KOMPENDIUM

GRUNDLÄGGANDE DATORTEKNIK FÖR HÖGSKOLANS INGENJÖRSUTBILDNINGAR

ROGER JOHANSSON

CHALMERS TEKNISKA HÖGSKOLA, 2013

Innehåll

1 Introduktion till datortekniken

Med ordet "dator" (från början benämningen *datamaskin*, härlett från latinets "datum"), menar vi i första hand en beräkningsmaskin, dvs. en apparat som utför räkneoperationer (aritmetiska operationer). Kanske är den engelska benämningen "computer" (ungefär *beräknare*) en bättre beteckning. Man kan säga att datorn föddes ur ett behov att utföra stora och komplicerade beräkningar snabbt och med stor noggrannhet. Dagens datorer har utvecklats till att vara en komponent i system som tillhandahåller funktioner som går långt utöver de jämförelsevis enkla beräkningsuppgifter de första datorerna var konstruerade att utföra. Ny *funktionalitet* tillhandahållen av datorsystem har gett oss den *tredje industriella revolutionen*. Datorn, som programmerbar maskin, har påverkat oss och våra levnadsbetingelser ännu mycket mer än någon tidigare teknisk innovation i historien.

Ett datorsystem består av *maskinvara* och *programvara*. I bland kallas det också för *hårdvara* respektive *mjukvara* (*hardware*, *software*). Uppdelningen är strukturell men inte funktionell, dvs., vi kan behandla och studera hårdvaran respektive mjukvaran för sig men det är alltid kombinationen av dessa som skapar möjlighet att tillhandahålla funktionalitet.

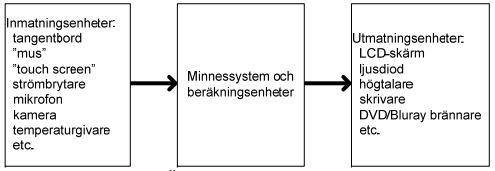
I detta läromedel studerar vi hur en dator är uppbyggd och hur den fungerar. Genom att stegvis bygga en enkel dator beskriver och förklarar vi grunderna för detta tämligen komplexa system. Samtidigt ges insikter i hur datorn, som programmerbar maskin, blir användbar först då den förses med ett *program*, dvs. en sekvens instruktioner som realiserar en lösningsmetod (*algoritm*). Detta utgör basen för dig som ämnar bli en god programmerare eftersom du kommer att bättre kunna inse såväl möjligheter som begränsningar dikterade av datorsystemets hårdvara. Du kan därefter själv tränga vidare in i teknikområdet och fortsätta med fördjupade studier inom datorarkitektur och elektronikkonstruktion om du så vill.

Datorsystem kan grovt indelas i tre olika kategorier baserat på användningsområde:

- *Persondatorer*, enkla stationära och bärbara datorer för generell användning men även specialicerade handhållna enheter, som "smarta" mobiltelefoner och "surfplattor".
- *Serversystem*, centralt baserade system oftast med mycket stor kommunikationsbandbredd för att möjliggöra samtidiga uppkopplingar mot många andra enheter, företrädesvis persondatorer. Serversystem används exempelvis för att tillhandahålla funktioner som säkerhetskopiering (cloud, drop-box) men även e-post och internet.
- *Styrsystem*, system för övervakning och styrning, den i särklass vanligast förekommande kategorin. Vi möter dessa system, oftast indirekt, hela tiden i vardagen, allt från enkla automatiskt öppningsbara dörrar till avancerade system för flygplan.

FIGUR 1.1 STATIONÄR, BÄRBAR OCH HANDHÅLLEN PERSONDATOR (MOBILTELEFON)

En övergripande beskrivning av ett datorsystem får man genom att detaljera de delsystem det är uppbyggt av. I figur 1.2 visas en konventionell blockbeskrivning av datorsystemets delar baserad på delsystemens funktioner. I figuren ges bland annat exempel på en rad olika typer av inmatningsenheter och utmatningsenheter. Pilarna illustrerar informationsflöden och huvudsakliga signalriktningar.

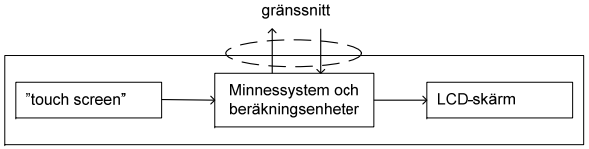


FIGUR 1.2 ÖVERGRIPANDE BESKRIVNING

Exempel 1.1 Signal kontra information

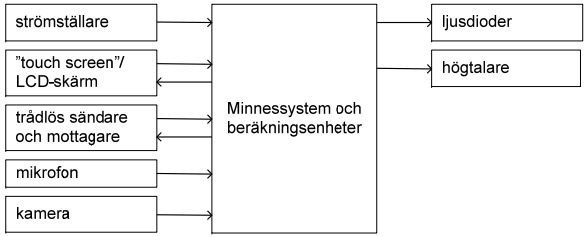
Tre olika signaler representerande färgerna rött, blått och grönt kan användas för att skapa miljontals olika nyanser. Proportionerna av rött blått och grönt (intensiteter) utgörs av signaler medan den resulterande nyansen, presenterad på en bildskärm, utgör information.

Flera in- och utmatningsfunktioner kan också kombineras i en enda enhet, exempelvis utgör ju en så kallad pekplatta såväl en utmatningsenhet i form av *LCD-display* som en inmatningsenhet i form av en *touchscreen*. Det är vanligt att sådana delsystem i själva verket kan beskrivas som ytterligare ett datorsystem. Oftast behöver man inte känna till detaljer om hur delsystem som in- och utmatningsenheter är uppbygga och man bestämmer i stället ett *gränssnitt* (*interface*) där såväl signalflöden som informationsflöden är definierade, se som exempel figur 1.3.

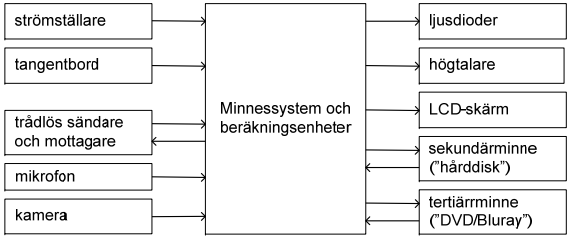


FIGUR 1.3 FUNKTIONEN "PEKPLATTA" AVGRÄNSAD SOM ETT DELSYSTEM

Strukturellt är de flesta datorsystem uppbyggda på ett mycket likartat sätt. Skillnader ligger huvudsakligen i vilka typer av in- och utmatningsenheter (*kringenheter*) de tillhandahåller samt deras prestanda i form av beräkningskapacitet och kommunikationshastigheter. En blockschematisk bild av en så kallad "smartphone" visas i figur 1.4, en snarlik bild i figur 1.5 illustrerar de väsentliga delsystemen i en bärbar persondator.



FIGUR 1.4 BLOCKSCHEMATISK BESKRIVNING AV EN MOBILTELEFON



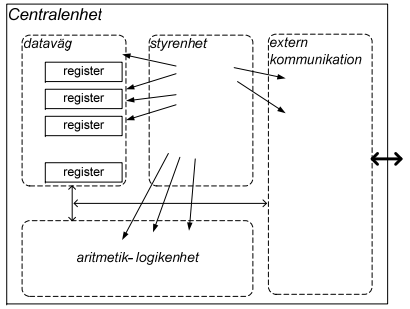
FIGUR 1.5 BLOCKSCHEMATISK BESKRIVNING AV BÄRBAR PERSONDATOR ("LAPTOP")

Datorsystemets centrala delar, dvs. minnessystem och beräkningsenhet, förekommer i alla typer av datorer. I detta läromedel har vi därför valt att inledningsvis tona ned behandlingen av gränssnitt och kringenheter och förstå datorsystemets funktion och arbetssätt genom att studera ett enkelt styrsystem med speciell inriktning mot beräkningsenheten och de nödvändiga minnessystemen. Vi kommer därefter att översiktligt för behandla principer för bättre prestanda och mer komplicerade kringenheter.

Det finns många typer av minnessystem för olika ändamål. Minnessystem indelas hierarkiskt baserat på åtkomsttid, dvs. ju snabbare man kan skriva/läsa data till/från ett minne, desto högre upp i minneshierarkin hamnar denna minnestyp. Den snabbaste minnestypen kallar vi *register*, däreftet följer *primärminnen*, *skundärminnen* och *tertiära minnen*. Det finns även olika typer av specialiserade beräkningsenheter, exempelvis speciellt utformade för att utföra bearbetningssekvenser för signalbehandling som att översätta ljud eller bildinformation "i realtid" till digital information ("signalprocessorer"). En annan typ av beräkningsenhet är specialiserad för beräkningar som inbegriper talvärden inom mycket stora talområden, så kallade "flyttal" (*floating point numbers*) osv. Den i särklass vanligaste beräkningsenheten i en dator är dock den som utför aritmetik- och logikoperationer på så kallade "heltal" och benämns *ALU* (*Arithmetic/Logic Unit*).

Den enhet som innehåller beräkningsenheter och ett mindre minnessystem kallas helt enkelt *centralenhet* (*Central Processing Unit*). Se figur 1.6 som beskriver en enkel centralenhet. En komplett centralenhet kan nu beskrivas av fyra delsystem:

- ALU, beräkningsenhet som kan utföra en rad olika heltalsoperationer
- *datavägen*, sammanfattande namn på minneselement som används tillsammans med beräkningsenheten för att, ofta under kort tid, lagra delresultat vid beräkningar. Datavägen är därför sammankopplad med ALU:n.
- *styrenhet*, vanligtvis den i särklass mest komplexa komponenten hos centralenheten. I styrenheten genereras signaler för att överföra minnesinnehåll i datavägen mellan register och ALU. Styrenheten genererar också signaler som bestämmer vilken operation ALU:n ska utföra. Styrenheten, i sin tur, genererar dessa styrsignaler baserat på ett *program*, med så kallade *maskininstruktioner*, speciellt utformade för den aktuella centralenheten.
- *extern kommunikation*, denna del av centralenheten används för kontrollera datalagring i minnet men även styra in- och utmatning i datorsystemet.



FIGUR 1.6 GRUNDLÄGGANDE FUNKTIONSBLOCK I EN CENTRALENHET

1.1 Beskrivningsnivåer

Kretsnivå

Ett datorsystem är i grunden uppbyggt av några få olika, tämligen enkla, kretselement som *transistorer*, *dioder*, *resistorer* och *kondensatorer*. Den lägsta nivåns beskrivning, *kretsnivån*, utgörs därför av kretsscheman som visar hur sådana komponenter kopplas samman för att bygga upp datorsystemet. Kunskaper i elekronikkonstruktion är vitala då det gäller att förstå olika prestandaegenskaper hos ett datorsystem, som exempelvis arbetstakt och energiförbrukning.

Logiknivå

På *logiknivån*, används kretselementen för att sätta samman *logikfunktioner* i form av komponenter som kallas *logikgrindar* och i grundläggande minnesfunktioner som komponenter vi kallar *vippor*. På logiknivå bygger vi *kombinatoriska nät* och *sekvensnät* med bland annat mer sammansatta och speciella funktioner som *register*, *ALU* och kretsar för att koppla signaler mellan dessa funktionsenheter.

Register-transfer-nivå

Beskrivningen av hur en enskild maskininstruktion utförs i centralenheten lämpar sig väl för *register-transfer-nivån*. På denna nivå beskrivs centralenheten med mer komplexa komponenter uppbyggda kring kombinatoriska nät och sekvensnät. Bilden av centralenheten börjar här ta form med komponenter som register, minne ALU och kopplingsvägar (bussar) mellan dessa enheter.

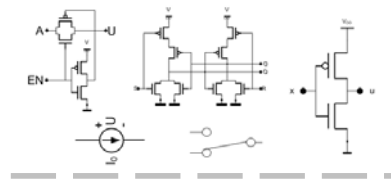
Maskinkod och assemblerkod

Maskinkod och assemblerkod är i allt väsentligt samma sak. Varje *assemblerinstruktion* motsvaras vanligtvis av exakt en *maskininstruktion* (även om undantag kan förekomma). I maskinkoden har en lång rad maskininstruktioner satts samman i sekvenser som, via styrenheten, bildar de signaler som krävs för att åstadkomma operationer i ALU:n och datakopiering i datavägen. Maskinkoden, som bara består av nollor och ettor, är praktiskt taget obegriplig för oss människor och vi använder därför assemblerspråket där vi gett varje maskininstruktion ett kort namn som direkt ger en association till vilken maskininstruktion det handlar om. På detta sätt blir assemblerprogrammet läsbart för oss. Ett assemblerprogram är anpassat för en speciell familj av centralenhet och kan normalt inte utnyttjas för en dator med en annan typ av centralenhet.

Specifikation och programkod

Mjukvaran i ett datorsystem består av ett program som på ett detaljerat sätt styr datorn att utföra dess uppgift. När man konstruerar ett program för en dator börjar man med att beskriva datorsystemets övergripande funktion i form av en *specifikation* av uppgiften och hur den kan lösas. Nästa steg blir att i detalj beskriva förloppet i ett programspråk. Används ett standardiserat högnivåspråk som exempelvis C, C++ eller Java kan programmet senare användas på en mängd olika typer av datorer, vi säger då att programmet är *portabelt*. Det krävs då att man har tillgång till en *kompilator* som översätter programmet till ett assemblerprogram för just den typ av centralenhet som används i datorn.

Beskrivningsnivåerna kretsnivå, programkod och specifikation faller vid sidan av detta läromedels huvudinriktning. Vi kommer därför bara att behandla dessa kortfattat, vilket vi huvudsakligen gör i återstoden av detta inledningskapitel. Därefter ägnar vi oss framför allt åt de fyra mellersta beskrivningsnivåerna; Logiknivå, register-transfer-nivå, maskinkod och assemblerkod.

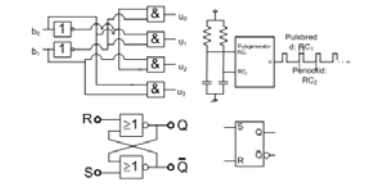


Kretsnivå

$$I_{DS} = I_{DSS}\left(1 - \frac{V_{GS}}{V_P}\right)^2$$

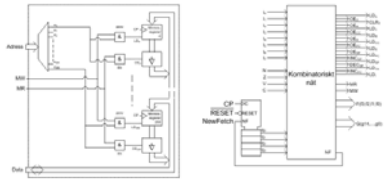
$$\Psi \equiv \oint D_n dA$$

.....



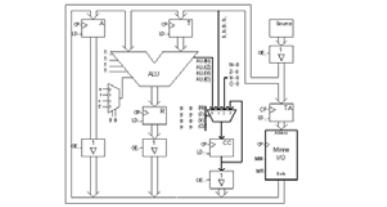
Logiknivå

```
CP1: OEPC=1, LDAdr =1, IncPC=1
CP2: MR=1, LDT=1
CP3: OEDR1=1, LDR, f3=1 f1=1
CP4: OER=1, LDDR1=1
...
```



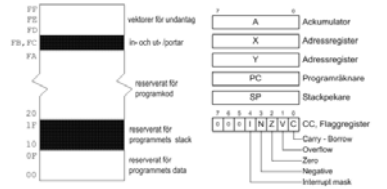
Register-transfer-nivå

```
PC → AdrReg, PC+1 → PC,
Mem(AdrReg) → TempReg
TempReg + DataReg1 → ResReg
ResReg → DataReg1
.....
```



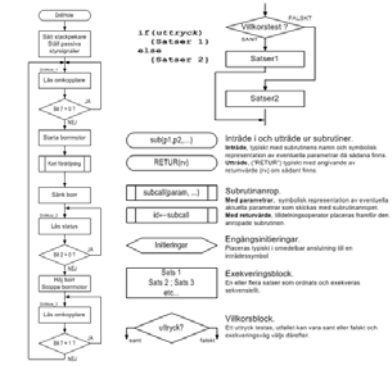
Maskinkod

```
00010011 11111101
00001011 11111110
00011001 00011000
01001010 00011000
01011110 11111000
.....
```



Assemblerkod

```
          STA     BorrStyr
L20:      LDA     BorrStat
          CMPA    #BorrNere
          BNE     L20
          ---
```



Programkod

```
while ( rpm < 1000 )
      spinup();
DrillDown();
   ....
```


Specifikation

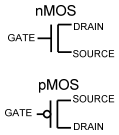
”Automatiskt styrd borrmaskin”

- Positionera borr
- Starta borr
- Borra genom arbetsstycke
- ...

1.2 Kretsnivån, transistorn som omkopplare

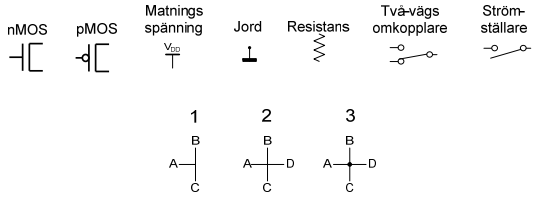
En speciell typ av transistorer tillverkade i MOS-teknik (*Metal Oxide Silicon*) är speciellt väl lämpad för användning inom digital elektronik. För många ändamål kan man helt enkelt modellera MOS-transistorn som en enkel omkopplare.

Det finns två varianter, nMOS och pMOS, där skillnaden förenklat kan sägas vara att nMOS leder då GATE potentialen är hög medan pMOS leder då GATE-potentialen är låg. Se figur 1.7 i den förenklade modellen har MOS-transistorn tre kontaktpunkter, GATE (styre), SOURCE (källa) och DRAIN (avlopp). Styret bestämmer transistorns funktion, dvs. leda eller spärra, SOURCE syftar på laddningsbärarna som ska ledas vidare och DRAIN syftar på destinationen för dessa laddningsbärare. För nMOS transistorer är laddningsbärarna elektroner, medan pMOS transistorns laddningsbärare är det motsatta, dvs. ”hål”.



FIGUR 1.7 SYMBOLER FÖR MOS-TRANSISTOR

I den övre raden i figur 1.8 visas några symboler vi använder för att illustrera grundläggande principer bakom digitala kretsar. Under dessa symboler visas tre olika kopplingar som också förekommer. Koppling 1 är en trevägskorsning där punkterna A,B och C har kontakt med varandra. Koppling 2 illustrerar en korsning där punkterna A,D respektive B,C har kontakt. Det finns alltså *ingen förbindning* mellan exempelvis A,B (eller A,C). Slutligen i koppling 3 visas en fyrvägsanslutning där alla punkter A,B,C och D är förbundna med varandra.

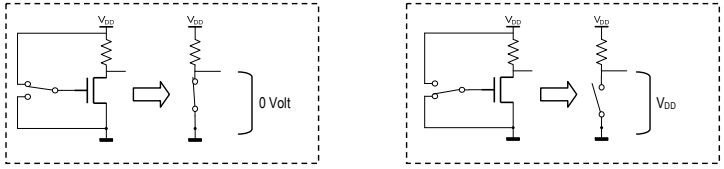


FIGUR 1.8 SYMBOLER FÖR ELEKTRONIKSCHEMAN

1.2.1 nMOS och pMOS-teknik

nMOS-transistorn betraktad som en omkopplare illustreras i figur 1.9. Till vänster i figuren visas hur en ledande transistor (GATE-nivån motsvarar V_{DD} potential) kan betraktas som en sluten omkopplare, utsignalen hamnar då på jordpotential.

Till höger i figuren visas hur en transistor som spärrar (GATE-nivån motsvarar GND-potential) motsvaras av en öppen omkopplare, utgången hamnar nu på V_{DD} potential.



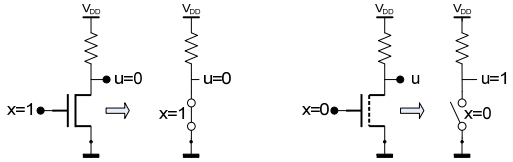
FIGUR 1.9 NMOS-TRANSISTOR SOM OMKOPPLARE

I digitala system föredrar vi att arbeta med *logikvärden*, snarare än spänningar och strömmar. Logikvärdet kan vara *sant*, indikerat av talvärdet 1, respektive *falskt*, indikerat av talvärdet 0. Sambandet mellan elektrisk spänning och logikvärden ges av konvention, man måste alltså i sammanhanget ange vilket samband som används och följande samband är möjliga:

- *Aktiv hög logiknivå* innebär att spänningen V_{DD} motsvarar logikvärdet *sant*, medan 0 Volt motsvarar logikvärdet *falskt*.
- *Aktiv låg logiknivå* innebär det omvända, dvs V_{DD} motsvarar logikvärdet *falskt* medan 0 Volt motsvarar logikvärdet *sant*.

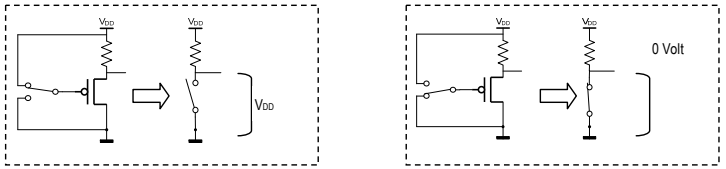
I detta studiematerial förutsätter vi alltid *aktiv hög logiknivå* om inget att sägs uttryckligt. För enkelhets skull använder vi dessutom hellre talvärdena 0 och 1, snarare än logikvärdena *falskt* och *sant*.

En transistorkoppling kan illustreras med ekvivalenta scheman där vi bytt ut transistorn mot en omkopplarfunktion. Till vänster i figur 1.10 illustreras hur kretsens ingång x, påförs talvärdet 1, i omkopplarschemat ser vi hur detta resulterar i att utgången u, kopplas direkt till jordpotential och har därmed talvärdet 0. På motsvarande sätt, till höger i figuren, då vi påför talvärdet 0 på ingången, kopplas utgången, via ett belastningsmotstånd till V_{DD}, dvs. talvärdet 1.



FIGUR 1.10 EKVIVALENTA SCHEMAN

Resonemanget kring nMOS-transistorn är liknande för pMOS-transistorn, se figur 1.11, polariteterna för en ledande funktion är dock här omvänd.

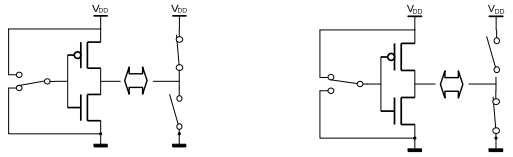


FIGUR 1.11 PMOS-TRANSISTOR SOM OMKOPPLARE

En stor nackdel med såväl nMOS som pMOS är att kretsar med ledande transistorer drar ström även i stationärtillståndet. Framför allt för att komma tillrätta med den statiska strömförbrukningen används i stället nMOS/pMOS-transistorer i par, vi kallar detta CMOS-teknik.

