>16</sub> |
|        | COMA           | 36   | 0A     | 3 |                                                        |
|        | STA \$FB       | 37   | E1     | 3 | Mata ut Pulsen                                         |
|        |                | 38   | FB     |   | slut                                                   |

b) Pulsen genereras efter  $3+2+3+25*(4+4)+2+3 = 213$  klockcykler, d v s vid  $t = 0,213$  ms.

Den är  $2+3+100*(4+4)+3+3 = 811$  klockcykler, d v s  $0,811$  ms lång.

d) Nej, koden är inte beroende av sin placering i minnet, förutom att den inte får överlappa adressen  $10_{16}$ .

| F.15 | Instruktion | Maskinkod | Effektiv adress  |
|------|-------------|-----------|------------------|
|      | LDA 0,X     | F3,00     | 20 <sub>16</sub> |
|      | LDA 16,X    | F3,10     | 30 <sub>16</sub> |
|      | LDA -16,X   | F3,F0     | 10 <sub>16</sub> |
|      | LDA ,X+     | F5        | 20 <sub>16</sub> |
|      | LDA ,-X     | F8        | 1F <sub>16</sub> |

|      |            |
|------|------------|
| F.16 | LDX #\$30  |
| LOOP | LDA \$FB   |
|      | ANDA #\$0F |
|      | LDA A,X    |
|      | STA \$FB   |
|      | BRA LOOP   |

| F.17 | Instruktion | A  | X (efter instruktionen) |
|------|-------------|----|-------------------------|
|      | LDX #\$12   | -  | 12                      |
|      | LDA \$3E    | 3E | 12                      |
|      | LDA 0,X     | 12 | 12                      |
|      | LDX 18,X    | 12 | 24                      |
|      | LDA \$E0,X  | 04 | 24                      |
|      | LDX 4,X     | 04 | 28                      |
|      | LDA ,X+     | 28 | 29                      |
|      | LDA 0,X     | 29 | 29                      |
|      | LDA ,-X     | 28 | 28                      |

|         |            |                     |
|---------|------------|---------------------|
| F.18    | PSHA       |                     |
| MASK    | PSHC       |                     |
|         | PSHX       |                     |
|         | LDX #\$40  | startadress         |
| LOOP    | LDA 0,X    | hämta byten         |
|         | ANDA #\$7F | nollställ ms bit    |
|         | STA ,X+    | skriv tillbaks byte |
|         | CMPX #\$61 |                     |
|         | BNE LOOP   | om inte: fortsätt   |
| MASKBIT | PULX       |                     |
|         | PULC       |                     |
|         | PULA       |                     |
|         | RTS        |                     |

|      |            |                             |
|------|------------|-----------------------------|
| F.19 | LDX #\$18  | startadress                 |
|      | LDA #4     | antal - 1                   |
|      | STA COUNT  | räknare för antal bytes     |
|      | LDA 0,X    | första värdet               |
|      | STX PLACE  | adress+1 till första värdet |
|      | STA VALUE  | första värdet är nu max     |
| LOOP | LDA ,+X    | hämta nästa värde           |
|      | CMPA VALUE | nytt maxvärde?              |
|      | BLS LESS   | nej                         |
|      | STA VALUE  | ja, lagra nytt värde        |
|      | STX PLACE  | adress+1 till första värdet |
| LESS | DEC COUNT  | antal bytes kvar - 1        |
|      | BNE LOOP   | färdigt? nej                |

```

F.20      LDSP    #$F0
            LDA     #$06
            PSHA
            LDA     #$35
            PSHA
            LDA     #$28
            PSHA
                                     (SP efter = ED16)

```

```

F.21      PSHA
            PULX

```

```

F.23      LDA     $FB
            TSTA
            BEQ     LEV1
            CMPA    #$18
            BEQ     LEV2
            JSR     FALL3
            BRA     END
LEV1        JSR     FALL1
            BRA     END
LEV2        JSR     FALL2
            END

```

```

F.24      ADDAY   PSHY
            ADDA    0, SP
            BCC     TST0
            LDY     #1
TST0        CMPY    #$00
            BNE     EXIT
            TSTA
EXIT        LEASP   1, SP
            RTS