1.2.2 CMOS-teknik

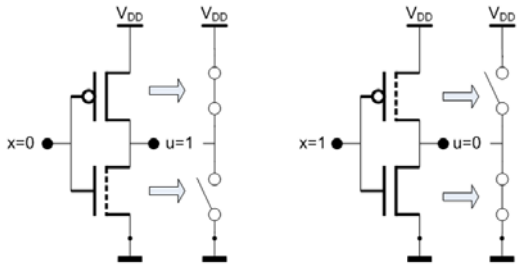
Digitala kretsfamiljer är ofta uppbyggda kring komplementär MOS-teknik (*Complementary MOS*). Fördelen med detta är att effektförbrukningen kan minskas genom att ström bara flyter genom kretsarna vid omslag. Följande enkla koppling (figur 1.12) illustrerar detta.



FIGUR 1.12 CMOS-KOPPLING

Eftersom insignalen är kopplad till båda transistorernas styren har vi alltså alltid stationärtillstånden att en transistor leder medan den andra transistorn spärrar. Utgångens nivå (spänningspotential) behålls i båda stationärtillstånden och laddningsförflyttningar ("strömförbrukning") sker enbart då insignalen ändras. Vi kan redan nu inse att digitala systems prestanda, dvs. beräkningskapacitet bland annat avgörs av hur ofta (snabbt) vi överför signalerna mellan kopplingar i systemet. Av detta följer direkt att ju snabbare system vi kräver, desto mer energi (elektrisk ström) måste vi kunna tillhandahålla. Detta som en konsekvens av att den högre arbetstakten (också kallad klockfrekvens) innebär fler omslag per tidsenhet.

Det ekvivalenta schemat (figur 1.13) kan användas för att analysera kopplingens överförings-funktion, dvs. u = f(x).



FIGUR 1.13 REALISERING AV LOGIKFUNKTIONEN ICKE

Eftersom signaler i digitala system endast kan ha två stationärtillstånd (V_{DD} eller GND), blir analysen enkel. Hela funktionen illustreras i en funktionstabell, figur 1.14 där de möjliga logikvärdena som kan påföras ingången listas tillsammans med de resulterande funktionsvärdena, dvs. logikvärdet på utgången. Denna krets realiserar exempelvis logikfunktionen ICKE (NOT).

Funktionstabell	
x	u = f(x)
0	1
1	0

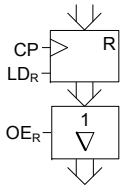
FIGUR 1.14 FUNKTIONSTABELL, LOGISK FUNKTION NOT

7 Dataväg, ALU och minne

I detta kapitel ska vi detaljstudera komponenter som ingår i datorns centralenhet. Vi börjar beskriva en liten centralenhet som vi kallar FLIS (Flexible Instrucction Set)- processorn. Vi kommer att beskriva de enklaste formerna av minne. Vi introducerar informellt RTN (*Register Transfer Notation*) ett enkelt skrivsätt som vi sedan kommer att använda för att beskriva de operationer som kan utföras i datavägen av centralenheten.

7.1 8-bitars register med LOAD och OUTPUT ENABLE

Den första komponenten vi detaljstuderar är ett 8-bitars minneselement, eller mera allmänt, *register* som används för att lagra data.



Med register R använder vi tre beteckningar:

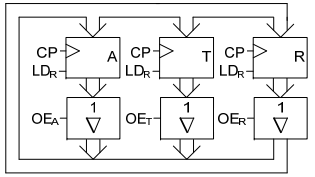
CP: *Clock Pulse*, kretsens klockingång.
LD_R: *Load Enable* R, styrsignal för att kunna ladda register R
OE_R: *Output Enable* R, styrsignal för att överföra innehållet i R till utgången.

FIGUR 7.1 ENKELT REGISTER SOM MINNESELEMENT

Signalen OE_R påverkar datavägen asynkront, dvs i samma ögonblick som den aktiveras kopplas också innehållet fram till registrets utgång, vi säger att registret "driver" bussen. Signalen LD_R avgör om register R ska laddas vid nästa klockpuls. Eftersom en ändring blir märkbar först då klockpulsen kommer (positiv flank) har LD_R ett synkront beteende.

Vi kan kombinera flera register, där vi kopplar samman in- och utgångarna via en buss, dvs. ett antal signalledningar för att sedan med hjälp av de olika styrsignalerna "dirigera" hur olika registerinnehåll kopieras och skenbart "flyttas" mellan de olika registren. I det här sammanhanget kallar vi ofta bussen för *dataväg*, eftersom det är denna del som används för transporten eller överföringen av registerinnehållen.

I kombination med datavägen har vi styrsignalerna (LD,OE) som de nödvändiga dirigenterna. De är alltså inte del av datavägen men nog så väsentliga. Styrsignaler måste genereras på något sätt och sekvensieringen (ordningsföljden) av styrsignalerna är helt och hållet beroende på den uppgift som ska utföras. Vi återkommer till olika metoder för att automatiskt generera sekvenser av styrsignaler i nästa kapitel där vi detaljstuderar centralenhetens styrenhet. I detta kapitel nöjer vi oss med att se hur vi kan påverka olika element i datavägen genom att generera specifika sekvenser av styrsignaler.



FIGUR 7.2 FLERA REGISTER KOPPLAS SAMMAN TILL EN "DATAVÄG"

Betrakta Figur 7.2. Data kan kopieras (överföras) mellan olika register i datavägen genom att

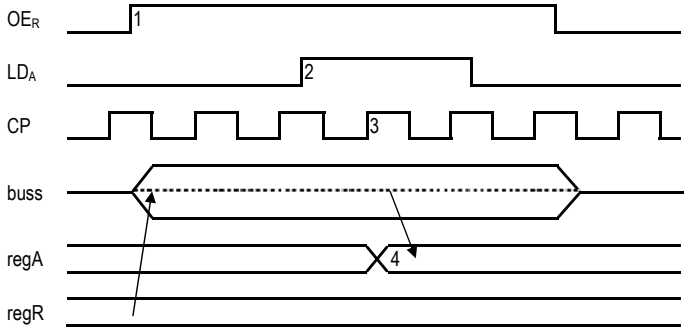
1. någon av OE-signalerna aktiveras
2. en eller flera LD-signaler aktiveras
3. en klockpuls (CP) ges till samtliga register

vid klockpulsen kommer innehållet i det register vars OE-signal är aktiverad att kopieras till samtliga register vars LD-signal är aktiverad.

Notera att endast en OE-signal får vara aktiv under en klockpuls. Vi kan däremot ha flera aktiva LD-signaler under samma klockpuls.

Exempel 7.1 Visa hur kopiering av innehållet i R till A sker i ett digitalt system (Figur 7.2)

Vi kopplar innehållet i register R till datavägen genom att aktivera signalen OE_R.(1)
Observera ”bussen” i figuren, signalninåerna går nu från att vara odefinierade (”three-state”) till väldefinierade nivåer (noll eller ett).
Vi dirigerar innehållet på datavägens buss till register A genom att aktivera signalen LD_A. Detta sker vid (2), ingången är klockad, det händer därför ingenting i datavägen före nästa positiva klockflank (3). Vid flanken hos klocksignalen CP, sker dataöverföringen, detta indikeras i figuren som att ”regA” får ett nytt värde (4).



7.2 Beskrivning med RTN

RTN (Register Transfer Notation) erbjuder ett förenklat skrivsätt för att specificera olika operationer där register ingår. I skrivsättet används enkla symboler och för att speciellt ange att RTN-notation avses använder vi ett avvikande typsnitt. För att ange överföring av data kan *piloperatorer* användas, här kommer några exempel:

A → T Innehållet i register A överförs till register T

RTN-operationer separerade med kommatecken betyder att operationerna utförs i denna ordning.

A → T, T → R Innehållet i register A överförs till register T, därefter överförs innehållet i register T till register R.

RTN-operationer separerade med semikolon betyder att dessas inbördes ordning är oväsentlig, operationerna kan i princip utföras samtidigt (parallellt). Flera register kan alltså tilldelas nya värden samtidigt, det kan dock bara finnas ett källregister:

A → T; A → R Innehållet i register A överförs till såväl register T som register R.

Beskrivning av hur registerinnehåll skiftas görs med en dubbelriktad piloperator:

X ↔ Y Innehållet i register X överförs till register Y samtidigt som innehållet i register Y överförs till register X.

Om vi inskränker oss till den enkla organisationen av dataväg och register i Figur 7.2, kan inte den sistnämnda operationen (skifte av registerinnehåll) utföras i ett enda parallellt steg. I sådana fall tvingas vi skapa en sekvens av operationer som i tur och ordning utför en rad deloperationer som tillsammans åstadkommer den dubbelriktade operationen, se exempel 7.2 nedan. Ofta är det på precis detta sätt vi använder RTN, dvs. att beskriva längre komplexa operationer med sekvenser av enklare operationer. På detta sätt behåller vi detaljrikedomen (entydigheten) samtidigt som vi kan arbeta med en mer överskådlig beskrivning

Exempel 7.2 Ömsesidigt byte av registerinnehåll

Åskådliggör operationen A↔R, så som den utförs i datavägen enligt Figur 7.2.
Lösning;
Eftersom datavägen inte tillåter samtidig överföring, dvs. vi kan bara överföra ett registerinnehåll åt gången, så måste det göras i två steg och vi måste dessutom använda ett tredje register (T) för mellanlagring:
A → T, R → A, T → R

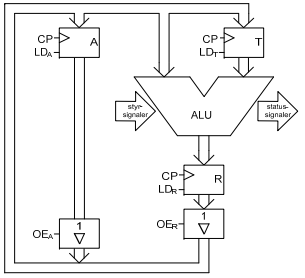
Utöver piloperatorerna används också alla de vanliga operatorsymbolerna så som likhet (=), olikhet (!=), relationsoperatorer (<,>), aritmetikoperatorer (+, - , × , /) och logikoperatorer (', &, |, ~) i RTN-beskrivningar, se Tabell 7.1. Vi kommer att introducera ytterligare några operatorer allt eftersom de behövs i den kommande framställningen.

operator	operation
n	Konstant uttryckt i talbas 10
N_r	Konstanten N uttryckt i talbasen r.
→	Kopiering
+	Addition
-	Subtraktion
∧	Logiskt ”OCH” (AND)
∨	Logiskt ”ELLER” (OR)
⊕	Logiskt ”EXKLUSIVT ELLER” (XOR)
Opr<<1 (*d*)	”Opr” skiftas vänster ett steg. Biten *d* skiftas in i den minst signifikanta positionen.
(*d*) 1>>Opr	”Opr” skiftas höger. Biten *d* skiftas in i den mest signifikanta positionen.
Opr'	Bitvis komplementering av operanden ”Opr”
=	likhet
!=	olikhet
>	större än
<	mindre än
>=	större än eller likhet
<=	mindre än eller likhet

TABELL 7.1 GRUNDLÄGGANDE OPERATIONER I RTN (REGISTER TRANSFER NOTATION)

7.3 ALU-funktioner

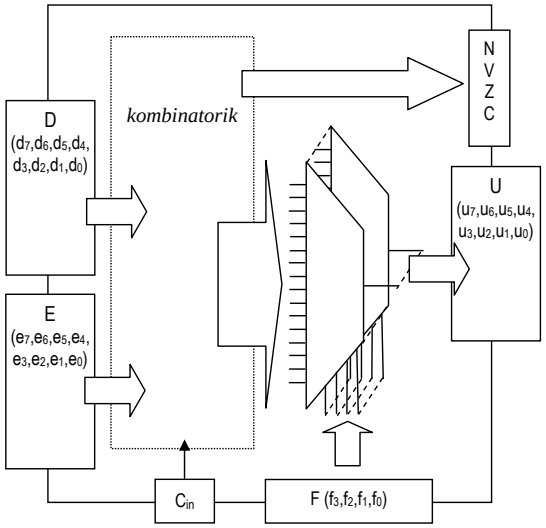
Vi har sett hur man med en enkla operationer kan kopiera data mellan olika register i datavägen. Nästa steg blir att på ett förutbestämt sätt förändra (manipulera) data under dataöverföringen. Vi behöver exempelvis kunna utföra åtminstone de enkla aritmetikoperationerna addition och subtraktion med registerinnehåll, vi behöver också kunna bilda komplement och utföra logikoperationer på registerinnehållen etc. Mer generellt måste vi kunna utföra någon förutbestämd operation på innehållet i något register och därefter återföra resultatet till ett godtyckligt (eventuellt samma) register. Vi inför därför en ny komponent i datavägen med sådana egenskaper (se Figur 7.3). Den nya komponenten kallar vi ”aritmetik- och logik- enhet”, på engelska, *Arithmetic/Logical Unit*, eller helt enkelt ”ALU”. Vår ALU är ett kombinatoriskt nät uppbyggt av de betydligt enklare komponenter vi tidigare behandlat.



FIGUR 7.3 DATAVÄG MED ALU

Vi ska ny beskriva den enkla ALU som vi använder i FLISP. De datamanipulerande enheterna i moderna mikroprocessorer uppvisar som regel betydligt större komplexitet som att hantera speciella datarepresentationer (bitfält, flyttal etc.) och att dessutom utföra flera uppgifter samtidigt med så kallad ”pipeline”-teknik. Vi återkommer till sådana prestandahöjande tekniker. Det är dock viktigt att inse att de konstruktioner och metoder vi beskriver här utgör grunderna och att ingenting principiellt nytt krävs för att konstruera dagens moderna mikroprocessorer.

FLISP’s ALU är ett kombinatoriskt nät som totalt består av 21 insignaler och 12 utsignaler. Det finns två 8-bitars ingångar, **D** och **E,** samt en 8-bitars utgång **U**. Ändringar på ingångarna **D** och **E** påverkar alltså innehållet i **U** momentant, det krävs ingen klockpuls. Se Figur 7.4.



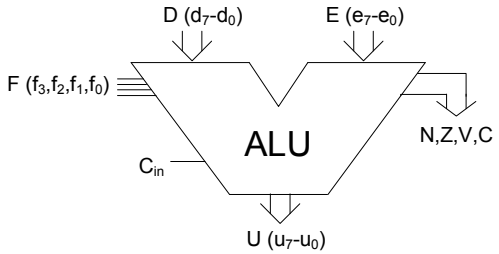
FIGUR 7.4 BLOCKSCHEMATISK BILD AV FLISP’S ALU

ALU’n kan utföra totalt 16 olika operationer och operationen (F) väljs med hjälp av 8 st. 1 av 16-väljare. Väljarens fyra styrsignaler kallar vi f_3, f_2, f_1 och f_0. Vi använder det förenklade skrivsättet:

F(0) för att beteckna $f_3, f_2, f_1, f_0 = 0,0,0,0$
F(1) för att beteckna $f_3, f_2, f_1, f_0 = 0,0,0,1$
...
F(15) för att beteckna $f_3, f_2, f_1, f_0 = 1,1,1,1$

En sista insignal C_{in} (carry in) är kopplad för användning tillsammans med flertalet olika ALU-operationer.

Det finns fyra villkorsindikatorer som utsignaler. Dessa är **N** (“Negative”), **Z** (“Zero”), **V**(“Overflow”) och **C** (“Carry”). Figur 7.5 visar en förenklad bild av ALU’n.



FIGUR 7.5 FÖRENKLAD BILD AV FLISP’S ALU

I Tabell 7.2 nedan sammanfattas ALU’ns operationer.

funktion				operation	utsignaler											
f_3	f_2	f_1	f_0	RTN	u_7	u_6	u_5	u_4	u_3	u_2	u_1	u_0	N	Z	V	C
0	0	0	0	U= 0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	1	U= FD_{16}	1	1	1	1	1	1	0	1	1	0	0	0
0	0	1	0	U= FE_{16}	1	1	1	1	1	1	1	0	1	0	0	0
0	0	1	1	U= FF_{16}	1	1	1	1	1	1	1	1	1	0	0	0
0	1	0	0	U= E	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	u_7	(1)	0	0
0	1	0	1	U= $D_{1k}+C_{in}$									u_7	(1)	(8)	(7)
0	1	1	0	U= D∨E	$d_7 \vee e_7$	$d_6 \vee e_6$	$d_5 \vee e_5$	$d_4 \vee e_4$	$d_3 \vee e_3$	$d_2 \vee e_2$	$d_1 \vee e_1$	$d_0 \vee e_0$	u_7	(1)	0	0
0	1	1	1	U= D∧E	$d_7 \wedge e_7$	$d_6 \wedge e_6$	$d_5 \wedge e_5$	$d_4 \wedge e_4$	$d_3 \wedge e_3$	$d_2 \wedge e_2$	$d_1 \wedge e_1$	$d_0 \wedge e_0$	u_7	(1)	0	0
1	0	0	0	U= D⊕E	$d_7 \oplus e_7$	$d_6 \oplus e_6$	$d_5 \oplus e_5$	$d_4 \oplus e_4$	$d_3 \oplus e_3$	$d_2 \oplus e_2$	$d_1 \oplus e_1$	$d_0 \oplus e_0$	u_7	(1)	0	0
1	0	0	1	U= $D+C_{in}$									u_7	(1)	(2)	(3)
1	0	1	0	U= $D+FF_{16}$									u_7	(1)	(2)	(3)
1	0	1	1	U= $D+E+C_{in}$									u_7	(1)	(2)	(3)
1	1	0	0	U= $D+E_{1k}+C_{in}$									u_7	(1)	(2)	(3)
1	1	0	1	U= D<<1 (C_{in})	d_6	d_5	d_4	d_3	d_2	d_1	d_0	C_{in}	u_7	(1)	0	(4)
1	1	1	0	U= (C_{in}) 1>>D	C_{in}	d_7	d_6	d_5	d_4	d_3	d_2	d_1	u_7	(1)	(6)	(5)
1	1	1	1	U= (d_7)1>>D	d_7	d_7	d_6	d_5	d_4	d_3	d_2	d_1	u_7	(1)	0	(5)

Anm:
D_{1k} = 1-komplement D = D'
E_{1k} = 1-komplement E = E'

(1) $Z = \overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$, dvs. Z=1 då samtliga bitar i register U är 0, Z=0 annars.

(2) $V = (\overline{u_7} \wedge d_7 \wedge e_7) \vee (u_7 \wedge \overline{d_7} \wedge \overline{e_7})$, dvs. V-flaggan sätts enligt reglerna för tvåkomplementsaritmetik.

(3) C = c_8, dvs. carry ut från additionen av de mest signifikanta siffrorna.
(4) C = utskiftad bit, dvs. bit d_7 före vänsterskiftet.
(5) C = utskiftad bit, dvs. bit d_0 före högerskiftet.
(6) V = $C_{in} \oplus d_7$ före skift, dvs. sätts till 1 om skiftet föranleder teckenbyte.
(7) C ettställs om D=0, C nollställs annars.
(8) V ettställs om D=$(10000000)_2$, V nollställs annars.

TABELL 7.2 OPERATIONER HOS FLISP’S ALU

För ALU’ns samtliga operationer gäller alltså att innehållen på ingångarna D, E och C_{in} tillsammans med en operation, bestämd av F, kan påverk utgången U och flaggorna som här betecknas ALU(N,V,Z,C).De möjliga operationerna kan indelas i tre grupper: *binära*, *unära* och *speciella* operationer, och samtliga operationer beskrivs i de följande avsnitten.

7.3.1 Binära operationer

Binära operationer, operationer med två operander utgörs av aritmetik- och logikoperationer. Som konvention placerar vi först den högra operanden i register T och den vänstra operanden kopplas till ingången D innan operationen utförs.

Logikoperationer

Logikoperationerna:

D∨E **F(6), logiskt ELLER**
D∧E **F(7), logiskt OCH**
D⊕E **F(8), logiskt exklusivt ELLER**

utför en bitvis operation mellan innehållen i register D och E. Resultatet placeras i U**.** Flaggsättningen för logikoperationer är meningsfylld för Z och N men saknar tolkning för C och V. En väldefinierad operation kräver entydighet och flaggorna C,V nollställs därför alltid.

ALU(Z) = $\overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
ALU(N) = u_7
ALU(C) = 0
ALU(V) = 0

Aritmetikoperationer

Det finns fyra olika varianter av binära aritmetikoperationer:

$D+E+C_{in}$ F(11), addition med eller utan carry-siffra

Vanligtvis sätts C_{in} till 0, och det är då en helt vanlig addition. Vi kommer längre fram att se hur C_{in} kan väljas som resulterande carry-siffra från en *tidigare* operation. På detta sätt kan tal med större ordbredd än den använda ALU'n också adderas.

$D+E1k+C_{in}$ F(12), subtraktion med eller utan carry (lånesiffra),

Subtraktion utförs genom addition av högeroperandens tvåkomplement, C_{in} sätts då till 1. Även vid subtraktion kan en alternativ användning av C_{in} ge möjlighet att subtrahera tal med större ordbredd än 8 bitar.

För båda dessa operationer sätts flaggorna enligt regler för binär aritmetik:

$ALU(Z) = \overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
$ALU(N) = u_7$
ALU(C) = vid addition: c_8, carry från addition av mest signifikanta siffror
ALU(C) = vid subtraktion: $\overline{c_8}$ =B, dvs. ALU'ns C-flagga representerar här en lånesiffra
$ALU(V) = (\overline{u_7} \wedge d_7 \wedge e_7) \vee (u_7 \wedge \overline{d_7} \wedge \overline{e_7})$

7.3.2 Unära operationer

Det finns en rad *unära* operationer och alla använder D som källa och U som destination.

Kopiering, inkrementering eller dekrementering

$D+C_{in} \rightarrow U$

Funktionen kan användas antingen för att överföra innehållet i D direkt till U då C_{in} sätts till 0, eller att inkrementera innehållet i D om C_{in} sätts till 1:

D $C_{in} = 0$ F(9), kopiering
D+1 $C_{in} = 1$ F(9), inkrementering

För dessa operationer sker flaggsättning enligt de vanliga reglerna för aritmetik, dvs:

$ALU(Z) = \overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
$ALU(N) = u_7$
ALU(C) = c_8, carry från addition av mest signifikanta siffror
$ALU(V) = (\overline{u_7} \wedge d_7 \wedge e_7) \vee (u_7 \wedge \overline{d_7} \wedge \overline{e_7})$

Man kan konstatera att flaggorna C och V saknar tolkning vid kopiering och samtidigt innebär reglerna för flaggsättningen att dessa sätts till 0 i detta fall.

$D+FF_{16}$ F(10), dekrementering

Operationen är ekvivalent med D-1 och flaggsättning sker också här enligt de vanliga reglerna för aritmetik. Observera dock att C här får betydelsen *lånesiffra*.

$ALU(Z) = \overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
$ALU(N) = u_7$
ALU(C) = vid subtraktion: $\overline{c_8}$ =B, dvs. ALU'ns C-flagga representerar här en lånesiffra
$ALU(V) = (\overline{u_7} \wedge d_7 \wedge e_7) \vee (u_7 \wedge \overline{d_7} \wedge \overline{e_7})$

Komplementbildning

ALU'n bildar 1-komplement respektive 2-komplement, beroende på C_{in}, under funktionen F(5):

$D_{1k}+C_{in}$

D_{1k} då $C_{in} = 0$ F(5), bilda 1-komplement

Denna funktion används för att invertera varje enskild bit i D och överföra resultatet till U. Kortare säger vi att 1-komplementet av innehållet i D överförs till U. Alternativa skrivsätt för operationen är $\overline{D}$ eller D'. Operationen är en logikoperation snarare än en aritmetikoperation.

$D_{1k}+1$ då $C_{in} = 1$ F(5), bilda 2-komplement

Alternativa skrivsätt här är D_{2k} eller 0-D , dvs. funktionen används för att negera (bilda 2-komplementet) av talet i D. Detta ska betraktas som en aritmetikoperation.

Flaggsättningen i båda dessa fall följer följande regler:

$ALU(Z) = \overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
$ALU(N) = u_7$
$ALU(C) = d_7 \vee d_6 \vee d_5 \vee d_4 \vee d_3 \vee d_2 \vee d_1 \vee d_0$
$ALU(V) = d_7 \wedge \overline{d_6} \wedge \overline{d_5} \wedge \overline{d_4} \wedge \overline{d_3} \wedge \overline{d_2} \wedge \overline{d_1} \wedge \overline{d_0}$

Vid bitvis invertering (logikoperation) är C och V-flaggorna meningslösa att tolka men för negering (aritmetikoperation) är flaggsättningen intressant:

C sätts till 1 om operanden är skild från 0 ("ELLER"-villkor för alla bitarna), eftersom det enda tal (utan inbyggt tecken) som korrekt kan negeras är just talet 0.

V sätts till 1 om operanden är 10000000_2 dvs -128 då talet tolkas med tecken. Det finns ingen representation på positiva tallinjen för talet +128, största positiva tal är 127, så detta blir den enda spillindikatorn för V vid komplementbildning.

7.3.3 Skiftoperationer

D<<1 (C_{in}) F(13), vänsterskift

Det vänsterskiftade innehållet i D placeras i U. Skiftet är en bitposition och den inskiftade biten kopieras från C_{in}. Den utskiftade biten kopieras till ALU'ns C-flagga. Två vänsterskiftoperationer är möjliga:

D<<1 då $C_{in} = 0$ F(13), logiskt vänsterskift

D<<1 då $C_{in} = C$ F(13), rotation vänster med carry

Flaggsättning:
$ALU(Z) = \overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
$ALU(N) = u_7$
ALU(C) = d_7 , dvs. den bit som skiftas ut och ej ryms i U.
ALU(V) = 0

(C_{in}) 1>>D F(14), högerskift

Det högerskiftade innehållet i D placeras i U. Skiftet är en bitposition och den inskiftade biten kopieras från C_{in}. Den utskiftade biten kopieras till ALU'ns C-flagga.

1>>D då $C_{in} = 0$ F(14), logiskt högerskift

1>>D då $C_{in} = C$ F(14), rotation höger med carry

Flaggsättning:
$ALU(Z) = \overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
$ALU(N) = u_7$
ALU(C) = d_0 dvs. dvs. den bit som skiftas ut och ej ryms i U.
$ALU(V) = d_7 \oplus C_{in}$, dvs. sätts till 1 om operationen ger teckenbyte, 0 annars.

(d_7)1>>D→ U F(15), aritmetiskt högerskift

Det högerskiftade innehållet i **D** placeras i **U**. Skiftet är en bitposition och den inskiftade biten kopieras från d_7. Detta innebär att talets tecken behålls. Aritmetiska högerskift bör inte utföras på tal utan tecken eftersom detta då ändrar talets värde. Den utskiftade biten kopieras till ALU'ns C-flagga.

Flaggsättning:
ALU(Z) = $\overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
ALU(N) = u_7
ALU(C) = d_0, dvs. den bit som skiftas ut och ej ryms i U.
ALU(V) = 0

7.3.4 Speciella operationer

Funktionerna F(0) till F(3) används för att generera speciella konstantvärden (0, FD_{16}, FE_{16} och FF_{16}). De valda konstanterna kan synas godtyckliga men vi kommer att se att de är användbara då vi så småningom ska färdigställa en automatisk styrenhet.

Funktionerna implementeras genom att multiplexerns ingångar antingen kopplas direkt till GND för en logisk nolla, eller till VDD för en logisk etta.

Flaggsättning för respektive operation är självklar för Z och N. Flaggsättningen för C och V har ingen meningsfylld tolkning för operationerna men väldefinierade operationer kräver entydighet och flaggorna nollställs därför alltid.

Funktion	RTN	N	Z	V	C
F(0)	U=0	0	1	0	0
F(1)	U=FD_{16}	1	0	0	0
F(2)	U=FE_{16}	1	0	0	0
F(3)	U=FF_{16}	1	0	0	0

TABELL 7.3 KONSTANTVÄRDEN

Denna funktion används för att kopiera innehållet i **E** till **U**. Flaggorna sätts följaktligen baserat på innehållet i **E**.

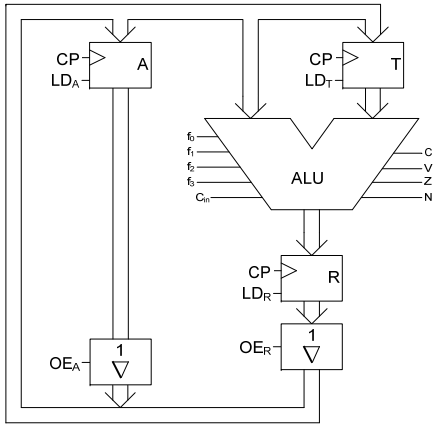
E F(4) kopiering av E

ALU(Z) = $\overline{u_7} \wedge \overline{u_6} \wedge \overline{u_5} \wedge \overline{u_4} \wedge \overline{u_3} \wedge \overline{u_2} \wedge \overline{u_1} \wedge \overline{u_0}$
ALU(N) = u_7
ALU(C) = 0
ALU(V) = 0

7.4 Användning av ALU och register

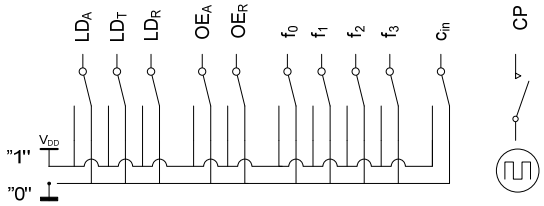
Betrakta Figur 7.6 som illustrerar en enkel dataväg med 8-bitars register och en ALU.

Register T, kopplat direkt till ALU:ns ingång E, är nödvändigt för att vi ska kunna utföra operationer med två operander. Register R som är kopplat till ALU:ns utgång används för att mellanlagra resultat av utförda operationer. Vi har också ett generellt användbart register A, som kan kopplas till såväl ingångar som utgång hos ALU:n.



FIGUR 7.6 ALU MED GENERELLT REGISTER A

Utöver OE- och LD- signaler krävs nu också styrsignalerna till ALU:n, dvs f_0, f_1, f_2, f_3 och C_{in}. Vi föreställer oss nu en panel med strömbrytare, med vars hjälp vi kan ställa in nivåerna, 1 eller 0, för varje enskild styrsignal. Panelen har också en återfjädrande omkopplare, med vars hjälp vi kan skapa en klockpuls till registren i datavägen, se Figur 7.7.



FIGUR 7.7 PANEL MED STRÖMBRYTARE FÖR STYRSIGNALER

I de följande avsnitten visar vi hur sammansatta operationer kan utföras genom att sekvenser av styrsignaler aktiveras i datavägen. Vi passar samtidigt på att introducera en rad operationer som utgör exempel på vanliga mikroprocessoroperationer.

Sekvenserna utförs genom att vi först ställer strömbrytarna i lämpliga positioner och sedan ger en klockpuls, detta kallas också för en *klockcykel*. Olika operationer kan komma att kräva antal klockcykler beroende på hur komplexa operationerna är att utföra.

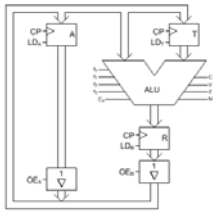
7.4.1 Unära operationer

Operationer med en operand kallas *unära*. I vår enkla dataväg kan vi exempelvis utföra unära operationer med register som operander.

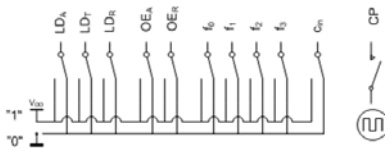
Exempel 7.3 Nollställning av register A

Vi kan nollställa innehållet i register A med en två-cykelsoperation.

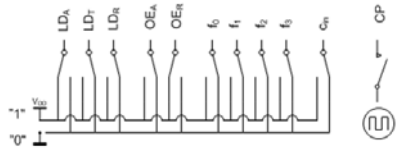
Cykel	Operation (RTN)	Aktiva styrsignaler	Beskrivning
1	0→R	LD_R	ALU'ns U-register nollställs ty F(0), dvs $f_3=f_2=f_1=f_0=0$. Vid klockpulsen överförs U till R.
2	R→A	OE_R; LD_A	Innehållet i register R överförs till register A



Under cykel 1 ställs brytarna i följande lägen:



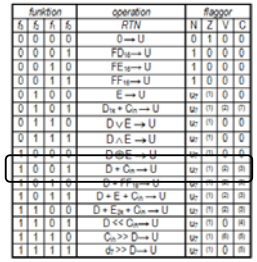
Under cykel 2 ställs brytarna i följande lägen:



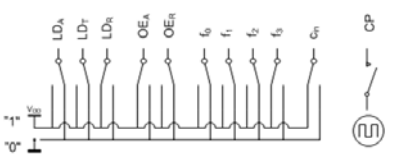
Exempel 7.4 Inkrementering av registerinnehåll A

Vi kan inkrementera (öka med 1) med ALU-funktionen F(9), $D+C_{in}\rightarrow U$, om vi sätter $C_{in}=1$.

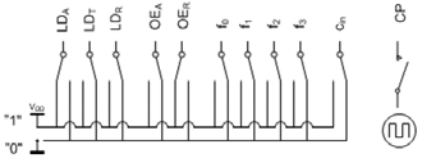
Cykel	Operation (RTN)	Styrsignaler	Beskrivning
1	A+1→R	OE_A; C_{in}; f_3; LD_R	A kopplas till ALU'n Carry in ingår i operationen F(9), dvs $f_3=1, f_2=f_1=0, f_0=1$. Vid klockpulsen överförs U till R.
2	R→A	OE_R; LD_A	Innehållet i register R överförs till register A



Under cykel 1 ställs brytarna i följande lägen:



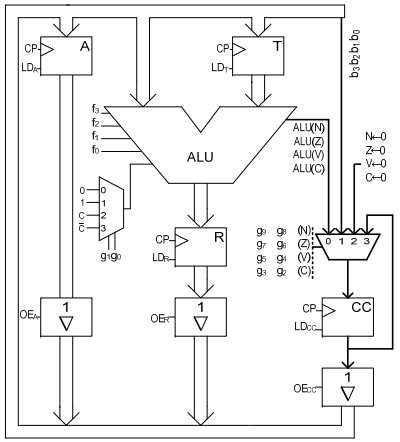
Under cykel 2 ställs brytarna i följande lägen:



7.5 Flaggregistret

ALU:ns sätt att påverka flaggorna är "flyktigt", dvs i samma ögonblick som styrsignalerna **F** ändras så ändras också vanligtvis någon eller några av flaggorna N,V,Z, och C. Olika typer av operationer kan förväntas påverka flaggorna på olika sätt och det blir därför nödvändigt att införa ett register för att hålla reda på de korrekta värdena hos flaggorna. Betrakta Figur 7.8, här har vi infört ett nytt register, CC (*Condition Codes*) där vi samlat våra "verkliga" flaggor. Då vi fortsättningsvis refererar till flaggorna N,V,Z och C avser vi alltså alltid register CC:s innehåll om inget annat tydligt anges.

Vi kan samtidigt utöka användbarheten av flera av ALU:ns operationer genom att påverka C_{in} på olika sätt. Med styrsignalerna g_1, g_0 kan vi nu välja att återkoppla C-flaggan från CC (eller dess invers) till C_{in} hos ALU:n.



FIGUR 7.8 DATAVÄG UTÖKAD MED FLAGGREGISTER (CC) OCH VALBAR C_{IN}-FUNKTION

Logiken för att påverka flaggorna i CC möjliggör att varje flagga, oberoende av övriga, kan väljas från en av fyra källor:

- *ALU* : Flaggan i CC kopieras från ALU:n dvs. sätts baserat på resultatet av ALU:ns senaste operation
- *Buss*: Flaggan i CC kopieras från datavägens buss
- *Konstant*: Flaggor N,Z,V,C nollställs
- *Ingen ändring*: Aktuella värden återförs (ändras ej).

Styrsignalerna g_2 t.o.m g_9 används för att välja flaggor enligt följande tabell:

g_3	g_2	**C** väljs enligt:	RTN	g_5	g_4	**V** väljs enligt:	RTN
0	0	C tas från ALU'n	ALU(C) → C	0	0	V tas från ALU'n	ALU(V) → V
0	1	C tas från bit 0 på bussen	b_0→ C	0	1	V tas från bit 1 på bussen	b_1→ C
1	0	Återställning (nollställning) av C	0→ C	1	0	Återställning (nollställning) av V	0→ V
1	1	C återförs (ändras ej)		1	1	V återförs (ändras ej)	

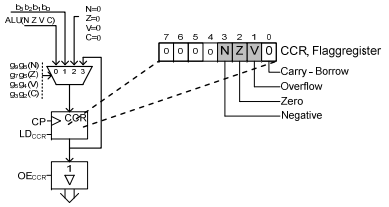
g_7	g_6	**Z** väljs enligt:	RTN	g_9	g_8	**N** väljs enligt:	RTN
0	0	Z tas från ALU'n	ALU(Z) → Z	0	0	N tas från ALU'n	ALU(N) → N
0	1	Z tas från bit 2 på bussen	b_2→ Z	0	1	N tas från bit 3 på bussen	b_3→ N
1	0	Återställning (nollställning) av C	0→ Z	1	0	Återställning (nollställning) av N	0→ N
1	1	Z återförs (ändras ej)		1	1	N återförs (ändras ej)	

Exempel 7.5 Nollställ C-flaggan

Operationen ska nollställa C-flaggan i CC medan övriga flaggor ska förbli oförändrade. Bitarna b_4 t.o.m b_7 i CC används inte och kan heller inte påverkas.

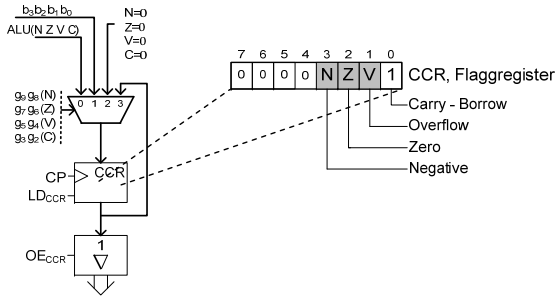
I detta fall utnyttjar vi lämpligen att '0' kan kopplas direkt till flaggorna via väljaren, dvs C tas från '0' medan övriga flaggor återkopplas.

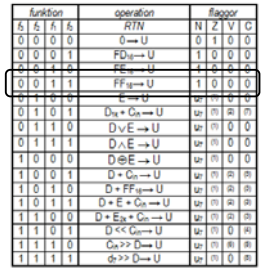
Cykel	Operation (RTN)	Aktiva styrsignaler	Beskrivning
1	0→C	g_3; g_5; g_4; g_7; g_6; g_9; g_8; LD_{CC}	$G_C(2)$, C tas från '0' $G_V(3)$, V återkopplas $G_Z(3)$, Z återkopplas $G_N(3)$, N återkopplas Vid klockpuls laddas CC

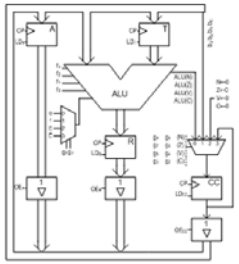


Exempel 7.6 Ettställ C-flaggan

Operationen ska ettställa C-flaggan i CC, övriga flaggor ska förbli oförändrade. Vi har ingen möjlighet att koppla '1' direkt till flaggorna via väljaren men vi kan utnyttja ALU'n för att generera $(FF)_{16}$ på bussen och använda b_0 (som då är '1') för att ett-ställa C-flaggan. För övriga flaggor använder vi "återkoppling" eftersom dessa inte ska ändras under denna operation. Eftersom ALU'ns utsignaler alltid måste passera register R, krävs det två cykler för att utföra denna operation.





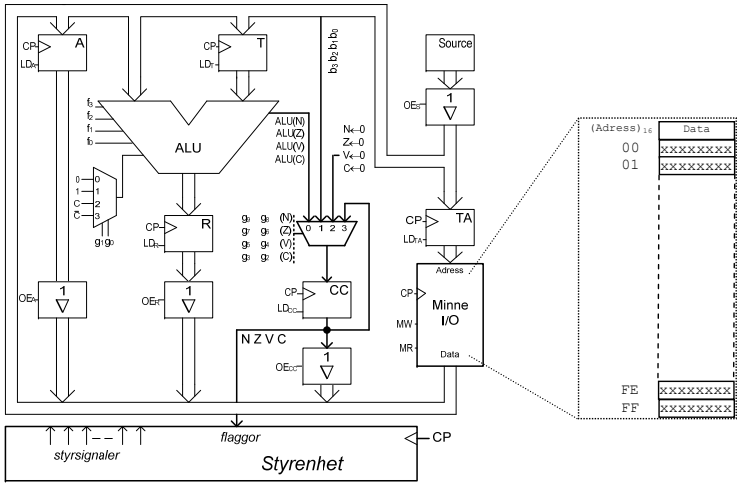


Cykel	Operation (RTN)	Aktiva styrsignaler	Beskrivning
1	1→b_0	f_1; f_0; LD_R;	F(3), dvs f_3=f_2=0, f_1=f_0=1. dvs. FF kopplas till bussen.
2	b_0→C	OE_R; g_2; g_5; g_4; g_7; g_6; g_9; g_8; LD_{CC};	 $G_C(1)$, C tas från b_0 ('1') $G_V(3)$, V återkopplas $G_Z(3)$, Z återkopplas $G_N(3)$, N återkopplas Vid klockpuls laddas CC

7.6 Minnet

Ett begränsat antal generella register innebär med nödvändighet också begränsningar på vad vi kan åstadkomma med vår centralenhet. Vi kan utöka vår centralenhets kapacitet avsevärt genom att lägga till ytterligare generella register men eftersom varje enskilt register kräver ytterligare två styrsignaler (LD/OE) inser vi att detta fort kan bli svårt att hantera.

Som alternativ kan vi bygga upp ett *minne* bestående av ett antal register placerade efter varandra, deras inbördes ordning kallar vi *minnesadress*. För att peka ut exakt ett utav minnets register inför vi ett nytt register TA (*Temporary Address*). För enkelhets skull kan vi ge även TA storleken 8 bitar och vårt minne kan då innehålla 2^8, dvs 256 unika minnesregister, se Figur 7.9. Vi har samtidigt lagt till ett "konstantregister" (*Source*) för att kunna förse datavägen med godtyckliga värden.



FIGUR 7.9 DATAVÄG UTÖKAD MED MINNE

Det finns många olika minnestyper men vi behandlar här bara det viktigaste, läs- och skrivbart minne (*Read Write Memory, RWM*). Vi behöver emellertid några nya begrepp och presenterar därför kort några vanligt förekommande minnestyper och dess användning.

RAM-minnen, (*Random Access Memory*), är beteckningen på en minnestyp där åtkomsttiden är den samma oavsett vilken del (vilket ord) av minnet som adresseras. RAM-minnen förekommer i flera olika varianter:

- PROM (*Programmable Read Only Memories*), minnestypen kan programmeras (skrivas) en gång och behåller därefter sitt innehåll även efter spänningsbortfall.
- FLASH, minnestypen kan programmeras utan speciell typ av utrustning, den behåller sitt innehåll efter spänningsbortfall. Olika typer av flashminnen har olika programmeringscykler vilket innebär att medan någon typ kan omprogrammeras flera tusen gånger kan andra typer endast omprogrammeras något hundratal gånger.
- RWM (*Read Write Memory*), innehållet försvinner vid spänningsbortfall, vid spänningstillslag är innehållet oftast slumpmässigt bestämt.

En minnestyp som behåller sitt innehåll även efter spänningsbortfall kallas *icke-flyktigt* minne (*non volatile memory*) medan RWM är ett *flyktigt* minne (*volatile memory*).

Blockminnen, är den gemensamma beteckningen på minnestyper där data endast är åtkomligt i form av block med ord. Exempel på blockminnen är s.k hårddiskar och flexskiveenheter. Blockstorlekar om 512 bytes (512×8 bitar) är vanligt.

Primärminne är det samlande begreppet för minnestyper RAM. För att data ska kunna bearbetas måste det finnas i primärminne. Data kan inte direkt bearbetas på ett blockminne utan måste först överföras till primärminne, av denna anledning kallas blockminnestyperna ofta *sekundärminnen*.

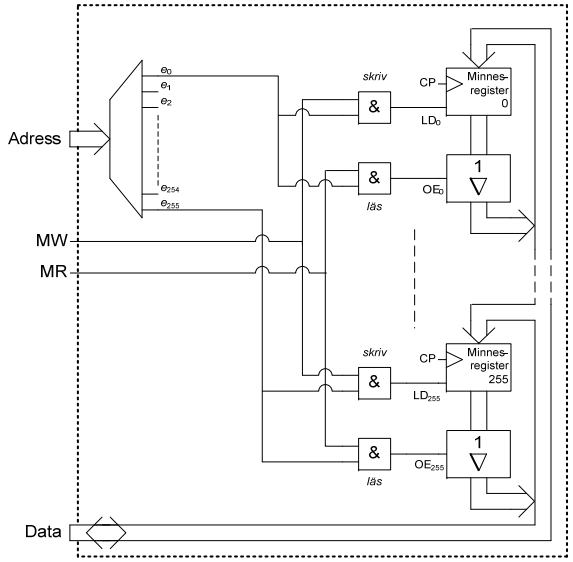
Minnesmodulen som vi använder tillsammans med vår dataväg är av RWM-typ, innehåller 256 register med *Load Enable* och *Output Enable*, ett register för varje minnesord, och speciell logik för att adressera (aktivera) de olika minnesregistren.