```

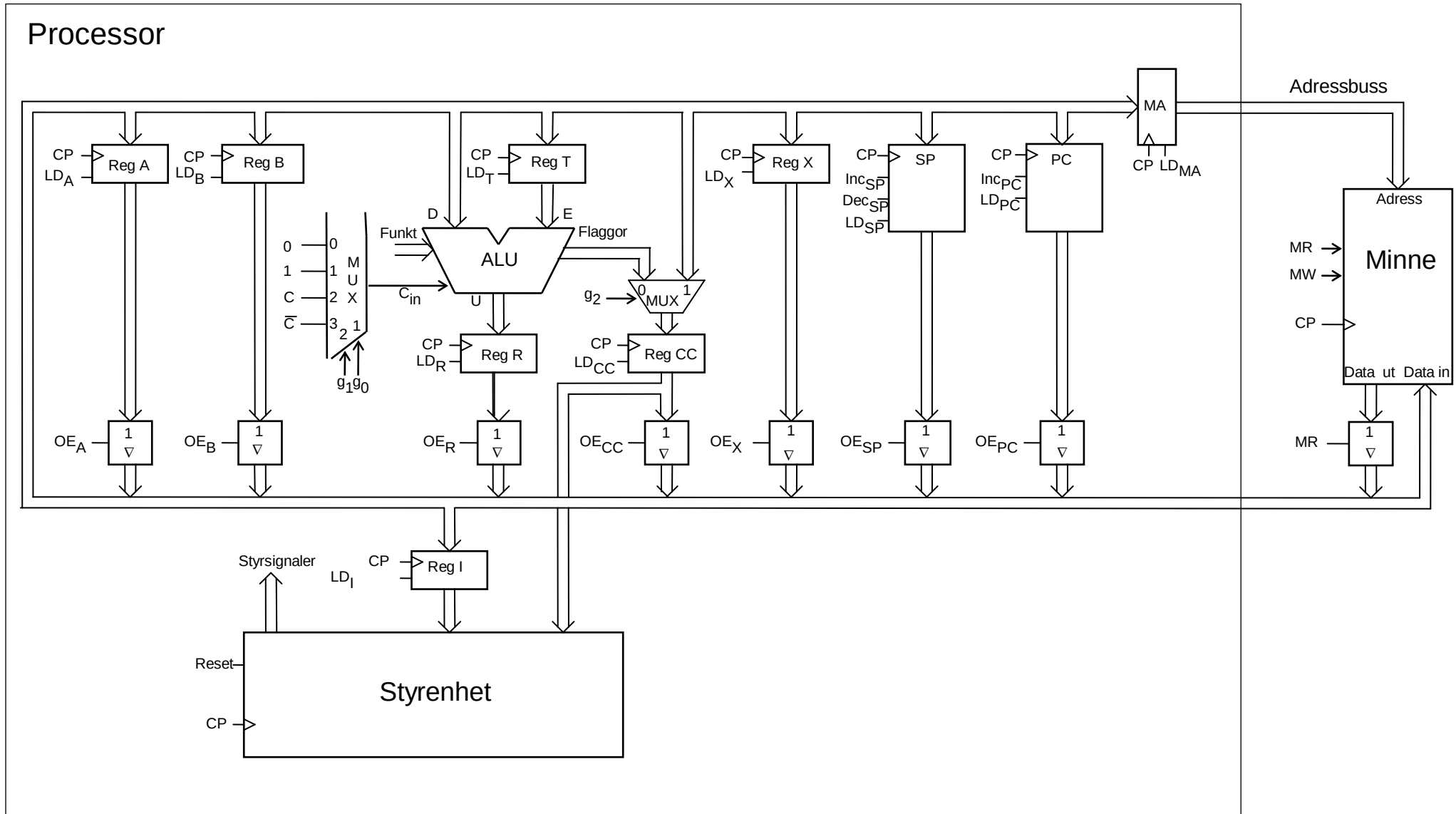
```

F.26      WRIMEM  PSHA
            PSHC
            PSHX
            LDX     #$00
            CLRA
LOOP        STA     , X+
            INCA
            CMPX    #$80
            BNE     LOOP
            PULX
            PULC
            PULA
            RTS

```

**F.27**    Obalanserad stack, fel återhopsadress

## Ext-8



Datorn FLEX sammansatt av dataväg, styrenhet och minnesenhet.