Figur 7.10 visar den principiella uppbyggnaden av minnesmodulen. Figuren illustrerar en krets med 256 minnesregister numrerade från 0 till 255 (endast första och sista minnesregister visas). För att kunna adressera varje enskilt minnesregister måste då adressbussens bredd vara 8 bitar (eftersom 2^8 = 256). Aktiveringen av ett speciellt minnesregister sker via en 8 till 256 avkodare, som kopplats till processorns adressbuss. De 256 aktiveringsignalerna kallas e_0-e_{255} och är kopplade till var sitt minnesregister. Endast ett minnesregister kan aktiveras åt gången,

Data som skall läsas *från* minnet eller skrivas *till* minnet överförs på databussen. Speciella styrsignaler används för att utföra skrivning (*Memory Write*, MW) respektive läsning (*Memory Read*, MR) i minnet.

Då styrsignalen MR är aktiv kommer det adresserade minnesregistrets innehåll att kopplas till databussen via bufferten på minnesregistrets utgång, dvs. det adresserade minnesregistrets OE-signal aktiveras.

Då styrsignalen MW är aktiv kommer en skrivning att utföras i minnet. Det adresserade minnesregistrets LD-signal aktiveras och databussens innehåll kommer därför att klockas in till minnesregistret vid nästa klockpuls.



FIGUR 7.10 PRINCIPBILD AV ETT 256-ORDS MINNE SOM KAN LÄSAS OCH SKRIVAS

7.6.1 Läs- och skriv- cykel

Kopiering av ett minnesinnehåll till något generellt register i datavägen eller till ALU:n kallas allmänt *läscykel*. På motsvarande sätt kallar vi kopiering från något av datavägens register till någon minnescell för *skrivcykel*. Det är uppenbart att det tar längre tid att överföra data mellan minnet och datavägen (register eller exempelvis ALU:n) än att överföra data från konstantregistret eller mellan

något generellt register i datavägen till ALU:n. Förklaringen till detta är förstås att vi för varje minnesåtkomst först måste bilda en adress och adressera det avsedda minnesregistret via TA.

Med begreppet *operand* menar vi det dataobjekt som ingår i en speciell operation, operanden anges av ett distinkt *adresseringssätt*. Adressbildningen sker på olika sätt beroende på det valda adresseringssättet och vi återkommer till detta längre fram, låt oss först bara, med några enkla exempel, se vad vi kan åstadkomma med vår konstruktion så här långt. Vi börjar med de allmänt förekommande metoderna *omedelbar*, *absolut*, *indirekt* och *register-indexerad*.

7.6.1.1 Omedelbar adressering

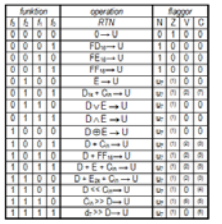
Omedelbar addressering innebär att ett konstant värde används som operand. Med vår enkla centralenhet kan då operanden ges med hjälp av *Source*-registret, dvs. med en uppsättning omkopplare. Anledningen till att vi betraktar detta som en konstant är att styrsignalerna (omkopplarens lägen) måste ändras då man vill ändra konstantens värde.

Omedelbar adressering används såväl för att ange värden (används direkt i beräkningarna) som att ange adresser till minnesregister.

Exempel 7.7 Omedelbar adressering, placera värdet 80_{16} i register A

Skapa en styrsignalsekvens som placerar konstanten 80_{16} i register A. Flaggorna i CC ska påverkas så att N och Z sätts baserat på operanden (dvs. 80_{16}), V och C ska nollställas. RTN för operationen är:

$80_{16} \rightarrow A$; cond $\rightarrow$ CC



Lösning:

Vi använder här konstantregistret (Source), placerar konstanten i detta register och klockar innehållet direkt till register A. För att få korrekt flaggsättning väljer vi samtidigt ALU-funktion F(9), dvs. bussens värde kopieras via ALU'ns ingång D till U och påverkar samtidigt ALU'ns flaggor. Observera speciellt att C_{in} nu måste vara 0 för att flaggsättningen ska bli riktig. Av flaggorna från ALU'n använder vi N och Z medan vi nollställer V och C i CC-registret.

I följande tabell har vi sammanfattat den åtta-poliga "Source"-omkopplarens olika lägen som ett hexadecimalt tal, medan övriga styrsignaler anges som aktiva ("1") alternativt passiva (tomma rutor).

Klock-puls	Source	Output Enable				Load Enable					ALU(f)				MUX(g)									MR	MW
		S	A	R	CC	A	T	TA	R	CC	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
1	80_{16}	1				1				1	1			1					1		1					

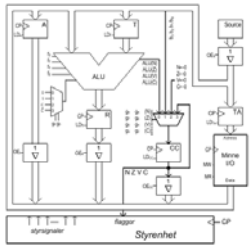
7.6.1.2 Absolut adressering

Absolut, eller som det också i bland kallas *utökad*, adressering används då operanden utgörs av innehållet på någon adress i minnet. Minnesadressen kan därför sägas vara en *platshållare* och styrenheten kan nu också ändra operanden (minnets innehåll) till skillnad från vid användning av omedelbar adressering. Ett dataobjekt som kan ändras som följd av att en sekvens styrsignaler utförs kallas också allmänt för en *variabel*, av just denna anledning.

Exempel 7.8 Läscykel med absolut adressering

Skapa en styrsignalsekvens som placerar värdet på adress 10_{16} i register A. Flaggorna i CC ska påverkas så att N och Z sätts baserat på operanden (dvs. innehållet på adress 10_{16}), V och C ska nollställas. RTN för operationen är:

$M(10_{16}) \rightarrow A$; cond $\rightarrow$ CC



Lösning:
Under första klockcykeln placerar vi adressen 10_{16} i TA-registret för att adressera minnet.
Under andra klockcykeln läses innehållet på adress 10_{16} (MR är aktiverad) till register A. Flaggsättningen sker på samma sätt som i föregående exempel.

Klock-puls	Source	Output Enable				Load Enable					ALU(f)				MUX(g)										MR	MW
		S	A	R	CC	A	T	TA	R	CC	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
1	10_{16}	1						1																		
2						1				1	1			1					1		1				1	

Förfarandet är likartat då vi vill skriva data till minnet.

Exempel 7.9 Skrivcykel med absolut adressering

Skapa en styrsignalsekvens som kopierar värdet i register A till adress 60_{16}. Flaggorna i CC får inte ändras. RTN för operationen är:

$A \rightarrow M(10_{16})$

Lösning:
Under första klockcykeln placeras adressen 60_{16} i TA-registret för att adressera minnet.
Under andra klockcykeln aktiveras OE_A och minnets skrivsignal MW.

Klock-puls	Source	Output Enable				Load Enable					ALU(f)				MUX(g)										MR	MW
		S	A	R	CC	A	T	TA	R	CC	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
1	60_{16}	1						1																		
2			1																							1

7.6.1.3 Register-indexerad adressering

Operandens address kan placeras i ett register och därefter refereras via detta register. Metoden kallas ofta register-indexerad och är speciellt användbar för så kallade *pekarvariabler*, dvs. variabler som används för att referera till ett speciellt dataobjekt, snarare än då dataojektet utgör variabeln.

Exempel 7.10 Register-indexerad adressering

Skapa en styrsignalsekvens där innehållet i register A används för att adressera minnet. Minnesinnehållet på denna adress ska placeras i register T för senare bearbetning i ALU'n. Flaggorna i CC ska inte påverkas. RTN för operationen är:

(A) → T

Lösning:
Under första klockcykeln kopieras innehållet från register A till TA-registret för att adressera minnet.
Under andra klockcykeln kopieras minnesinnehållet till register T.

Klock-puls	Source	Output Enable				Load Enable					ALU(f)				MUX(g)										MR	MW
		S	A	R	CC	A	T	TA	R	CC	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
1			1					1																		
2							1																		1	

Vi har sett exempel på operationer som inbegriper det generella registret A på något sätt. Detta är dock inte nödvändigt, man kan också tänka sig ”minne till minne”-operationer där generella arbetsregisters innehåll lämnas opåverkade. Vi tillåter dock ändring av registren T och R i datavägen. Man kan då tänka sig att kombinera flera adresseringssätt i en operation som i följande exempel.

Exempel 7.11 Initiering av minnesinnehåll

Skapa en styrsignalsekvens som placerar värdet 44_{16} på adress 20_{16} i minnet. Innehållen i registren A och CC får inte påverkas av operationen. RTN för operationen är:

$44_{16} \rightarrow M(20_{16})$

Lösning:
Under första klockcykeln kopieras konstantregistret med värdet 20_{16} till TA-registret för att adressera minnet.
Under andra klockcykeln läggs konstantregistret med värdet 44_{16} ut på bussen i en skrivcykel.

Klock-puls	Source	Output Enable				Load Enable					ALU(f)				MUX(g)										MR	MW
		S	A	R	CC	A	T	TA	R	CC	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
1	20_{16}	1						1																		
2	44_{16}	1																								1

Exempel 7.12 Kopiering av minnesinnehåll

Skapa en styrsignalsekvens som kopierar innehållet på adress 30_{16} till adress 31_{16} i minnet. Innehållen i registren A och CC får inte påverkas av operationen. RTN för operationen är:

$M(30_{16}) \rightarrow M(31_{16})$

Lösning:
Under första klockcykeln kopieras konstantregistret med värdet 30_{16} till TA-registret i en läscykel.
Under andra klockcykeln kopieras minnesinnehållet till register T.
Under tredje klockcykeln läggs konstantregistret med värdet 31_{16} ut på bussen i en skrivcykel. Register T kopplas vidare genom ALU'n till register R utan att värdet ändras.
Under sista klockcykeln kopieras innehållet i register R till minnet.

Klock-puls	Source	Output Enable				Load Enable					ALU(f)				MUX(g)										MR	MW
		S	A	R	CC	A	T	TA	R	CC	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
1	30_{16}	1						1																		
2							1																		1	
3	31_{16}	1							1																	
4				1																						1

7.6.2 Binära operationer

Det tillförda minnet ger oss också möjlighet att utföra binära operationer som, genom att kombineras med olika adresseringssätt, ökar flexibiliteten hos operationerna. Hur ska då operanderna specificeras, dvs. vilka adresseringssätt ska kunna användas för operationerna? Vi har sett hur minnesåtkomster genererar en extra klockcykel vilket medför längre exekveringstider ju fler minnesoperationer som krävs. Ett enkelt sätt att minska sådana effekter är att alltid låta ett generellt register ingå i operationen och dels tillhandahålla den ena operanden och samtidigt utnyttjas för att lagra resultatet. Binära operationer får då formen:

A op operand2 →A

Dvs. initialvärdet i register A utgör den första operanden, *operand2* anges med något av de möjliga adresseringssätten, och *op* är någon av de binära operationer som kan utföras av ALU'n.

Operationen kan delas in i tre steg:

1. Operand 2 kopieras till register T, detta kan ta flera klockcykler beroende på adresseringssättet.
2. Innehållet i register A placeras på bussen för att konfigurera ALU'ns operand 1
3. Operationen utförs och resultatet överförs till register R, ALU'ns flaggbitar överförs till CC på ett för operationen korrekt sätt.

Den binära operationen kan huvudsakligen beskrivas med följande generella styrsignalsekvens:

Steg	Operation (RTN)	Aktiva styrsignaler	Beskrivning
1	Operand2→T	LD_T; minnesoperation	Operand 2 (högerled) överförs till register T
2	A*op*T→R	LD_R; F(*op*); (G(op); LD_{CC})	ALU'n utför någon operation (*op*) och resultatet placeras i resultatregistret R, ev. flaggpåverkan
3	R→A	OE_R; LD_A;	Återför resultatet till register A

Antalet cykler som krävs för att utföra en hel operation på detta viset varierar med det valda adresseringssättet hos den utförda operationen. Observera också att operandernas ordning vid binära operationer egentligen här bara har betydelse för subtraktionsoperationen. Lägg också märke till att vi anger *aktiva* styrsignaler i datavägen, övriga styrsignaler förutsätts vara passiva, varför operationen kan anses vara entydigt definierad även med denna enkla beskrivning.

Vi avslutar detta kapitel med em detaljerad beskrivning av hur en binär operation utförs i vår dataväg.

Exempel 7.13 Addera och ackumulera till register A

Skapa en styrsignalsekvens som adderar innehållet på adress 30_{16} till innehållet i register A och slutligen återför resultatet till register A. RTN för operationen är:

$A + M(30_{16}) \rightarrow A$

Detta är en ordinär aritmetisk operation och flaggorna ska påverkas enligt reglerna för addition, dvs:

N: Ettställs om resultatets teckenbit (bit 7) får värdet 1 vid additionen
Z: Ettställs om samtliga åtta bitar i resultatet blir noll vid additionen
V: Ettställs om overflow vid 2-komplementsrepresentation inträffar vid additionen
C: Ettställs om summan vid additionen ej ryms i åtta bitar, dvs blir större än eller lika med 256

Lösning:
För denna operation används ALU'ns funktion F(11), dvs addition, med C_{in}=0, se Tabell 7.2, vi noterar också att flaggorna då sätts precis så som specificeras ovan.
Under första klockcykeln kopieras konstantregistret med värdet 30_{16} till TA-registret i en läscykel.
Under andra klockcykeln kopieras minnesinnehållet till register T.
Under tredje klockcykeln läggs register A ut på bussen under ALU-operationen F(11), samtidigt överförs resultatet från ALU'n till register R (LD_R) och ALU'ns flaggsättning till register CC (LD_{CC})
Register T kopplas vidare genom ALU'n till register R utan att värdet ändras.
Under sista klockcykeln kopieras innehållet i register R till register A.

Operationen $A + M(30_{16}) \rightarrow A$ kan nu fullständigt beskrivas enligt följande:

Steg	Operation (RTN)	Aktiva styrsignaler	Beskrivning
1	30_{16}→TA	OE_S ; LD_{TA}	Förbered läsning från minnet
2	(30_{16})→T	MR ; LD_T	Innehållet på adress 30_{16} överförs till register T
2	A+T→R	OE_A; F(11);LD_R; LD_{CC}	Innehållet i A läggs på bussen, ALU'n utför addition, resultatet placeras i resultatregistret R och flaggorna i CC
3	R→A	OE_R; LD_A	Återför resultatet till register A

Följande alternativa sätt beskriver också fullständingt operationen $A + (30_{16}) \rightarrow A$:

Klock-puls	Source	Output Enable				Load Enable					ALU(f)				MUX(g)										MR	MW
		S	A	R	CC	A	T	TA	R	CC	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
1	30_{16}	1						1																		
2							1																		1	
3			1						1	1	1		1	1												
4				1																						1

8 Styrenheten

Den databehandlande enheten med minne från förra kapitlet, återgiven i Figur 7.9, har stora likheter med de första helt elektroniska programmerbara maskinerna byggda redan strax efter andra världskrigets slut. Man kunde vid denna tid, dvs. i slutet av 1940-talet, konstruera processorer vars styrenheter var utformade för att lösa speciella problem, som exempelvis beräkningar av projektilbanor. Dessa styrenheter var konstruerade för att generera ändringsbara styrsignalsekvenser och var därför i någon mening "programmerbara" styrenheter. Programmeringen gick till så att man ställde in styrenhetens beteende genom att koppla om ledningar och ställa omkopplare.

Den första generationen elektroniska beräkningsmaskiner var alltså mycket krävande i form av ”programmerare”. Först måste man analysera beräkningsalgoritmen och konstruera en sekvens av operationer, dvs. styrsignaler, som stegvis utförda ledde fram till algoritmens slutpunkt, dvs. ”svaret”. Därefter måste man, baserat på beskrivningen av styrsignaler, koppla dessa fysiskt i beräkningsmaskinen. Det är ganska klart att denna organisation inte tillåter generella program i någon större omfattning, det blir helt enkelt för dyrt och komplicerat att ställa in alla omkopplare för att lösa ett nytt beräkningsproblem. Inte desto mindre användes denna typ av maskiner för att beräkna en lång rad funktionsvärden som publicerades i olika typer av tabellverk under flera påföljande årtionden.

Nästa steg i datorevolutionen, tidigt 1950-tal, blev att kombinera styrsignalsinformation i så kallade *instruktioner*, med data i maskinens minne. Vi ser nu *det lagrade programmets princip*, vilket automatiserar det sista steget i programmeringsprocessen, dvs. att fysiskt ställa om alla omkopplare för ett nytt program. I stället lagras styrsignalsinformationen nu tillsammans med data i maskinens minne. Styrsignalinformationen, dvs. instruktionerna, lagras sekvensiellt för sig, medan data kan lagras på något helt annat ställe i maskinens minne. Ersättningen av mänskliga programmerare är naturligtvis inte gratis. Jobbet måste fortfarande göras och man måste nu i stället konstruera en speciell styrenhet som skapar styrsignalsekvenser korrekt utifrån någon speciell instruktion, *den automatiska styrenheten*.

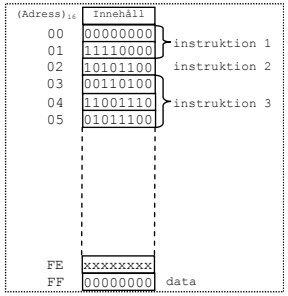
I detta kapitel behandlar vi den automatiska styrenheten vilken oftast är den i särklass mest komplicerade digitala konstruktionsdelen av en modern dator och den kan byggas på många olika sätt. I detta kapitel visar vi hur styrsignaler för en autonom styrenhet, kan genereras från enkla disjunktiva booleska uttryck, dvs. styrenheten kan byggas upp enbart med enkla grindar som NOT, AND och OR. speciellt beskriver vi FLIS- (Flexible Instruction Set) processorn, eller mer kortfattat FLISP.

8.1 Det lagrade programmets princip

”Det lagrade programmets princip” tillskrivs ofta den amerikanske matematikern John von Neumann. Idén innebär att maskinoperationer kodas som binära tal, precis som data. På detta sätt kan operationer och data lagras om vartannat i datorns minne och vi slipper speciella enheter med strömbrytande funktioner för programmering av de operationer som ska utföras.

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

En instruktion är 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 av två delar; *operation* och *operandinformation*.



FIGUR 8.1 EXEMPEL PÅ MINNE MED INSTRUKTIONER OCH DATA

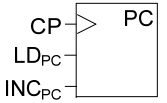
Operanden är antingen data (datavärdet) eller någon form av adress till data. Vissa instruktioner saknar dock operand och därmed också operanddel. Jämför med Figur 8.1 som illustrerar instruktioner med olika längd. Observera också att det inte finns något sätt att skilja instruktioner från data i minnet enbart genom observation av minnesinnehållet. Vi måste veta exakt *var* varje instruktion börjar och *vad* vi ska tolka som data.

Den del av instruktionen som specificerar operationen och anger operandkallas *operationskod*, OP. Utöver detta krävs också information om eventuella operander. Unära operationer kräver då en operandinformation, binära operationer kräver två operandinformationer. Det finns också speciella typer av operationer som kräver någon extra operandinformation överhuvudtaget och vi återkommer senare till sådana.

8.2 En automatisk styrenhet

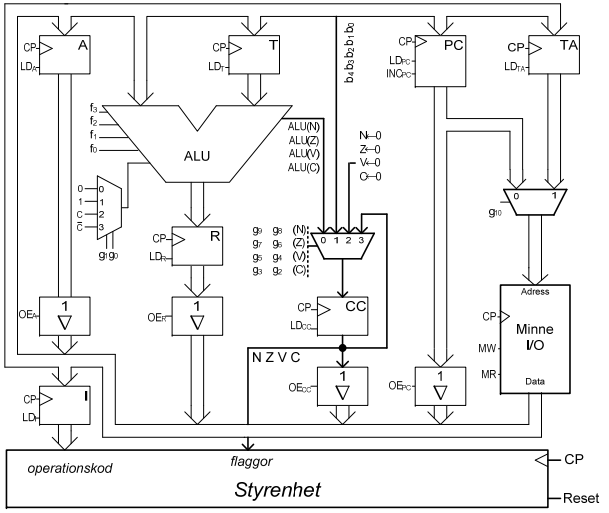
En automatisk styrenhet i en von Neumann-dator förutsätter att instruktioner placerats efter varandra (”konsekutivt”) i minnet. En komplett instruktionssekvens som utför någon specifik databehandlingsuppgift kallas ett *program*. Minnet kan då innehålla både program och data.

Det måste finnas en mekanism i processorn som skiljer mellan instruktionerna i ett program 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 speciellt register, *programräknare* (*program counter*, PC). Programräknarens innehåll uppdateras automatiskt efter hand som instruktioner hämtas och utförs. Eftersom operationen PC+1→PC är mycket frekvent så bör PC förses med den speciella funktionen INC_{PC}, i övrigt kan PC utformas som ett generellt register (Figur 8.2).



FIGUR 8.2 SYMBOL FÖR PC I DATAVÄGEN

Se Figur 8.3, här har vi ersatt vår manuella styrenhet med en automatisk och samtidigt infört såväl en programräknare, PC som ett instruktionsregister I. Instruktionsregistrets ingång har anslutits till databussen så att instruktionens operationskod kan laddas i I-registret då processorn läser instruktionen från minnet.



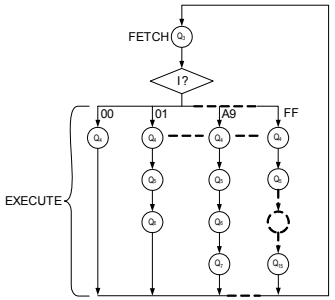
FIGUR 8.3 DATABEHANDLANDE ENHET KOMPLETTERAD MED PROGRAMRÄKNARE, PC OCH INSTRUKTIONSREGISTER I

8.2.1 Instruktionsutförande

I sin enklaste form kan utförandet av en instruktion delas in i två faser:

I den första fasen, som kallas *hämtfasen* (eng. *FETCH*), hämtas (läses) operationskoden för instruktionen i minnet och lagras i instruktionsregistret I, se även Exempel 8.1. Därmed är hämtfasen slut och processorn övergår till utföradefasen.

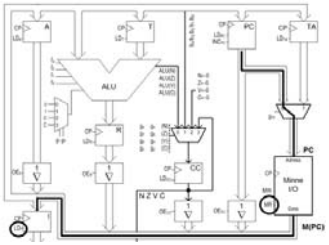
I den andra fasen, som kallas *utförandefasen* (eng. *EXECUTE*), avkodar styrenheten innehållet i instruktionsregistret I, dvs operationskoden, och utför en styrsignalsekvens som beror på vilken instruktion som skall utföras. När utförandefasen är slut startas nästa instruktions hämtfas enligt tillståndsgrafen i Figur 8.4. Varje nod i tillståndsgrafen representerar en cykel, varje graf illustrerar en övergång då klockpulsen ges. I figuren illustreras också hur exekveringsfasen hos olika instruktioner (operationskoder) kan ha olika antal tillstånd medan hämtfasen är identisk för alla instruktioner.



FIGUR 8.4 FÖRENKLAD TILLSTÅNDSGRAF FÖR INSTRUKTIONERNAS TVÅ FASER

Exempel 8.1 Styrsignaler för att läsa en instruktion från minnet (FETCH)

Hämtfasen börjar med att innehållet i programräknaren PC, läggs ut på minnets adressbuss. En läsning, styrsignalen MR är 1, görs från minnet. Det aktuella minnesordet, M(PC), som nu förutsätts vara en instruktions operationskod, läggs då ut på bussen av minnet, 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.



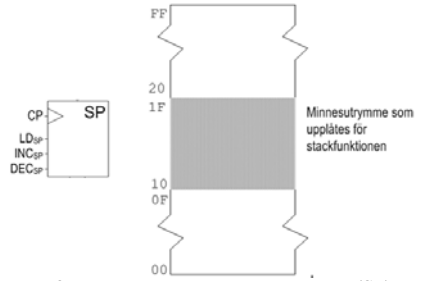
Styrsignalerna för FETCH-fasen blir då:

RTN-beskrivning	Styrsignaler	Kommentarer
M(PC)→I;	MR=1; LD_I=1; INC_{PC}=1;	Adressen för nästa instruktions operationskod dvs. PC kopplas till minnets adressbuss. Läs operationskoden från minnet och placera i instruktionsregistret I. Adressen som finns i PC ökas med ett.

8.2.2 Stackfunktion

En *stackfunktion* är en enkel mekanism som används för kort tids undanlagring av registerinnehåll i minnet. Funktionen underhålls med hjälp av en *stackpekare*, ett adressregister som används för att adressera den del av minnet som upplåts för stacken. Operationerna SP+1→SP respektive SP-1→SP är vanligt förekommande, och därför utökar vi registerkonstruktionen med dessa operationer.

Själva stackfunktionen kan nu implementeras med två olika operationer, "lägg på stacken" (*push*) och plocka från stacken (*pull*). Här betecknar nu r ett godtyckligt register (A,CC eller PC) i datavägen.



FIGUR 8.5 ILLUSTRATION AV STACPEKARE (SP) OCH MINNESUTRYMME FÖR STACKFUNKTION

push

- Först minskas SP med 1 för att skapa utrymme för registerinnehållet
- Därefter placeras registerinnehållet på det nu utpekade minnesutrymmet.

SP-1→SP,
r→M(SP)

pull

- Värdet på den adress som SP innehåller, kopieras till ett register
- Därefter ökas SP med 1, och återställer därmed stacken

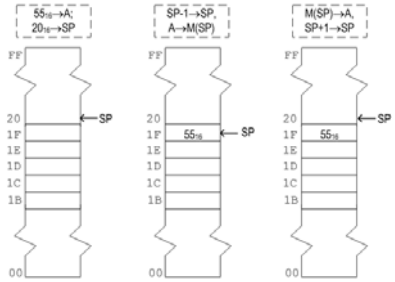
M(SP)→r,
SP+1→SP

Exempel 8.2 Illustration av stackoperationer

Instruktionssekvensen

```
LDSP  #20₁₆
LDA   #55₁₆
PSHA
PULA
```

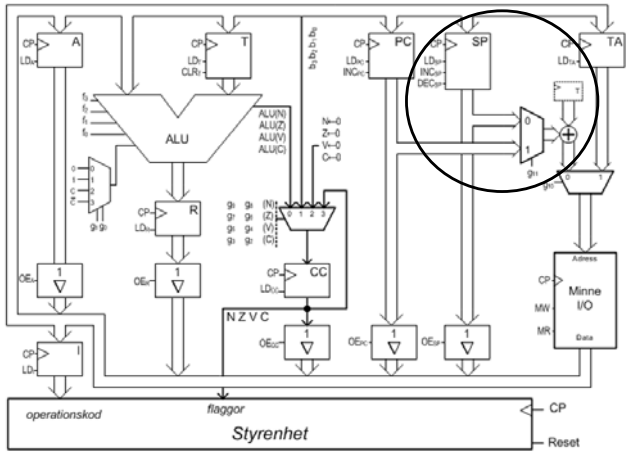
Illustreras i figuren .



Den första bilden visar situationen efter det att register A getts värdet 55_{16} och stackpekaren SP värdet 20_{16}. Vi säger att SP ”pekar på” adressen 20_{16}, detta minnesutrymme används dock inte eftersom *push*-operationen minskar stackpekaren med 1 innan ett nytt värde skrivs till stacken. Detta illustreras i den mellersta bilden som visar situationen efter *push*-operationen. Den sista bilden visar stackens utseende efter *pull*-operationen. Observera speciellt att det ditlagda värdet 55_{16}, naturligtvis fortfarande finns kvar i minnet. Det kommer dock att skrivas över av nästa *push*-operation.

Stackpekarens initiala värde kallas ofta *bottom of stack* (BOS) medan stackpekarens aktuella värde kallas *top of stack* (TOS).

Stackregistret placeras in i datavägen för att enkelt kunna adressera minnet på samma sätt som PC. En väljare, med styrsignal g_{11} för att välja register, införs. För att ytterligare utöka användbarheten av stacken kopplar vi temporärregistret T till en adderare som lägger till värdet i T till SP innan minnet adresseras. Det blir då möjligt att komma åt, inte bara det värde som sist lades på stacken, utan vilket som helst, av de värden som tidigare lagts dit. Se Figur 8.6 nedan.



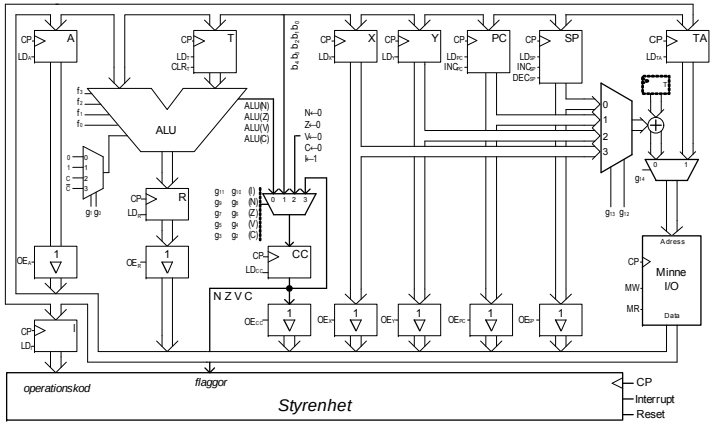
FIGUR 8.6 DATABEHANDLANDE ENHET KOMPLETTERAD MED STACKREGISTER, SP

8.3 FLIS-processorn

FLISP arbetar genomgående med åtta bitars ordlängd, vilket innebär att den har en åtta bitars ALU, åtta bitars register samt åtta bitars adress- och databuss. Med åtta bitars adressbuss begränsas primärminnets storlek till $2^8 = 256$ adresser. Det är dock viktigt att inse att de begränsningar som FLIS-processorn innebär kan överbryggas. Busbredden kan exempelvis utvidgas till 32 bitar utan att våra resonemang förändras, antalet register kan utökas, det är bara storleken (komplexiteten) och förståeligheten som ändras.

Utöver tidigare beskrivna register i datavägen har FLISP dessutom försetts med ytterligare två register (X och Y) speciellt avsedda för adressberäkningar.

I Figur 8.7 ges en detaljerad beskrivning av FLIS-processorn's dataväg medan styrenheten här illustrerats som ett enkelt block. Vi återkommer strax till en detaljerad beskrivning av styrenheten.



FIGUR 8.7 FLIS-PROCESSORNS DATAVÄG

De nödvändiga signalerna för att kontrollera datavägen, *styrsignalerna*, kan nu sammanfattas i följande tabell:

Signal	Funktion
Insignaler till styrenheten	
CP	"Clock Pulse", systemklocka
Reset	Asynkron återställningssignal
Interrupt	Extern avbrottsignal, samplas vid CP (positiv flank),kontrolleras vid NF
Q_{state}	Tillstånd, "state" anges på hexadecimal form med siffrorna 0 t.o.m F, dvs totalt 16 olika tillstånd
I_{opcode}	Operationskod, anger innehållet i instruktionsregistret avkodat av en "en av 256-väljare". "opcode" anges på hexadecimal form, dvs 0 t.o.m FF.
N-flagga	N-flaggan, dvs. bit 3 i CC
Z-flagga	Z-flaggan, dvs. bit 2 i CC
V-flagga	V-flaggan, dvs. bit 1 i CC
C-flagga	C-flaggan, dvs. bit 0 i CC
Utsignaler från styrenheten	
LD_A	"Load register A enable", Ackumulator
LD_I	"Load register I enable", instruktionsregister
OE_A	"Output enable register A", Ackumulator
LD_T	"Load register T enable", Temporärregister
CLR_T	"Clear register T", Temporärregister
LD_R	"Load register R enable", Resultatregister
OE_R	"Output enable register R", Resultatregister
LD_{CC}	"Load register CC enable", Flaggregister (Condition Codes)
OE_{CC}	"Output enable register CC", Flaggregister (Condition Codes)
LD_X	"Load register X enable", Adressregister
OE_X	"Output enable register X", Flaggregister
LD_Y	"Load register Y enable", Adressregister
OE_Y	"Output enable register Y", Flaggregister
LD_{PC}	"Load register PC enable", Programräknare
OE_{PC}	"Output enable register PC", Programräknare
INC_{PC}	"Increment register PC", Addera 1 till programräknare
LD_{SP}	"Load register SP enable", Stackpekare
OE_{SP}	"Output enable register SP", Stackpekare
INC_{SP}	"Increment register PC", Addera 1 till stackpekare
DEC_{SP}	"Decrement register SP", Subtrahera 1 från stackpekare
LD_{TA}	"Load register TA enable", Adressberäkningsregister
MR	"Memory Read", Avkoda adressbuss och klocka ut data från minne/"Input"
MW	"Memory Write", Avkoda adressbuss och klocka in data till minne/"Output"
$f_3..f_0$	Funktionsväljare för ALU
$g_{14}..g_0$	Styrsignaler för väljarfunktioner

TABELL 8.1 SIGNALER HOS FLIS-PROCESSORNS STYRENHET

8.4 FLISP's fasta styrenhet

Konstruktion av fasta styrenheter omfattar en rad överväganden mellan mängden hårdvara som används, högsta möjliga exekveringshastighet, ofta liktydigt med klockfrekvens och förstås kostnaden för konstruktionsprocessen som sådan. En typisk centralenhet innehåller åtskilliga styrsignaler och när många olika instruktioner ska konstrueras kommer dessa styrsignaler att behöva kombineras på många olika sätt vilket ger en omfattande och svåröverskådlig konstruktion.

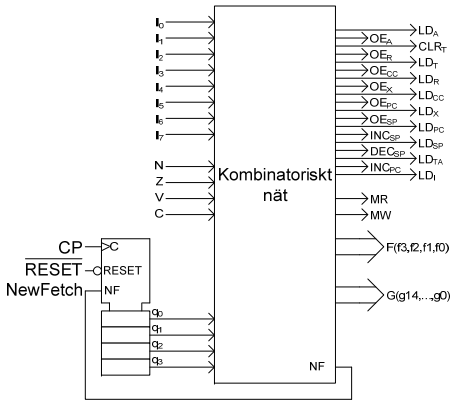
Då det gäller FLIS-processorns fasta styrenhet har vi prioriterat en lättförståelig realisering snarare än att försöka hitta den effektivaste lösningen. En översiktlig bild av FLIS-processorn's styrenhet visas i Figur 8.8. Styrenhetens uppgift är alltså att generera samtliga olika styrsignaler som krävs, i datavägen, för utförandet av maskininstruktionerna. Dessa styrsignaler utgör styrenhetens utsignaler.

Vi har tidigare sett hur en instruktion utförs i flera steg, ju komplexare instruktion, desto fler steg. Av denna anledning bygger vi styrenheten som ett sekvensnät med en (omfattande) kombinatorisk del. Eftersom varje instruktion utförs i flera steg krävs en *sekvensierare* som i detalj, dvs. varje sekvens, av maskininstruktionen beskriver exakt vilka styrsignaler som ska vara aktiva, respektive passiva.

Styrenheten har flera olika typer av insignaler:

- För att styrenheten ska aktivera rätt sekvens av kontrollsignaler måste maskininstruktionen (Operationskoden) finnas tillgänglig. FLISP's dataväg tillhandahåller denna via instruktionsregistret.

- Villkorliga instruktioner utförs på basis av innehållet i flaggregistret (CC), dessa flaggor finns därför tillgängliga som insignaler i styrenheten.
- För att styrenheten skall kunna startas på ett entydigt definierat sätt krävs en startsignal *Reset*.
- Sekvensnät kan vara *synkrona*, dvs sådana som vi hittills behandlat, eller *asynkrona*. Det är inte inom ramen för denna litteratur att behandla asynkrona sekvensnät varför vi alltid har en klocksignal (CP) närvarande.

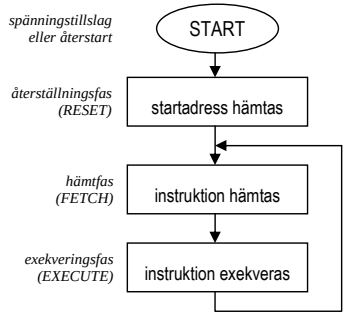


FIGUR 8.8 FLIS-PROCESSORNS STYRENHET

Låt oss börja med styrenhetens sekvensierare, dvs. den del som bestämmer i vilken ordning saker och ting ska utföras.Vi skall strax återkomma till *hur* den åstadkommer saker och ting men koncentrerar oss först på *vilka* saker och ting som ska utföras.

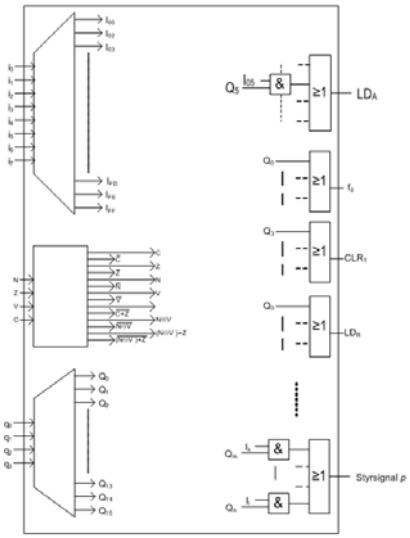
Uppgifterna för styrenhetens sekvensierare kan delas in i tre olika faser:

- Återställningsfas
- Hämtfas
- Utförandefas



FIGUR 8.9 INSTRUKTIONSCYKELN KOMPLETTERAD MED S K ÅTERSTÄLLNINGSFAS

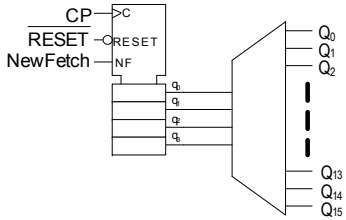
Det kombinatoriska nätet i styrenheten illustreras i Figur 8.10 nedan. Förutom avkodning av tillståndsräknare och instruktionernas OP-koder innehåller det en s k AND-OR area där styrsignalerna skapas som (Booleska) summor av produkttermer. Produkttermerna består av tillståndsvariablerna q_0 - q_3, OP-kodbitarna i_0 - i_7 och eventuellt flaggbitarna om en villkorlig instruktion skall utföras.



FIGUR 8.10 KOMBINATORISK DEL AV FAST KOPPLAD LOGIK

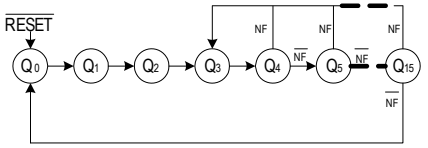
8.4.1 Tillståndsgenerering

Av Figur 8.11 framgår att räknarens utsignaler q_3, q_2, q_1 och q_0 är kopplade till en 4/16-avkodare med de sexton utsignalerna Q_0 – Q_{15}. Av kopplingen framgår att endast en av dessa signaler kan vara aktiv åt gången och att någon av signalerna alltid måste vara aktiv.



FIGUR 8.11 TILLSTÅNDSGENERERING MED RÄKNARE OCH 4/16-AVKODARE

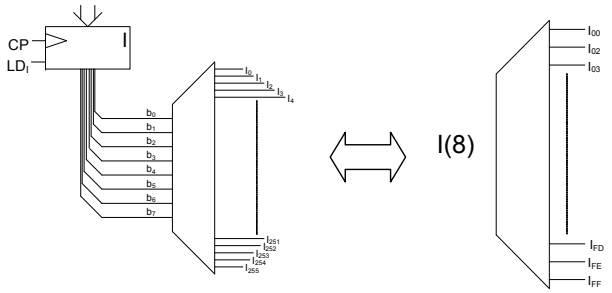
Konstruktionen i Figur 8.11 resulterar i följande tillståndsgraf, dvs. tillståndsmaskinen befinner sig alltid i ett av tillstånden Q_0 t.o.m Q_{15}.



FIGUR 8.12 TILLSTÅNDSGRAF FÖR FLIS-PROCESSORN

8.4.2 Instruktionsavkodning

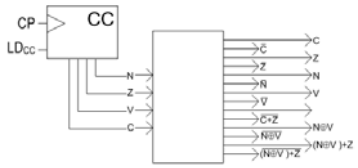
Instruktionsregistret som ingår i datavägen används för att lagra instruktionens operationskod under instruktionsutförandet. I Figur 8.13 illustreras hur instruktionsregistrets innehåll omvandlas till styrsignaler i styrenheten med hjälp av en 8/256 avkodare. Till höger i figuren visas en förenklad illustration där vi namngett de 256 utsignalerna I_{00}, I_{01}, ...I_{FE} och I_{FF}. Även i denna koppling ser vi hur exakt en av de 256 signalerna alltid är aktiv medan övriga är 0.



FIGUR 8.13 AVKODNING AV INSTRUKTIONSREGISTRET

8.4.3 Generering av flaggvillkor

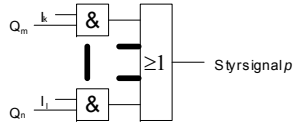
De olika flaggvillkoren som används vid villkorliga "branch"-instruktioner bildas i ett kombinatoriskt nät av insignalerna N,Z,V och C som hämtats direkt från datavägens CC-register.



FIGUR 8.14 FLAGGVILLKOR FÖR VILLKORLIGA "BRANCH"-INSTRUKTIONER

8.4.4 Generering av styrsignaler

Varje enskild styrsignal i datavägen bildas av ett AND/OR-logiknät (summa av produkter). Som insignaler används någon av den avkodade räknarens signaler Q_0-Q_{15}. Genom att kombinera Q med någon av signalerna (I) från det avkodade instruktionsregistret, skapas unika signaler för varje kombination av *tillstånd*/*instruktion*.



FIGUR 8.15 BILDANDE AV STYRSIGNAL I ETT AND/OR-NÄT

Betrakta Figur 8.15, till varje AND-grind finns exakt en I-signal och exakt en Q-signal ansluten. Uttrycket I_k Q_m är alltså sant för utförandet av precis instruktionen I_k som samtidigt befinner sig i klockcykel Q_m.Det är uppenbart att en enskild styrsignal i kan aktiveras för flera olika instruktioner och i olika tillstånd varför ett antal AND-grindar krävs

Betrakta nu på nytt Figur 8.9. Där illustreras hur uppstart sker när en startsignal ges till styrenheten, återställningsfasen (ett antal tillstånd) kommer då att genomlöpas. Därefter övergår styrenheten till hämtfasen för att placera operationskoden till en instruktion i instruktionsregistret. Instruktionen

avkodas och utförs i exekveringsfasen och styrenheten fortsätter därefter med att hämta och utföra instruktioner enligt figuren.

För att i detalj illustrera styrenhetens arbetssätt ska vi i de nästkommande avsnitten beskriva såväl återställningsfas som hämtfas och dessutom ge exempel på exekveringsfaser för några utvalda instruktioner.

8.4.5 Återställningsfasen (RESET)

Återställning förbereds (initieras) då FLIS-processorns insignal Reset aktiveras. Den därpå följande klockpulsen startar återställningen som därefter utförs under tre klockcykler. Processorn genomlöper denna tillståndssekvens, den s.k. återställningssekvensen (RESET-sekvensen) och syftet med sekvensen är att kunna övergå till FETCH-fasen.

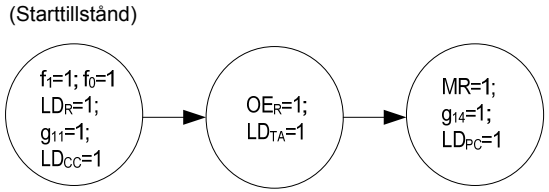
För att kunna inleda en FETCH-fas måste vi förutsätta att den första instruktionen finns att hämta på en definierad adress i minnet. Det är inte lämpligt att bestämma absoluta adresser för den första instruktionen eftersom detta inskränker användbarheten. Vi gör i stället så att den nödvändiga startadressen i sig alltid finns på samma adress i minnet. För FLIS-processorn använder vi den högsta minnesadressen (FF_{16}) för detta ändamål. Eftersom innehållet på denna adress alltid förutsätts innehålla adressen till programmets första instruktion kallas också minnesadressen för *resetvektor*. Utöver detta ska den så kallade I-flaggan i CC, sättas till 1 (vi återkommer till den i ett senare kapitel om *avbrott*).

RESET-sekvensen börjar därför 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 utföras. Det lästa dataordet, som alltså är startadressen för ett program, placeras därför i programräknaren (PC) och styrenheten kan därefter övergå till hämtfasen. Återställningssekvensen tre tillstånd, och de styrsignaler som måste genereras, beskrivs beskrivs i Tabell 8.2. Vi ser här att summatermen inte innehåller någon signal från instruktionsregistret, återställningsfasen utförs alltså oavsett vad som finns i instruktionsregistret.

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler	Kommentarer
Q_0	(Q_0●1)	$(FF)_{16}$→R	f_1=1; f_0=1; LD_R=1; g_{11}=1; LD_{CC}=1;	ALU-funktionen väljs så att talet FF_{16} finns på ALU:ns utgång, dvs funktionskod 3, ALU-funktion = $(0011)_2$. Laddingången på R-registret ettställs så att utvärdet från ALU'n FF_{16} laddas i R-registret vid nästa klockpuls. I-flaggan i CC sätts till 1, övriga flaggor får initialvärden (1,0,0,0)
Q_1	(Q_1●1)	R→TA	OE_R=1; LD_{TA}=1;	Talet FF_{16} i R-registret kopplas ut på bussen. Talet FF_{16} på bussen laddas i temporäradressregistret vid nästa klockpuls.
Q_2	(Q_2●1)	M→PC	MR=1; g_{14}=1; LD_{PC}=1;	Minnesinnehållet på adress FF_{16} läses genom att minnet aktiveras för läsning. Temporäradressregistret adresserar minnet Det dataord som läses placeras i PC vid nästa klockpuls.

TABELL 8.2 RESET-FAS I FLISP

Styrsignalsekvensen i Tabell 8.2 kan också uttryckas av en tillståndsgraf med tre tillstånd i ett sekvensnät som i Figur 8.16. De tre ringarna representerar var sitt (unika) tillstånd. Övergång mellan ringarna verkställs av klockpulser till styrenheten. I varje ring (tillstånd) indikeras vilka styrsignaler som ska aktiveras (ettställas).

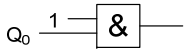


FIGUR 8.16 TILLSTÅNDSGRAF FÖR RESET-FASEN

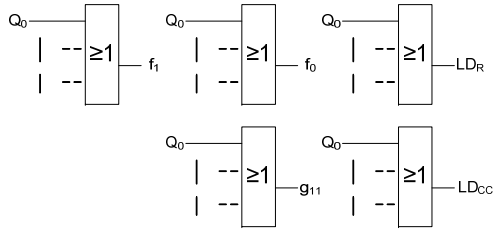
Vi implementerar nu återställningsfasen RESET genom att beskriva logiken som krävs för att generera styrsignalerna till datavägen i tillstånden Q_0, Q_1 och Q_2.

Styrsignaler för tillstånd Q_0

Signalerna som ska aktiveras i tillstånd 0, dvs vid tillståndssignal Q_0 är f_1, f_0, LD_R, g11 och LD_{CC}. Signalerna ska aktiveras oavsett vad som finns i instruktionsregistret. I sådana fall blir AND-villkoret:

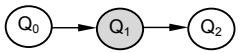


Samma villkor gäller gäller alltså för samtliga signaler varför logiken i styrenheten för detta tillstånd konstrueras enligt Figur 8.17.

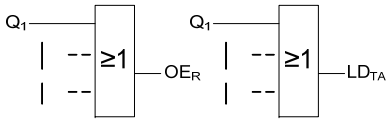


FIGUR 8.17 LOGIK FÖR RESET-FAS, TILLSTÅND 0 I DEN FASTA STYRENHETEN

Styrsignaler för tillstånd Q_1

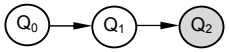


Nästa tillstånd dvs Q_1 kräver att styrsignalerna OE_R och LD_{TA} aktiveras. Logiken för detta visas i Figur 8.18.

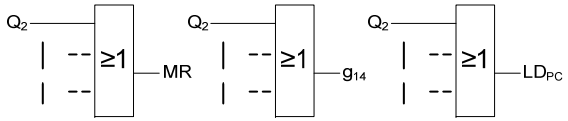


FIGUR 8.18 LOGIK FÖR RESET-FAS, TILLSTÅND 1 I DEN FASTA STYRENHETEN

Styrsignaler för tillstånd Q_2



I återställningsfasens sista tillstånd utförs en minnesläsning för att kopiera innehållet i resetvektorn till register PC (Figur 8.19).



FIGUR 8.19 LOGIK FÖR RESET-FAS, TILLSTÅND 2 I DEN FASTA STYRENHETEN

Med detta har vi implementerat styrenhetens återställningsfas och övergår till hämtfasen.

8.4.6 Hämtfasen (FETCH)

En instruktion utförs genom att operationskoden först hämtas från minnet med en läsoperation och placeras i instruktionsregistret under hämtfasen. Därefter inleds exekveringsfasen vars arbete styrs av den aktuella operationskoden i I-registret. Då hämtfasen utförs förutsätts det att programräknaren PC pekar på den instruktion som står i tur att utföras.

Hos FLISP implementeras hämtfasen med ett enda tillstånd där vi låter PC addressera minnet och samtidigt utför en minnesläsning, data placeras i instruktionsregistret och i samma klockcykel ökas dessutom PC-värdet med 1. Ökningen sker *efter* minnesläsningen så efter hämtfasen innehåller PC antingen adressen till en operand för den instruktion som nu finns instruktionsregistret, eller (om denna instruktion inte har någon operand) innehåller PC adressen till nästa instruktion i programmet.

Utöver detta nollställs också innehållet i temporärrregistret T i varje hämtfas. Detta beror på att registret används vid adressberäkningar tillsammans med registren X,Y eller SP (dock ej PC). För sådana adresseringssätt är det viktigt att T är nollställt vid inledningen av exekveringsfasen.

Skeendet under hämtfasen beskrivs i Tabell 8.3.

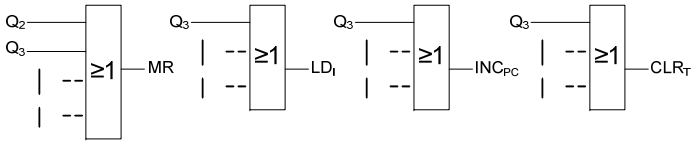
Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler	Kommentarer
Q_3	$(Q_3 \bullet 1)$	M(PC)→I; 0→T;	MR=1; LD_I=1; INC_{PC}=1; CLR_T=1;	Adressen för nästa instruktions operationskod dvs. PC kopplas till minnets adressbuss. Läs operationskoden från minnet och placera i instruktionsregistret I. Adressen som finns i PC ökas med ett. Register för index vid adressberäkningar nollställs.

TABELL 8.3 HÄMT-FAS I FLISP

Vi implementerar hämtfasen FETCH genom att beskriva logiken som krävs för att generera styrsignalerna till datavägen i tillstånd Q_3.

Styrsignaler för tillstånd Q_3

Hämtfasen utförs alltså i ett enda tillstånd. I Figur 8.20 ser vi hur ytterligare ett villkor för MR-signalen nu påförs OR-grinden samtidigt som styrsignalerna LD_I, INC_{PC} och CLR_T försetts med sina första villkor.



FIGUR 8.20 LOGIK FÖR FETCH-FAS, I DEN FASTA STYRENHETEN

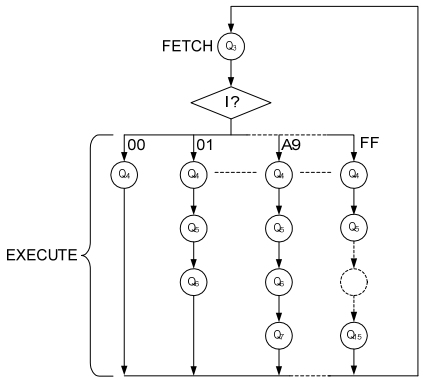
Med detta är vi nu förberedda att ge exempel på utförandefaser för några olika FLISP-instruktioner.

8.4.7 Exekveringsfaser (EXECUTE)

Under utförandefasen, EXECUTE, kommer styrenhetens beteende alltid att bestämmas av såväl operationskoden i instruktionsregistret som tillståndet. För de villkorliga "branch"-instruktionerna tillkommer dessutom flaggvillkor bildade av flaggorna N,V,Z och C från CC-registret.

Operationskoden består av åtta bitar och kan därför ha 256 olika värden. Den representerar alltså 256 möjliga unika exekveringsfaser. Eftersom olika instruktioner har olika komplexitet kommer exekveringsfasernas längd att variera, från den kortaste (*no operation*) som klaras av med endast ett tillstånd (Q_4), till den mest komplicerade (*illegal instruction*) som kräver samtliga tillstånd som räknaren presterar (Q_4 - Q_{15}).

Övergången från hämtfasen till någon av exekveringsfaserna illustreras av den förenklade tillståndsgrafen i Figur 8.21.



FIGUR 8.21 FÖRENKLAD TILLSTÅNDSGRAF FÖR FETCH/EXECUTE FASERNA I STYRENHETEN

8.4.8 Översikt av instruktionsuppsättning

Innan vi går vidare med exempel på styrsignalsekvenser för FLIS-processorns instruktioner ger vi här en mycket kortfattad översikt av den kompletta instruktionslistan. Instruktionerna delas in i grupper med avseende på operationer. Följande instruktionsgrupper finns representerade i FLISP:

”Load/Store”

Instruktioner som används för att överföra data mellan register och minne, `LD` (*load register from memory*) respektive `ST` (*store register into memory*). Instruktionerna `LEA` (*load effective address*) används typiskt för att placera adressen till data i något adressregister.

```
LDA, LDX, LDY, LDSP, LEAX, LEAY, LEASP
STA, STX, STY, STSP
```

Dessutom har instruktionerna `CLRA` (*clear register A*) och `CLR` (*clear memory*) införts för att hantera vanliga specialfall.

”Data movement”

Instruktioner för att överföra data direkt mellan register, `TFR` (*transfer*) kopierar data från ett register till ett annat medan `EXG` (*exchange*) byter innehållen mellan två register.

```
TFR, EXG
```


”Integer arithmetic”

FLISP har instruktioner för två av de fyra räknesätten, addition och subtraktion.

```
ADDA, ADCA, SUBA, SBCA
```

`ADD` (*add*) och `SUB` (*subtract*) utför operationer på 8-bitars tal medan `ADC` (*add with carry*) och `SBC` (*subtract with carry*) används för operationer på tal med godtycklig längd. Första operanden förutsätts finnas i register A, operand två finns på någon minnesposition och reultatet av operationen placeras i register A.

För att minska kodstorlek hos program har även här vanliga operationer getts egna instruktioner:

```
DECA, DEC, INCA, INC
```

`DEC` (*decrement*) för att minska en operand med ett, `INC` (*increment*) för att öka en operand med ett. Slutligen finns även instruktionen `NEG` (*negate*) för att negera en operand.

```
NEGA, NEG
```

”Integer test”

Jämförelser kan göras mellan innehåll i register och minne `CMP` (*compare*), bitvis test kan göras på innehåll i register A, `BITA` (*bitwise test register A*) och test av 8-bitars dataord kan göras med register `TSTA` (*test register A*) eller minne, `TST` (*test operand*).

```
CMPA, CMPX, CMPY, CMPSP, BITA, TSTA, TST
```

”Logical operations”

Gruppen logikoperationer omfattar instruktioner för bitvis manipulation av register A, `ANDA` (*logical and*), `ORA` (*logical or*) och `EORA` (*exclusive or*), bitvis manipulation av CC-registret: `ANDCC` och `ORCC`, samt bitvis komplement av operand `COM` (*complement*).

```
ANDA, ORA, ANDCC, ORCC,
EORA, COMA, COM
```

”Shift/rotate”

Operationerna skift och rotation används för aritmetik och bitmanipulationer. Såväl vänsterskift `LSL` (logical shift left) , `ROL` (rotate left) som högerskift `LSR` (logical shift right), `ASR` (arithmetic shift right) kan utföras. Operanden kan vara antingen i register A eller på någon minnesposition.

```
ASRA, ASR,
LSLA, LSL, LSRA, LSR,
ROLA, ROL, RORA, ROR
```

”Stack operations”

Instruktioner för att spara till/återställa från stacken (*push*/*pull*):

```
PSHA, PSHCC, PSHX, PSHY, PULA, PULCC, PULX, PULY
```

”Program (Flow) control”

```
JMP, JSR, BRA, BSR, B(condition), RTS, RTI
```

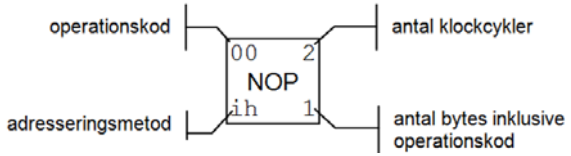
Instruktionerna JMP (*jump*) och BRA (*branch always*) används för en ovillkorlig programflödesändring, JSR (*jump to subroutine*) och BSR (*branch to subroutine*) används tillsammans med RTS (*return from subroutine*) för att modularisera programmet och dela in det i subrutiner. Instruktionen RTI (*return from interrupt*) används i samband med avbrottshantering.

”Misc.”

```
NOP
```

Instruktionen (*no operation*) finns med av historiska skäl. Den kommer till användning huvudsakligen vid test och felsökning i program.

FLIS-processorns instruktioner ges på mycket kortfattad form i Figur 8.22 nedan. Ur denna figur kan man utläsa de väsentliga detaljerna kring en instruktion:



00 2	10 3	20 5	30 3	40 3	50 3	60 3	70 3	80 3	90 2	A0 3	B0 3	C0 3	D0 3	E0	F0 2
NOP	PSHA	BSR	STX	STX	STX	STX	STX	STX	LDX	LDX	LDX	LDX	LDX		LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2		im 2
01 4	11 3	21 4	31 3	41 3	51 3	61 3	71 3	81 3	91 2	A1 3	B1 3	C1 3	D1 3	E1 3	F1 3
ANDCC	PSHX	BRA	STY	STY	STY	STY	STY	STY	LDY	LDY	LDY	LDY	LDY	STA	LDA
im 2	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	ab 2	ab 2
02 4	12 3	22 4	32 3	42 3	52 3	62 3	72 3	82 3	92 2	A2 3	B2 3	C2 3	D2 3	E2 3	F2 3
ORCC	PSHY	BMI	STSP	STSP	STSP	STSP	STSP	STSP	LDSP	LDSP	LDSP	LDSP	LDSP	STA	LDA
im 2	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	ns 2	ns 2
03	13 3	23 4	33 2	43 2	53 4	63 4	73 4	83 4	93 4	A3 5	B3 5	C3 5	D3 5	E3 3	F3 3
	PSHCC	BPL	JMP	RTS	JMP	JMP	JMP	JMP	SBCA	SBCA	SBCA	SBCA	SBCA	STA	LDA
	ih 1	pc 2	ab 2	ih 1	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	nx 2	nx 2
04	14 3	24 4	34 4	44 6	54 5	64 5	74 5	84 5	94 4	A4 5	B4 5	C4 5	D4 5	E4 3	F4 3
	PULA	BEQ	JSR	RTI	JSR	JSR	JSR	JSR	SUBA	SUBA	SUBA	SUBA	SUBA	STA	LDA
	ih 1	pc 2	ab 2	ih 1	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	ax 1	ax 1
05 3	15 3	25 4	35 3	45 3	55 3	65 3	75 3	85 3	95 4	A5 5	B5 5	C5 x	D5 5	E5 4	F5 4
CLRA	PULX	BNE	CLR	CLR	CLR	CLR	CLR	CLR	ADCA	ADCA	ADCA	ADCA	ADCA	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	x+ 1	x+ 1
06 3	16 3	26 4	36 4	46 4	56 4	66 4	76 4	86 4	96 4	A6 5	B6 5	C6 5	D6 5	E6 4	F6 4
NEGA	PULY	BVS	NEG	NEG	NEG	NEG	NEG	NEG	ADDA	ADDA	ADDA	ADDA	ADDA	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	x- 1	x- 1
07 3	17 3	27 4	37 4	47 4	57 4	67 4	77 4	87 4	97 3	A7 4	B7 4	C7 4	D7 4	E7 4	F7 4
INCA	PULCC	BVC	INC	INC	INC	INC	INC	INC	CMPA	CMPA	CMPA	CMPA	CMPA	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	+x 1	+x 1
08 3	18 2	28 4	38 4	48 4	58 4	68 4	78 4	88 4	98 3	A8 4	B8 4	C8 4	D8 4	E8 4	F8 4
DECA	TFR A,C	BCS	DEC	DEC	DEC	DEC	DEC	DEC	BITA	BITA	BITA	BITA	BITA	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	-x 1	-x 1
09 2	19 2	29 4	39 3	49 3	59 3	69 3	79 3	89 3	99 4	A9 5	B9 5	C9 5	D9 5	E9 3	F9 3
TSTA	TFR C,A	BCC	TST	TST	TST	TST	TST	TST	ANDA	ANDA	ANDA	ANDA	ANDA	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	ny 2	ny 2
0A 3	1A 2	2A 4	3A 4	4A 4	5A 4	6A 4	7A 4	8A 4	9A 4	AA 5	BA 5	CA 5	DA 5	EA 3	FA 3
COMA	TFR X,Y	BHI	COM	COM	COM	COM	COM	COM	ORA	ORA	ORA	ORA	ORA	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	ay 1	ay 1
0B 3	1B 2	2B 4	3B 4	4B 4	5B 4	6B 4	7B 4	8B 4	9B 4	AB 5	BB 5	CB 5	DB 5	EB 4	FB 4
LSLA	TFR Y,X	BLS	LSL	LSL	LSL	LSL	LSL	LSL	EORA	EORA	EORA	EORA	EORA	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	y+ 1	y+ 1
0C 3	1C 2	2C 4	3C 4	4C 4	5C 4	6C 4	7C 4	8C 4	9C 3	AC 4	BC 4	CC 4	DC 4	EC 4	FC 4
LSRA	TFR X,S	BGT	LSR	LSR	LSR	LSR	LSR	LSR	CMPX	CMPX	CMPX	LEAX	LEAX	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ns 2	y- 1	y- 1
0D 3	1D 2	2D 4	3D 4	4D 4	5D 4	6D 4	7D 4	8D 4	9D 3	AD 4	BD 4	CD 4	DD 4	ED 4	FD 4
ROLA	TFR S,X	BGE	ROL	ROL	ROL	ROL	ROL	ROL	CMPY	CMPY	CMPY	LEAY	LEAY	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	ny 2	ns 2	+y 1	+y 1
0E 3	1E 2	2E 4	3E 4	4E 4	5E 4	6E 4	7E 4	8E 4	9E 3	AE 4	BE 4	CE 4	DE 4	EE 4	FE 4
RORA	TFR Y,S	BLE	ROR	ROR	ROR	ROR	ROR	ROR	CMPSP	CMPSP	LEASP	LEASP	LEASP	STA	LDA
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	im 2	ab 2	ns 2	nx 2	ny 2	-y 1	-y 1
0F 3	1F 2	2F 4	3F 4	4F 4	5F 4	6F 4	7F 4	8F 4	9F 4	AF 4	BF 4	CF 4	DF	EF	FF
ASRA	TFR S,Y	BLT	ASR	ASR	ASR	ASR	ASR	ASR	EXG A,C	EXG X,Y	EXG X,S	EXG Y,S			
ih 1	ih 1	pc 2	ab 2	ns 2	nx 2	ax 1	ny 2	ay 1	ih 1	ih 1	ih 1	ih 1			

FIGUR 8.22 KARTA MED OPERATIONSKODER FÖR FLISP

Efter denna korta översikt av instruktionsuppsättningen är det nu dags att titta på hur dessa instruktioner kan implementeras i FLIS-processorn.

8.5 Implementering av instruktioner i den fasta styrenheten

I detta avsnitt ger vi en rad exempel på hur olika instruktioner kan implementeras i FLIS-processorns fasta styrenhet. Exemplen är valda för att ge en god bild av samtliga instruktionsgrupper och förekommande adresseringssätt. För att kunna implementera instruktionerna korrekt räcker det naturligtvis inte med den översiktliga information som gavs i det föregående avsnittet, vi måste också ta del av de detaljerade beskrivningar som åtföljer FLIS-processorns dokumentation, vi återger därför utdrag ur denna här i samband med exemplen.

Exempel 8.3 Implementering av utförandefas OP-kod 00, (NOP, No Operation)

Instruktionen NOP specificeras i FLISP's instruktionslista enligt följande:

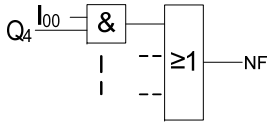
Instruktion **NOP**	Adressering				Operation	Flaggor			
	metod	OP	#	~		N	Z	V	C
NOP	Inherent	00	1	2	No operation	-	-	-	-

Det framgår av instruktionens beskrivning att inget ska göras, inga flaggor påverkas. Det krävs 2 cykler (tillstånd) för att utföra instruktionen, detta omfattar även FETCH-fasen varför denna utförandefas skall klaras av med hjälp av ett tillstånd. Det blir problemfritt eftersom det enda vi behöver göra här är att aktivera styrsignalen NF, dvs hämta nästa instruktion.

Specifikation för implementering i styrenheten blir:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	($Q_4 \bullet I_{00}$)		NF	Instruktion klar, hämta nästa...

Instruktionens bidrag till styrenhetens AND/OR-nät blir därför:



8.5.1 Flaggpåverkan

Väldigt många instruktioner påverkar en eller flera av flaggorna i CC-registret på något sätt. FLISP har totalt 8 olika styrsignaler för att kontrollera påverkan av de enskilda flaggorna i detalj. Varje flagga kan, individuellt, påverkas på ett av följande fyra olika sätt (se även Tabell 8.4):

1. Det nya värdet tas från ALU'n. Detta är det vanligast förekommande eftersom ALU'n konstruerats med detta i åtanke och ofta bildar "rätt" flaggor redan då operationen utförs.
2. Det nya värdet tas från bussen. Detta ger ett verktyg för att behandla flaggorna generellt.
3. Flaggan nollställes, nollställning av flaggor kan på detta sätt utföras betydligt effektivare än den mer generella metoden i punkt 2.
4. Eftersom endast hela CC-registret kan laddas, genom att styrsignalen LD_{CC} aktiveras, måste vi också ha en lösning för de flaggor som eventuellt inte ska modifieras. Lösningen blir att helt enkelt återföra det befintliga värdet vid klockningen.

g_3	g_2	**C** väljs enligt:	RTN
0	0	C tas från ALU'n	ALU(C)→C
0	1	C tas från bit 0 på bussen	b_0→ C
1	0	Återställning (nollställning) av C	0→ C
1	1	C återförs (ändras ej)	

g_5	g_4	**V** väljs enligt:	RTN
0	0	V tas från ALU'n	ALU(V)→V
0	1	V tas från bit 1 på bussen	b_1→ V
1	0	Återställning (nollställning) av V	0→ V
1	1	V återförs (ändras ej)	

g_7	g_6	**Z** väljs enligt:	RTN
0	0	Z tas från ALU'n	ALU(Z)→ Z
0	1	Z tas från bit 2 på bussen	b_2→ Z
1	0	Återställning (nollställning) av C	0→ Z
1	1	Z återförs (ändras ej)	

g_9	g_8	**N** väljs enligt:	RTN
0	0	V tas från ALU'n	ALU(N)→N
0	1	N tas från bit 3 på bussen	b_3→ N
1	0	Återställning (nollställning) av N	0→ N
1	1	N återförs (ändras ej)	

TABELL 8.4 STYRSIGNALER FÖR FLAGGPÅVERKAN

Exempel 8.4 Implementering av utförandefas OP-kod 05_{16}, (CLRA, Clear A)

Instruktionen CLRA specificeras i FLISP's instruktionslista enligt följande:

Instruktion **Clear** Variant	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
CLRA	Inherent	05	1	3	0 → A	0	1	0	0

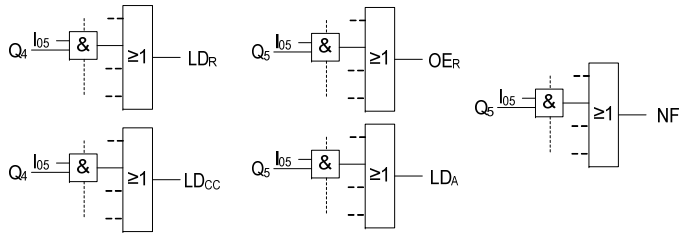
Instruktionen kan utföras under 2 exekveringscykler.

- Under den första av dessa utnyttjar vi enklast ALU'ns funktion för nollställning och överför ALU'ns värde till resultatregistret R. Vi ser också att ALU'ns flaggsättning överensstämmer med specifikationen av instruktionen och kan därför samtidigt överföra flaggorna från ALU'n till CC-registret.
- Under den andra exekveringscykeln överförs resultate från R till register A i datavägen, därmed är instruktionen klar.

Specifikation för implementering i styrenheten blir:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	(Q_4•I_{05})	0→R; ALU(N,V,Z,C) 0→CC;	LD_R; LD_{CC};	Alla ALU'ns f-signaler är 0 vilket ger ALU-funktion "nollställning", värdet 0 placeras därför i R Samtliga flaggor från ALU'n överförs till CC
Q_5	(Q_5•I_{05})	R→A	OE_R; LD_A; NF	Resultat (0) överförs till register A Instruktion klar, hämta nästa...

Styrenhetens logik utökas därför enligt följande:



8.5.2 Läsning från minnet – "LOAD"

De instruktioner som *enbart* överför data från minne till något register, samtidigt som flaggregistret påverkas är:

```
LDA, LDX, LDY, LDSP
```

Implementering är likartad och följande exempel bör studeras noggrannt eftersom det visar den grundläggande metodiken för konstruktion av styrsignalsekvenser.

Exempel 8.5 Implementering av utförandefaser för LDY (Load register Y)
LDY #Data OP-kod 91_{16}
LDY Adr OP-kod $A1_{16}$

Följande utdrag ur instruktionslistan för FLISP ger detaljer om instruktionen LDY och dess olika adresseringssätt:

Instruktion **LD** Variant	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
LDY #Data	Immediate	91	2	2	Data→ Y	Δ	Δ	0	-
LDY Adr	Absolute	A1	2	3	M(Adr) → Y				
LDY n,SP	Indexed	B1	2	3	M(n+SP) → Y				
LDY n,X	Indexed	C1	2	3	M(n+X) → Y				
LDY n,Y	Indexed	D1	2	3	M(n+Y) → Y				

Instruktionen laddar ett värde till register Y, de olika varianterna skiljer sig åt endast med avseende på var ifrån detta värde hämtas. Låt oss för ett ögonblick jämställa alla adresseringssätt, vi kallar operanden helt enkelt för 'Opnd', RTN för operationen blir då;

Opnd → Y;

Flaggsättningen är den samma oavsett variant, vi ser att N och Z ska påverkas av 'Opnd' medan V ska nollställas och C ska lämnas opåverkad. Det bästa sättet är nu att utföra flaggsättningen samtidigt som 'Opnd' finns på bussen, därmed också ALU'ns ingång. Genom att välja funktionskoden (1001) för ALUoperationen

D + C_{in} → U

och låta C_{in} vara 0 får vi operationen D → U. Vi kan då ta flaggorna N och Z direkt från ALU'ns flaggor. Från Tabell 8.4 ovan ser vi nu att g_5 ska vara 1 för att V-flaggan ska nollställas och g_3,g_2 ska båda vara 1 för att C-flaggan ska lämnas opåverkad.

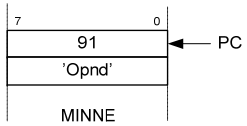
De styrsignaler som åstadkommer korrekt flaggsättning då 'Opnd' finns på bussen kan därför sammanfattas:

f_3; f_1; g_5; g_3; g_2; LD_{CC}

Vi övergår nu till själva dataöverföringen Opnd → Y, vi ser att i det första fallet:

LDY #Data

räcker det med att låta PC adressera minnet, läsa detta värde 'Opnd' och placera det i register Y samtidigt som flaggsättning sker. De styrsignaler som krävs för detta är:

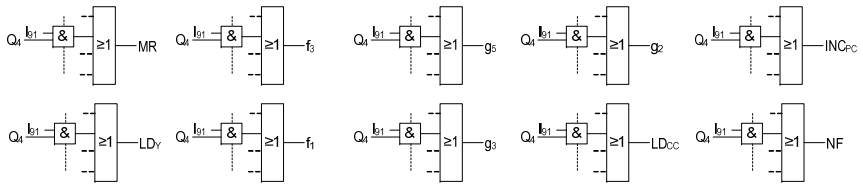


LD_Y; INC_{PC}; MR;

eftersom instruktionen nu är utförd lägger vi också till NF-signalen här och specifikationen för implementering i styrenheten blir:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	(Q_4•I_{91})	M(PC)→Y; ALU(N,V) →CC; 0→CC(V); CC(C) →CC(C); PC+1→PC;	MR; LD_Y; f_3; f_1; g_5; g_3; g_2; LD_{CC}; INC_{PC}; NF	'Opnd' läses direkt från minnet till register Y Flaggsättning sker.. PC uppdateras förbi 'Opnd' Instruktion klar, hämta nästa...

Styrenhetens logik utökas nu enligt följande:



LDY Adr

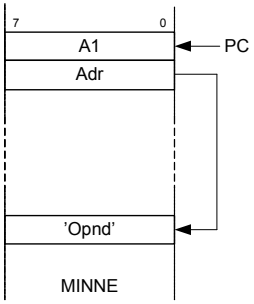
Detta adresseringssätt kräver två läsningar i minnet, först läses adressen till operanden (Adr) och först därefter kan operanden 'Opnd' läsas.

I FLISP har register TA införts för att hantera precis dessa situationer. Med RTN beskriver vi instruktionens operandläsning i två steg:

M(PC) →TA; PC+1→PC,

M(TA) →Y;

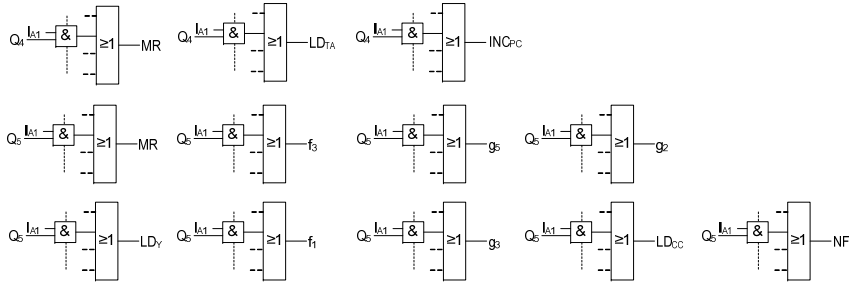
Observera att operanden 'Opnd' som ska påverka flaggregistret nu finns på bussen under den andra exekveringscykeln.



Specifikation för implementering i styrenheten blir:

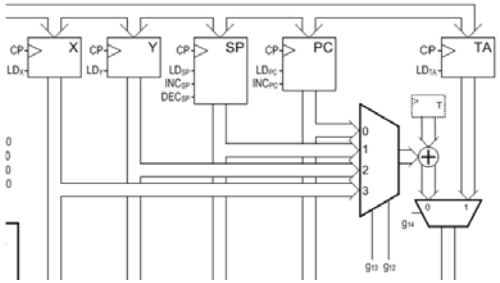
Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	($Q_4 \bullet I_{A1}$)	M(PC)→TA; PC+1→PC;	MR; LD_{TA}; INC_{PC};	'Adr' läses från minnet till TA PC uppdateras förbi 'Adr'
Q_5	($Q_5 \bullet I_{A1}$)	M(TA)→Y; ALU(N,V) →CC; 0→CC(V); CC(C) →CC(C);	MR; LD_Y; f_3; f_1; g_5; g_3; g_2; LD_{CC}; NF	'Opnd' läses nu från minnet till register Y Flaggsättning sker.. Instruktion klar, hämta nästa...

Styrenhetens logik utökas därefter enligt följande:



8.5.3 Adressberäkningsenheten

FLISP har försetts med en enkel adressberäkningsenhet för att underlätta implementeringen av indexerade adresseringssätt. Se Figur 8.23, en väljare används så att något av registren X,Y,SP eller PC kan kopplas direkt till minnets adressbuss. Då något av registren X,Y eller SP väljs kommer också värdet i temporärregister T att adderas till minnesadresseringen.



FIGUR 8.23 ADRESSBERÄKNINGSENHETEN I DATAVÄGEN

g_{14}	g_{13}	g_{12}	Register till adressbuss:	RTN
0	0	0	Register PC	M(PC)
0	0	1	Register SP ("bas") och register T ("offset")	M(T+SP)
0	1	0	Register Y ("bas") och register T ("offset")	M(T+Y)
0	1	1	Register X ("bas") och register T ("offset")	M(T+X)
1	0	0	Adressberäkningsregister, ingen offset	M(TA)
1	0	1	Adressberäkningsregister, ingen offset	M(TA)
1	1	0	Adressberäkningsregister, ingen offset	M(TA)
1	1	1	Adressberäkningsregister, ingen offset	M(TA)

TABELL 8.5 STYRSIGNALER FÖR ADRESSBERÄKNING

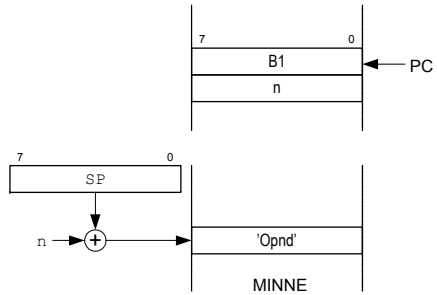
Följande exempel illustrera hur de indexerat adresseringssätt med konstant offset implementeras i FLISP.

Exempel 8.6 Indexerade adresseringssätt, konstant offset
Implementering av utförandefaser för LDY (Load register Y)
LDY n,SP OP-kod $B1_{16}$
LDY n,X OP-kod $C1_{16}$
LDY n,Y OP-kod $D1_{16}$

LDY n,SP

För indexerade adresseringssätt använder vi adress-beräkningsenheten i FLISP. Denna förutsätter att konstanten 'n' som ska adderas till innehållet i något adressregister (SP, X eller Y) finns i temporärregister T. Konstanten 'n' finns omedelbart efter operationskoden så operationen beskrivs med RTN av de båda stegen:

M(PC) →T; PC+1→PC,
M(T+SP) →Y;

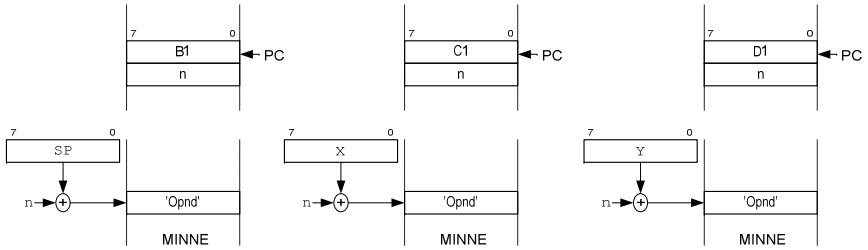


Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	$(Q_4 \bullet I_{B1})$	M(PC)→T; PC+1→PC;	MR; LD_T; INC_{PC};	'n' läses från minnet till T PC uppdateras förbi 'n'
Q_5	$(Q_5 \bullet I_{B1})$	M(T+SP)→Y; ALU(N,V) →CC; 0→CC(V); CC(C) →CC(C);	MR; g_{12}; LD_Y; f_3; f_1; g_5; g_3; g_2; LD_{CC}; NF	'Opnd' läses nu från minnet till register Y via register SP Flaggsättning sker.. Instruktion klar, hämta nästa...

LDY n,X

LDY n,Y

För implementeringen av de sista båda adresseringssätten är det lämpligt att betrakta likheterna hos de indexerade adresseringssätten:



Vi ser att de är identiska i strukturen och skiljer sig åt endast i operationskod och adressregister. Vi kan därför snabbt sluta oss till

LDY n,X

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	$(Q_4 \bullet I_{C1})$	M(PC)→T; PC+1→PC;	MR; LD_T; INC_{PC};	'n' läses från minnet till T PC uppdateras förbi 'n'
Q_5	$(Q_5 \bullet I_{C1})$	M(T+X)→Y; ALU(N,V) →CC; 0→CC(V); CC(C) →CC(C);	MR; g_{13}; g_{12}; LD_Y; f_3; f_1; g_5; g_3; g_2; LD_{CC}; NF	'Opnd' läses nu från minnet till register Y via register X Flaggsättning sker.. Instruktion klar, hämta nästa...

LDY n,Y

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	$(Q_4 \bullet I_{B1})$	M(PC)→T; PC+1→PC;	MR; LD_T; INC_{PC};	'n' läses från minnet till T PC uppdateras förbi 'n'
Q_5	$(Q_5 \bullet I_{B1})$	M(T+Y)→Y; ALU(N,V) →CC; 0→CC(V); CC(C) →CC(C);	MR; g_{13}; LD_Y; f_3; f_1; g_5; g_3; g_2; LD_{CC}; NF	'Opnd' läses nu från minnet till register Y via register Y Flaggsättning sker.. Instruktion klar, hämta nästa...

Avslutningsvis implementeras instruktionerna i styrenheten genom att komplettera AND/OR-logiken.

8.5.4 Skrivning till minne – ”STORE”

De instruktioner som *enbart* överför data från något register till minne, är:

`STA, STX, STY, STSP`

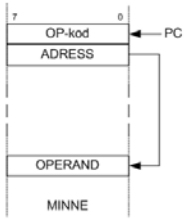
Flaggregistret påverkas inte av dessa instruktioner.

Exempel 8.7 Implementering av utförandefaser för
STY Adr (Store register Y)
STY n,SP

Följande utdrag ur instruktionslistan ger detaljerna om de olika varianterna av STY-instruktionen:

Instruktion **ST** Variant	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
`STY Adr`	Absolute	31	2	3	Y → M(Adr)	-	-	-	-
`STY n,SP`	Indexed	41	2	3	Y → M(n+SP)				

Det sker ingen flaggsättning.



STY Adr

I tillstånd Q_4 läses adressen och placeras i register TA.
I tillstånd Q_5 skrivs innehållet i register Y till adressen.

Q4: M(PC)→TA; PC+1→PC
Q5: Y→M(TA)

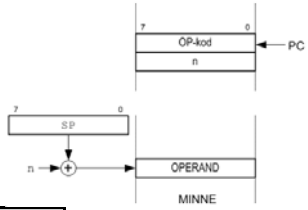
Styrsignalsekvensen blir då:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	$(Q_4 \bullet I_{31})$	M(PC)→TA; PC+1→PC;	MR; LD_{TA}; INC_{PC};	ADRESS läses från minnet till TA PC uppdateras förbi ADRESS
Q_5	$(Q_5 \bullet I_{31})$	Y→ M(TA);	MW; g_{14}; OE_Y; NF	register Y skrivs till minnet Instruktion klar, hämta nästa...

STY n,SP

I tillstånd Q_4 läses konstanten ’n’ och placeras i register T.
I tillstånd Q_5 skrivs innehållet i register Y till adressen som ges av T+SP.

Q4: M(PC)→T; PC+1→PC
Q5: Y→M(T+SP)



Styrsignalsekvensen blir då:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	$(Q_4 \bullet I_{41})$	M(PC)→T; PC+1→PC;	MR; LD_T; INC_{PC};	'n' läses från minnet till T PC uppdateras förbi 'n'
Q_5	$(Q_5 \bullet I_{41})$	Y→ M(T+SP);	MW; g_{12}; OE_Y; NF	register Y skrivs till minnet Instruktion klar, hämta nästa...

8.5.5 Kombinerad LOAD/STORE – ”READ/MODIFY/WRITE”

Av prestandaskäl, för att minska kodstorleken hos program, kan vanligt förekommande unära operationer utföras direkt i minnet. Instruktionerna kallas därför allmänt för *read/modify/write*. Utförandefasen delas då in i tre steg:

1. Operandens adress bestäms, detta kan innebära att adressen placeras i register TA, eller, om indexerad adressering används att en offset placeras i register T.
2. Operationen utförs i ALU:n och resultatet sparas i register R, under detta steg sker också flaggsättningen.
3. Resultatet i register R skrivs tillbaks till operandens adress.

Exempel 8.8 Implementering av utförandefas för NEG Adr (Negate)

Operationen ”byter tecken” på operanden:

Instruktion **NEG** Variant	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
NEG Adr	Absolute	36	2	4	– M(Adr) → M(Adr)	Δ	Δ	Δ	Δ

Adressen läses från minnet och placeras i TA där den måste finnas både vid operandläsning och då resultatet skrivs tillbaks:

1. M(PC)→TA; PC+1→PC

Operationen utförs och flaggorna sätts:

2. $M(TA)_{1k}$ + 1→R; ALU(CC)→CC

Resultatet skrivs tillbaks till operandens adress i minnet

3. R→ M(TA);

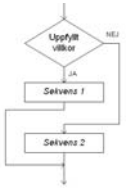
Den resulterand styrsignalsekvensen kan nu bestämmas:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	(Q_4●I_{36})	M(PC)→TA; PC+1→PC;	MR; LD_{TA}; INC_{PC};	'Adr' läses från minnet till TA PC uppdateras förbi 'Adr'
Q_5	(Q_5●I_{36})	$M(TA)_{1k}$ + 1→R; ALU(CC)→CC	MR;g_{14};f_2;f_0;g_0; LD_R; LD_{CC}	Läsning och tvåkomplementering flaggsättning
Q_6	(Q_6●I_{36})	R→ M(TA);	MW; g_{14}; OE_R; NF	register R skrivs till minnet Instruktion klar, hämta nästa...

8.6 Kontroll av programflöde

Vi har så här långt sett hur vår automatiska styrenhet utför ett sekvensiellt lagrat program. Det måste dock finnas olika sätt att påverka själva programflödet på. I vanliga programspråk finns konstruktioner som kräver programflödesändringar i olika former. Betrakta följande programkonstruktion utgörande ett *val*. Baserat på en test av något villkor ska antingen ”Sekvens 1” eller ”Sekvens 2” utföras. Varje ”box” i fuguren i utgörs av en sekvens maskininstruktioner hos datorn.

```
if ( Uppfyllt villkor ) then
   do Sekvens 1;
else
   do Sekvens 2;
end if;
```



FIGUR 8.24 VALKONSTRUKTION FÖR PROGRAMFLÖDESKONTROLL

Av flödesgrafen i Figur 8.24 framgår att programflödet måste kunna ändras, från villkorstesten, om villkoret inte är uppfyllt, till ”Sekvens 2”, eller från slutet av ”Sekvens 1” till omedelbart efter ”Sekvens 2”. Det är dessutom klart att vi behöver kunna utföra såväl villkorliga som ovillkorliga programflödesändringar. Figuren visar exempelvis en sekvens av instruktioner som utför någon test och där den sista instruktionen används för att välja programflödet baserat på hur den tidigare testoperationen utfallit (”Uppfyllt villkor”). Detta kallar vi en *villkorlig programflödesändring*. Sist i instruktionssekvensen ”Sekvens 1” ser vi hur programflödet ska ändras till efter ”Sekvens 2”, alltså en *ovillkorlig programflödesändring*.

8.6.1 Ovillkorlig programflödesändring

Ovillkorlig programflödesändring sker genom att PC laddas med adressen till den instruktion där exekveringen ska återupptas. Instruktionen förekommer i olika varianter `JMP` (*jump*) och `BRA` (*branch always*) och som skiljer sig åt genom att operanden, dvs. destinationsadressen, kodas på olika sätt.

JMP Jump

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

Detaljer:

Instruktion **JMP** Variant	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
JMP Adr	Absolute	33	2	2	Adr → PC	-	-	-	-
JMP n,X	Indexed	53	2	4	n+X → PC				
JMP A,X	Indexed	63	1	4	A+X → PC				
JMP n,Y	Indexed	73	2	4	n+Y → PC				
JMP A,Y	Indexed	83	1	4	A+Y → PC				

Exempel 8.9 Implementering av utförandefas för instruktionen JMP ”Jump”
JMP Adr OP-kod 33_{16}
JMP n,Y OP-kod 73_{16}
JMP A,Y OP-kod 83_{16}

`JMP Adr`

Det räcker med en cykel för exekveringsfasen, där operanden, dvs. destinationsadressen, läses från minnet direkt till register PC.

M(PC)→PC

Styrsignalerna blir:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	(Q_4●I_{33})	M(PC)→PC;	MR; LD_{PC}; NF	'Adr' läses från minnet till PC Instruktion klar, hämta nästa...

För de indexerade adresseringssätten måste vi använda ALU:n för att addera offseten till innehållet i basregistret:

1. Offseten, detta kan vara en konstant som läses från minnet, eller innehållet i register A, placeras i register T, som förberedelse.
2. Basregistret (X eller Y) läggs ut i datavägen, addition utförs i ALU:n och resultatet sparas i register R.
3. Resultatet i register R skrivs till PC.

JMP n,Y

Implementeringen följer direkt av ovanstående beskrivning:

Q_4: M(PC)→T;
Q_5: T+Y→R;
Q_6: R→PC

Den resulterande styrsignalsekvensen kan nu bestämmas:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	(Q_4•I_{73})	M(PC)→T;	MR; LD_T;	'n' läses från minnet till T
Q_5	(Q_5•I_{73})	T+Y→R;	OE_Y; f_3;f_1;f_0; LD_R;	adressberäkning
Q_6	(Q_6•I_{73})	R→ PC;	OE_R; LD_{PC}; NF	register R skrivs till PC Instruktion klar, hämta nästa...

JMP A,Y

Implementeringen av denna variant skiljer sig från den föregående endast i offseten, det räcker därför med att modifiera i första tillståndet:

Q_4: A→T;

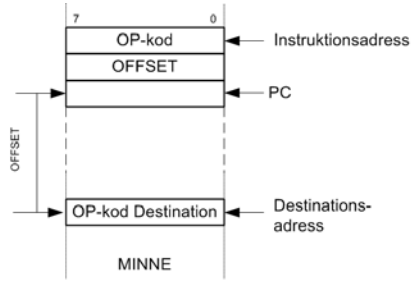
Den resulterande styrsignalsekvensen blir då:

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler (=1)	Kommentarer
Q_4	(Q_4•I_{83})	A→T;	OE_A; LD_T;	offset läses från A till T
Q_5	(Q_5•I_{83})	T+Y→R;	OE_Y; f_3;f_1;f_0; LD_R;	adressberäkning
Q_6	(Q_6•I_{83})	R→ PC;	OE_R; LD_{PC}; NF	register R skrivs till PC Instruktion klar, hämta nästa...

8.6.2 PC-relativ adressering

En alternativ form av ovillkorlig programflödesändring utnyttjar så kallad programräknar-relativ adressering (PC-relativ). Destinationens läge anges här med en offset, snarare än en absolut adress. Offseten bestäms som skillnaden mellan destinationsadressen och instruktionen. Eftersom PC vid adressberäkningen ”pekar på” instruktionen som omedelbart följer efter den PC-relativa instruktionen får vi sambandet:

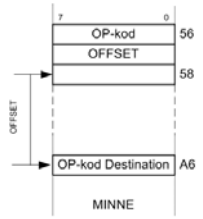
Destinationsadress = Instruktionsadress + 2 + OFFSET.



FIGUR 8.25 PC-RELATIV ADRESSERING

Exempel 8.10 PC-relativ offsetberäkning

Bestäm offset för en instruktion med PC-relativ adressering på adress $(56)_{16}$, och destinationsadress $(A6)_{16}$



Lösning:

Offset = Destinationsadress – (Instruktionsadress+2)=
A6 – (56+2) = 4E

Exempel 8.11 Utförandefasen för instruktionen ”Branch Always”

BRA Branch always

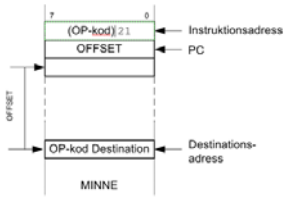
RTN	PC+Offset → PC
Flaggor	Påverkas ej
Beskrivning	Ett hopp utförs till adressen ADRESS = PC+Offset. Offset räknas från adressen efter branchinstruktionen, dvs vid uträkningen av hoppadressen pekar PC på operationskoden som (eventuellt) finns direkt efter branchinstruktionen i minnet

Detaljer:

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

Då exekveringsfasen inleds innehåller PC adressen till OFFSET, RTN för hela instruktionen kan därför skrivas:

PC+1+(PC)→PC



Vi delar upp detta i en styrsignalsekvens genom att först placera PC i T-registret för den kommande beräkningen och samtidigt placera PC i register TA för inläsning av OFFSET från minnet. Observera att värdet för PC i register T nu är 1 mindre än det värde som *ska* användas vid beräkningen av destinationsadressen:

PC→T; PC→TA

Därefter utförs själva adressberäkningen med hjälp av ALU:n på samma sätt som i Exempel 8.9. Vi kompenserar nu värdet för PC med att addera även konstanten 1 till den resulterande destinationsadressen och resultatet placeras i register R:

M(TA)+T+1→R

Slutligen återförs resultatet till PC, varefter instruktionen är utförd:

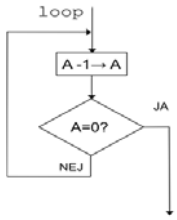
R→PC

Den resulterande styrsignalsekvensen blir därför:

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	(Q_4•I_{21})	PC→T; PC→TA	OE_{PC}; LD_T; LD_{TA}	Operandhämtning
Q_5	(Q_5•I_{21})	M(TA)+T+1→R	MR;f_3; f_1; f_0;g_0;g_{14} LD_R	Adressberäkning
Q_6	(Q_6•I_{21})	R→PC	OE_R; LD_{PC}; NF	Resultatet återförs till PC

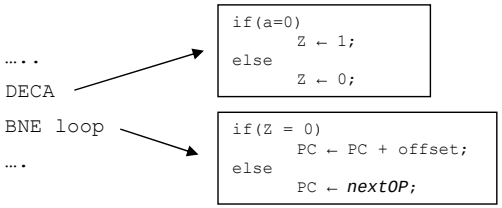
8.6.3 Villkorlig programflödesändring

Villkorlig programflödesändring använder flaggorna i CC-registret för att bestämma om en programflödesändring ska utföras, eller inte. Vi kan välja att testa enskilda flaggor men även olika flaggkombinationer. Den automatiska styrenheten omfattar därför även kombinatorik som tillhandahåller dessa flaggkombinationer.



Låt oss illustrera villkorlig programflödesändring med ett fall där vi tänker oss att vi testar innehållet i register A. Vi har tidigare sett hur DECA minskar innehållet i register A och därefter påverkar Z-flaggan i CC. En villkorlig operation, baserad på Z-flaggan, kan därför användas omedelbart efter DECA-instruktionen. Den villkorliga instruktionen, vi kallar den ”branch if not equal” (BNE) , ska resultera i en programflödesändring om Z-flaggan är 0, dvs ”not-equal”-villkor är uppfyllt. Om villkoret (Z=0) inte är uppfyllt ska i stället programexekveringen fortsätta med nästa påföljande instruktion, dvs. som om ingenting hade hänt.

Programkonstruktionen sätts samman av två instruktioner, där den första instruktionen påverkar flaggan Z och den andra instruktionen testar flaggan. Detta innebär att vi inte utan vidare kan placera ytterligare instruktioner mellan dessa.



Uttrycket ”Z =0” kallar vi villkorsindikator och uttryckets värde uppfattas som ”sant” om villkoret är uppfyllt. Om villkorsindikatorn är sann ska flödesändringen utföras, annars ska instruktionen inte göra någonting, utan exekveringen fortsätter med instruktionen omedelbart efter branch-instruktionen. Vid en jämförelse med *branch always*-instruktionen i Exempel 8.11 ser vi att den första fasen kan göras identisk:

PC→T; PC→TA;

I nästa fas sker beräkning av destinationsadress (i fall villkorsindikator är sann) precis som tidigare, vi kan samtidigt förbereda för sekventiell exekvering (i fall villkorsindikatorn är falsk) genom att öka PC med 1:

M(TA)+T+1→R;
PC+1→PC

I den avslutande fasen ska PC laddas med destinationsadressen (från R) om villkorsindikatorn är sann, annars ska PC lämnas eftersom registret ju nu innehåller adressen till nästa instruktion.

if(Villkorsindikation)
R→PC

Styrsignal för denna konstruktion blir:

LD_{PC} = Villkorsindikator

Det finns totalt 14 olika villkorsindikatorer som baseras på olika typer av test och jämförelser. Alla villkorliga instruktioner kan utformas på samma sätt där endast villkorsindikatorn skiljer sig. För test av en operand kan villkorsindikatorer i form av enkla flaggtest användas, se Tabell 8.6 nedan, av tabellen framgår att exekveringsfaserna skiljer sig åt endast i användande av villkorsindikatorer i form av flaggor N,V,C och Z samt flaggornas inverser.

Instruktion (Mnemonic)	Funktion	Villkorsindikation
“Branch if carry set” (BCS)	“Hopp” om *carry*	C=1
“Branch if carry clear” (BCC)	“Hopp” om ICKE *carry*	C=0
“Branch if equal” (BEQ)	“Hopp” om *zero*	Z=1
“Branch if not equal” (BNE)	“Hopp” om ICKE *zero*	Z=0
“Branch if minus” (BMI)	“Hopp” om *negative*	N=1
“Branch if plus” (BPL)	“Hopp” om ICKE *negative*	N=0
“Branch if overflow set” (BVS)	“Hopp” om *overflow*	V=1
“Branch if overflow clear” (BVC)	“Hopp” om ICKE *overflow*	V=0

TABELL 8.6 ENKLA FLAGGTEST

Exempel 8.12 Implementering av utförandefas OP-kod $(25)_{16}$, (BNE <Adr>, Branch on Not Equal)

Instruktionen beskrivs av:

Instruktion BNE	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
BNE Adr	Relativ	25	2	4	if(Z=0) PC+Offset → PC	-	-	-	-

Villkorsindikatorn, i detta fall Z=0, realiseras med det boolska uttrycket !Z, dvs. inversen av Z-biten från CC-registret, styrsignalsekvensen blir därför:

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	$(Q_4 \bullet I_{25})$	PC→T; PC→TA	OE_{PC}; LD_T; LD_{TA}	Placera adress till offset i TA och PC i T.
Q_5	$(Q_5 \bullet I_{25})$	M(TA)+T+1→R; PC+1→PC	MR;f_3; f_1; f_0;g_0;g_{14};LD_R; INC_{PC}	Adressberäkning, destinationsadress till R Uppdatera PC till att peka på nästa instruktion
Q_6	$(Q_6 \bullet I_{25})$	if(Z=0) R→PC	OE_R; LD_{PC} =!Z; NF	Om Z=0 återförs destinationsadressen till PC annars hämtas nu nästa instruktion

Relationsoperatorer används för att beskriva jämförelse av två tal:

> ”större än...”
≥ “större än eller lika med ...
< ”mindre än...”
≤ “mindre än eller lika med...”

Instruktioner för jämförelse (compare) utförs med en operand (vänsterled) i något register (A,X,Y eller SP) och den andra operanden (högra ledet) i minnet. Vi låter R beteckna något av de tillåtna registren och analyserar de möjliga utfallen i följande tabell där vi åskådliggör hur operationen R-M kan resultera i tre olika utfall och hur dessa utfall i sin tur påverkar villkorsflaggorna. Vi börjar med en betraktelse av tal utan inbyggt tecken, dvs. flaggorna C och Z.

R-M	C	Z
R<M	1	x
R≥M	0	x
R>M	0	0

I tabellen till vänster ser vi att ”mindre än”-villkoret indikeras av att C=1. Eftersom C och Z inte kan vara 1 samtidigt räcker det här med att använda C som villkorsindikator. Detta betyder samtidigt att det komplementära villkoret, ”större än eller lika med” indikeras korrekt av att C=0.

Villkoret ”större än” indikeras av att både C och Z är 0. Motsvarande komplementära villkor, ”mindre än eller lika med” indikeras följaktligen av att någon av C eller Z är 1.

Dessa resultat sammanfattas som instruktioner i Tabell 8.7.

Instruktion (Mnemonic)	Relation	Villkorsindikation
”Branch if higher” (BHI)	R>M	C + Z = 0
”Branch if higher or same” (BHS)	R≥M	C=0
”Branch if lower” (BLO)	R<M	C=1
“Branch if lower or same” (BLS)	R≤M	C + Z = 1

TABELL 8.7 TEST AV TAL UTAN TECKEN

Exempel 8.13 Implementering av utförandefas OP-kod $(2A)_{16}$, (BHI <Adr>, Branch if HIgher)

Instruktionen beskrivs av:

Instruktion **BHI**	Adressering				Operation	Flaggor			
	metod	OP	#	~		N	Z	V	C
BHI Adr	Relativ	2A	2	4	If (C+Z = 0): PC+Offset → PC	-	-	-	-

Villkorsindikatorn, i detta fall C+Z=0, realiseras med det boolska uttrycket !(C+Z), styrsignalsekvensen blir därför:

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	$(Q_4 \bullet I_{2A})$	PC→T; PC→TA	OE_{PC}; LD_T; LD_{TA}	Placera adress till offset i TA och PC i T.
Q_5	$(Q_5 \bullet I_{2A})$	M(TA)+T+1→R; PC+1→PC	MR;f_3; f_1; f_0;g_0;g_{14};LD_R; INC_{PC}	Adressberäkning, destinationsadress till R Uppdatera PC till att peka på nästa instruktion
Q_6	$(Q_6 \bullet I_{2A})$	if(C+Z=0) R→PC	OE_R; LD_{PC} =!(C+Z); NF	Om Z=0 återförs destinationsadressen till PC annars hämtas nu nästa instruktion

R-M	N	Z
R<M	1	x
R≥M	0	x
R>M	0	0

För villkorsindikatorer för tal med inbyggt tecken kan vi i princip ersätta C-flaggan med N-flaggan, se tabellen till vänster. Sådana villkorsindikatorer skulle fungera för alla fall där tvåkomplementsspill inte uppstår (V=0). I de fall där V=1 blir det dock fel eftersom detta ju är en indikator för att N-flaggan är fel.

Vi kan hantera situationen genom att helt enkelt testa verkliga N då V=0 men i stället testa inversen av N då V=1. Den booleska funktionen N ⊕ V gör just detta. Villkorsindikatorer för relationer mellan tal med inbyggt tecken och deras tillhörande instruktioner sammanfattas i Tabell 8.8 där vi helt enkelt ersatt C med N ⊕ V .

Instruktion (Mnemonic)	Relation	Villkorsindikation
“Branch if greater than” (BGT)	R>M	(N⊕V) + Z = 0
“Branch if greater or equal” (BGE)	R≥M	N⊕V = 0
“Branch if less than” (BLT)	R<M	N⊕V = 1
“Branch if less or equal” (BLE)	R≤M	(N⊕V) + Z = 1

TABELL 8.8 TEST AV TAL MED TECKEN

Exempel 8.14 Implementering av utförandefas OP-kod $(2C)_{16}$, (BGT <Adr>, Branch if Greater Than)

Instruktionen beskrivs av:

Instruktion **BGT**	Adressering				Operation	Flaggor			
	metod	OP	#	~		N	Z	V	C
BGT Adr	Relativ	2C	2	4	If ((N⊕V) + Z = 0) PC+Offset → PC	-	-	-	-

Villkorsindikatorn, i detta fall (N⊕V) + Z = 0, realiseras med det boolska uttrycket !((N⊕V) + Z), styrsignalsekvensen blir därför:

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	$(Q_4 \bullet I_{2C})$	PC→T; PC→TA	OE_{PC}; LD_T; LD_{TA}	Placera adress till offset i TA och PC i T.
Q_5	$(Q_5 \bullet I_{2C})$	M(TA)+T+1→R; PC+1→PC	MR;f_3; f_1; f_0;g_0;g_{14};LD_R; INC_{PC}	Adressberäkning, destinationsadress till R Uppdatera PC till att peka på nästa instruktion
Q_6	$(Q_6 \bullet I_{2C})$	if((N⊕V) + Z = 0) R→PC	OE_R; LD_{PC}= !((N⊕V) + Z); NF	Om Z=0 återförs destinationsadressen till PC annars hämtas nu nästa instruktion

8.6.4 Stackoperationer

Vi introducerade tidigare, i avsnitt 8.2.2, begreppet "stackfunktion". Stacken används för tillfällig undanlagring, typiskt i sådana fall där processorns register inte räcker till av någon anledning. Stacken är en central mekanism som vi kommer att återkomma till i flera olika sammanhang. I detta avsnitt koncentrerar vi oss på hur syrsignalsekvenser, för de olika stackoperationerna, kan konstrueras.

Stackoperationerna *push* och *pull* i FLISP, är definierade så att de använder register SP. *push*-operationen definieras av RTN-beskrivningen:

SP-1→SP
R→M(SP)

dvs. stackpekaren SP minskas först med 1 för att skapa utrymme i minnet, därefter skrivs värdet från ett register, R kan vara något av A,X eller Y, till denna minnesposition. Detta innebär alltså att SP alltid pekar på det senast lagrade värdet. Den omvända operationen , *pull*, utförs i "spegelvänd" ordning och definieras av RTN-beskrivningen:

M(SP)→R
SP+1→SP

Funktionen kallas i bland *LIFO* (*last in first out*) .

Exempel 8.15 Utförandefaser för instruktionerna PSHY och PULY

push-operationerna definieras enligt följande:

PSH Push register on Stack

RTN	SP–1 → SP R → M(SP)
Flaggor	Påverkas ej
Beskrivning	Stackpekaren uppdateras först. Angivet registerinnehåll skrivs sedan på stacken

Detaljer:

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

Styrsignalsekvens för PSHY blir därför:

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	$(Q_4 \bullet I_{12})$	SP-1→SP	DEC_{SP}	Minska stackpekare
Q_5	$(Q_5 \bullet I_{12})$	Y→M(SP)	OE_Y; g_{12};MW; NF	Skriv register Y till stack

pull-operationerna definieras enligt:

PUL Pull register from stack

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

Detaljer:

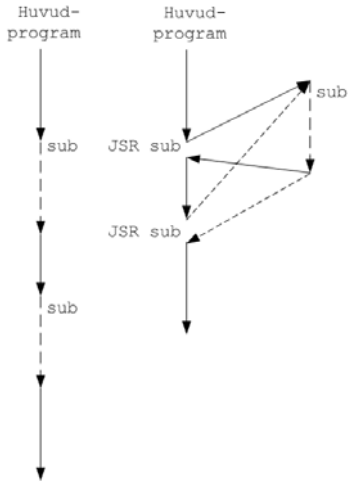
Instruktion PUL Variant	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
PULA	Inherent	14	1	3	M(SP) → PC, SP+1 → SP	-	-	-	-
PULX	Inherent	15	1	3	M(SP) → X SP+1 → SP	-	-	-	-
PULY	Inherent	16	1	3	M(SP) → Y SP+1 → SP	-	-	-	-
PULCC	Inherent	17	1	3	M(SP) → CC SP+1 → SP	Δ	Δ	Δ	Δ

Styrsignalsekvens för PULY blir därför:

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	$(Q_4 \bullet I_{16})$	M(SP) → X	LD_Y; g_{12};MR	Läs register Y från stack
Q_5	$(Q_5 \bullet I_{16})$	SP+1 → SP	INC_{SP}; NF	Öka stackpekare

8.6.5 Modularisering i subrutiner

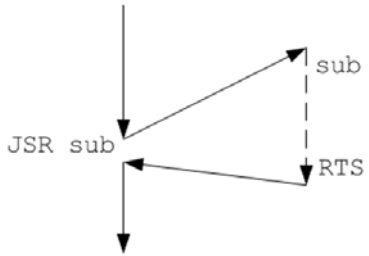
För att spara minnesutrymme är det lämpligt att koda uppgifter som utförs ofta i programmet i form av subrutiner. Om vi tänker oss ett program, dvs. en sekvens av instruktioner, som utför ett *huvudprogram*, och att i denna sekvens finns det delsekvenser av instruktioner som upprepas flera gånger, kalla en sådan delsekvens *sub* och betrakta Figur 8.26.



FIGUR 8.26 ILLUSTRATION AV ”IN-LINE” KODNING RESPEKTIVE SUBRUTIN

Till vänster i Figur 8.26 visas hur delskevensen *sub* repeteras varje gång den ska utföras, vi säger då att delsekvensen *sub* kodas ”in-line”. Ett alternativt sätt, är att, som till höger i figuren, placera en instans av delsekvensen på någon annan plats i minnet och därefter utföra sekvensen via ”anrop” från huvudprogrammet, vi säger då att vi kodat delsekvensen *sub* som en *subrutin*. Detta illustreras till höger i figuren, fördelen med detta är uppenbar, i stället för att repetera samma kod många gånger skapar vi i stället instruktioner för att anrop av en subrutin, `JSR` (*jump to subroutine*).

Ett subrutinanrop kan ske från praktiskt taget godtyckliga adresser i huvudprogrammet och det krävs då att information om *återhoppsadressen*, dvs. den adress där programexekveringen ska återupptas efter subrutinen, finns tillgänglig då subrutinen avslutas. Innan programräknaren PC ges adressen till subrutinens första instruktion måste registret alltså sparas och vi såg i förra avsnittet hur sådan tillfällig undanlagring av registerinnehåll kan göras med användning av stacken. För att återställa PC till återhoppsadressen skapar i instruktionen `RTS` (return from subroutine), se Figur 8.27



FIGUR 8.27 SUBRUTINANROP JSR/RTS

Exempel 8.16 Utförandefaser för instruktionerna Jump to Subroutine och Return from subroutine

Detaljer för JSR:

Instruktion **JSR** variant	Adressering metod	OP	#	~	Operation	Flaggor N	Z	V	C
`JSR Adr`	Absolute	34	2	4	SP–1 → SP PC → M(SP) Adr → PC	-	-	-	-

I utförandefasens första klockcykel läses destinationadressen genom ALU:n till register R under den sista klockcykeln ska sedan denna adress överföras till PC. Samtidigt uppdateras PC.

M(PC)→R;
PC+1→PC

Efter uppdateringen pekar nu PC på nästa sekvensiella instruktion, dvs. PC innehåller *återhoppsadressen*, vilken nu ska läggas på stacken. Denna operation måste delas upp i två klockcykler: först minskas SP med 1, därefter skrivs PC till minnet:

SP–1→SP,
PC→M(SP)

Slutligen överförs destinationsadressen till PC, varefter instruktionen är slutförd.

R →PC

Styrsignalsekvensen blir då:

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	$(Q_4 \bullet I_{34})$	M(PC)→R; PC+1→PC; SP-1→SP;	MR; f_3; f_0; LD_R; INC_{PC}; DEC_{SP};	Adress från minnet till register R Uppdatera PC Minska stackpekare
Q_5	$(Q_5 \bullet I_{34})$	PC→M(SP)	OE_{PC}; g_{12}; MW	Lägg PC på stacken
Q_6	$(Q_6 \bullet I_{34})$	R →PC	OE_R; LD_{PC} ; NF	Adress från register R till PC

RTS

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

Destinationsadressen förutsätts nu ligga överst på stacken. Under utförandefasens första klockcykel läses denna till PC:

M(SP) →PC;

Under den andra klockcykeln återställs stackpekarens värde till det den hade precis före subrutinanropet:

SP+1→SP

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q_4	$(Q_4 \bullet I_{43})$	M(SP) →PC; SP+1→SP;	MR; g_{12}; LD_{PC}; INC_{SP}; NF	Adress från minnet, överst på stacken, till register PC Återställ stackpekare

Exempel 8.17 Användning av instruktionerna JSR och RTS

Detta exempel illustrerar hur ett subrutinanrop och återhopp sker. Det förutsätts att ”huvudprogrammet” placerats med start på adress 30_{16} i minnet, att subrutininen, som bara innehåller en NOP-instruktion (OP-kod 00) och en RTS-instruktion (OP-kod 43) placerats på adress 53_{16}. Minnesutrymmet 10_{16} t.o.m $1F_{16}$ har reserverats för stack.

För att utföra huvudprogrammet placeras nu först dess startadress (30_{16}) i PC, därefter placeras adressen 20_{16} i SP. Observera att denna adress egentligen inte används för stacken eftersom en *push*-operation alltid inleds med att minska SP med 1. Den först använda adressen i stacken (TOS) blir därför $1F_{16}$. Figuren längst till vänster illustrerar situationen efter dessa operationer.

Huvudprogrammets enda instruktion är `JSR` 53_{16}, och när denna utförts har vi den situation som illustreras av den mellersta figuren, SP har minskats med 1, och överst på stacken finns återhoppsadressen (32_{16}), PC har fått värdet 53_{16}, dvs. adressen till subrutinens första instruktion.

Slutligen, i figuren längst till höger, illustreras situationen efter det att båda instruktionerna i subrutinen utförts. PC har nu fått värdet som ligger överst på stacken, dvs återhoppsadressen 32_{16}, därefter har SP ökats med 1 och därmed återfått värdet registret hade före subrutinanropet.